Sourabh Sharma , Rajesh RV
David Gonzalez

Microservices: Building Scalable Software

Learning Path

Discover how to easily build and implement scalable microservices from scratch





Packt

Microservices: Building Scalable Software

Table of Contents

Microservices: Building Scalable Software
Microservices: Building Scalable Software
Credits
Preface
- What this learning path covers
- What you need for this learning path
- Who this learning path is for
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

1. Module 1
- 1. A Solution Approach
  - Evolution of µServices
  - Monolithic architecture overview
  - Limitation of monolithic architecture versus its solution with µServices
    - One dimension scalability
    - Release rollback in case of failure
    - Problems in adopting new technologies
    - Alignment with Agile practices
    - Ease of development – could be done better
    - Microservices build pipeline
    - Deployment using a container such as Docker
      - Containers
      - Docker
      - Docker's architecture
        - Docker image
      - Docker container
      - Deployment
  - Summary
- 2. Setting Up the Development Environment
  - Spring Boot configuration
    - Spring Boot overview
    - Adding Spring Boot to the rest sample
    - Adding a Jetty-embedded server
  - Sample REST program
    - Writing the REST controller class

@RestController
@RequestMapping
@RequestParam
@PathVariable
Making a sample REST app executable
Setting up the application build
Running the Maven tool
Executing with the Java command
REST API testing using the Postman Chrome extension
Some more positive test scenarios
Negative test scenarios
NetBeans IDE installation and setup
References
Summary
3. Domain-Driven Design
Domain-driven design fundamentals
Building blocks
Ubiquitous language
Multilayered architecture
Presentation layer
Application layer
Domain layer
Infrastructure layer
Artifacts of domain-driven design
Entities
Value objects
Frequently asked questions
Services
Aggregates
Repository
Factory
Modules
Strategic design and principles
Bounded context
Continuous integration
Context map
Shared kernel
Customer-supplier
Conformist
Anticorruption layer
Separate ways
Open host service
Distillation

Sample domain service
Entity implementation
Repository implementation
Service implementation
Summary
4. Implementing a Microservice
OTRS overview
Developing and implementing µServices
Restaurant µService
Controller class
API versioning
Service classes
Repository classes
Entity classes
Booking and user services
Registration and Discovery service (Eureka service)
Execution
Testing
References
Summary
5. Deployment and Testing
An overview of microservice architecture using Netflix OSS
Load balancing
Client-side load balancing
Server-side load balancing
Circuit breaker and monitoring
Using Hystrix's fallback methods
Monitoring
Setting up the Hystrix Dashboard
Setting up Turbine
Microservice deployment using containers
Installation and configuration
Docker Machine with 4 GB
Building Docker images with Maven
Running Docker using Maven
Integration testing with Docker
Pushing the image to a registry
Managing Docker containers
References
Summary
6. Securing Microservices
Enabling Secure Socket Layer
Authentication and authorization

- OAuth 2.0
  - Usage of OAuth
  - OAuth 2.0 specification – concise details
    - OAuth 2.0 roles
      - Resource owner
      - Resource server
      - Client
      - Authorization server
    - OAuth 2.0 client registration
      - Client types
        - Confidential client type
        - Public client type
        - Web application
        - User agent-based application
        - Native application
      - Client identifier
      - Client authentication
    - OAuth 2.0 protocol endpoints
      - Authorization endpoint
      - Token endpoint
      - Redirection endpoint
    - OAuth 2.0 grant types
      - Authorization code grant
        - Authorization code requests and responses
      - Implicit grant
        - Implicit grant requests and responses
      - Resource owner password credentials grant
      - Client credentials grant
  - OAuth implementation using Spring Security
    - Authorization code grant
    - Implicit grant
    - Resource owner password credential grant
    - Client credentials grant
  - References
  - Summary
- 7. Consuming Services Using a Microservice Web App
  - AngularJS framework overview
    - MVC
    - MVVM
    - Modules
    - Providers and services
    - Scopes
    - Controllers

Filters
Directives
UI-Router
Development of OTRS features
Home page/restaurant list page
index.html
app.js
restaurants.js
restaurants.html
Search Restaurants
Restaurant details with reservation option
restaurant.html
Login page
login.html
login.js
Reservation confirmation
Setting up the web app
Summary
8. Best Practices and Common Principles
Overview and mindset
Best practices and principals
Nanoservice (not recommended), size, and monolithic
Continuous integration and deployment
System/end-to-end test automation
Self-monitoring and logging
A separate data store for each microservice
Transaction boundaries
Microservices frameworks and tools
Netflix Open Source Software (OSS)
Build – Nebula
Deployment and delivery – Spinnaker with Aminator
Service registration and discovery – Eureka
Service communication – Ribbon
Circuit breaker – Hystrix
Edge (proxy) server – Zuul
Operational monitoring – Atlas
Reliability monitoring service – Simian Army
AWS resource monitoring – Edda
On-host performance monitoring – Vector
Distributed configuration management – Archaius
Scheduler for Apache Mesos – Fenzo
Cost and cloud utilization – Ice
Other security tools – Scumblr and FIDO

Scumblr
Fully Integrated Defence Operation (FIDO)
References
Summary
9. Troubleshooting Guide
Logging and ELK stack
A brief overview
Elasticsearch
Logstash
Kibana
ELK stack setup
Installing Elasticsearch
Installing Logstash
Installing Kibana
Tips for ELK stack implementation
Use of correlation ID for service calls
Let's see how we can tackle this problem
Dependencies and versions
Cyclic dependencies and their impact
It needs to be analyzed while designing the system
Maintaining different versions
Let's explore more
References
Summary
2. Module 2
1. Demystifying Microservices
The evolution of microservices
Business demand as a catalyst for microservices evolution
Technology as a catalyst for the microservices evolution
Imperative architecture evolution
What are microservices?
Microservices – the honeycomb analogy
Principles of microservices
Single responsibility per service
Microservices are autonomous
Characteristics of microservices
Services are first-class citizens
Characteristics of services in a microservice
Microservices are lightweight
Microservices with polyglot architecture
Automation in a microservices environment
Microservices with a supporting ecosystem
Microservices are distributed and dynamic

- Antifragility, fail fast, and self-healing
- Microservices examples
  - An example of a holiday portal
  - A microservice-based order management system
  - An example of a travel agent portal
- Microservices benefits
  - Supports polyglot architecture
  - Enabling experimentation and innovation
  - Elastically and selectively scalable
  - Allowing substitution
  - Enabling to build organic systems
  - Helping reducing technology debt
  - Allowing the coexistence of different versions
  - Supporting the building of self-organizing systems
  - Supporting event-driven architecture
  - Enabling DevOps
- Relationship with other architecture styles
  - Relations with SOA
    - Service-oriented integration
    - Legacy modernization
    - Service-oriented application
    - Monolithic migration using SOA
  - Relations with Twelve-Factor apps
    - A single code base
    - Bundling dependencies
    - Externalizing configurations
    - Backing services are addressable
    - Isolation between build, release, and run
    - Stateless, shared nothing processes
    - Exposing services through port bindings
    - Concurrency to scale out
    - Disposability with minimal overhead
    - Development and production parity
    - Externalizing logs
    - Package admin processes
- Microservice use cases
  - Microservices early adopters
  - The common theme is monolithic migrations
- Summary

2. Building Microservices with Spring Boot
- Setting up a development environment
- Developing a RESTful service – the legacy approach
- Moving from traditional web applications to microservices

Using Spring Boot to build RESTful microservices
Getting started with Spring Boot
Developing the Spring Boot microservice using the CLI
Developing the Spring Boot Java microservice using STS
Examining the POM file
Examining Application.java
Examining application.properties
Examining ApplicationTests.java
Testing the Spring Boot microservice
Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example
What's next?
The Spring Boot configuration
Understanding the Spring Boot autoconfiguration
Overriding default configuration values
Changing the location of the configuration file
Reading custom properties
Using a .yaml file for configuration
Using multiple configuration profiles
Other options to read properties
Changing the default embedded web server
Implementing Spring Boot security
Securing microservices with basic security
Securing a microservice with OAuth2
Enabling cross-origin access for microservices
Implementing Spring Boot messaging
Developing a comprehensive microservice example
Spring Boot actuators
Monitoring using JConsole
Monitoring using SSH
Configuring application information
Adding a custom health module
Building custom metrics
Documenting microservices
Summary
3. Applying Microservices Concepts
Patterns and common design decisions
Establishing appropriate microservice boundaries
Autonomous functions
Size of a deployable unit
Most appropriate function or subdomain
Polyglot architecture
Selective scaling
Small, agile teams

        Single responsibility
        Replicability or changeability
        Coupling and cohesion
        Think microservice as a product
    Designing communication styles
        Synchronous style communication
        Asynchronous style communication
        How to decide which style to choose?
    Orchestration of microservices
    How many endpoints in a microservice?
    One microservice per VM or multiple?
    Rules engine – shared or embedded?
    Role of BPM and workflows
    Can microservices share data stores?
    Setting up transaction boundaries
        Altering use cases to simplify transactional requirements
        Distributed transaction scenarios
    Service endpoint design consideration
        Contract design
        Protocol selection
            Message-oriented services
            HTTP and REST endpoints
            Optimized communication protocols
            API documentations
    Handling shared libraries
    User interfaces in microservices
    Use of API gateways in microservices
    Use of ESB and iPaaS with microservices
    Service versioning considerations
    Design for cross origin
    Handling shared reference data
    Microservices and bulk operations
Microservices challenges
    Data islands
    Logging and monitoring
    Dependency management
    Organization culture
    Governance challenges
    Operation overheads
    Testing microservices
    Infrastructure provisioning
The microservices capability model
    Core capabilities

Infrastructure capabilities
Supporting capabilities
Process and governance capabilities
Summary
4. Microservices Evolution – A Case Study
Reviewing the microservices capability model
Understanding the PSS application
Business process view
Functional view
Architectural view
Design view
Implementation view
Deployment view
Death of the monolith
Pain points
Stop gap fix
Retrospection
Shared data
Single database
Native queries
Stored procedures
Domain boundaries
Microservices to the rescue
The business case
Plan the evolution
Evolutionary approach
Identification of microservices boundaries
Analyze dependencies
Events as opposed to query
Events as opposed to synchronous updates
Challenge requirements
Challenge service boundaries
Final dependency graph
Prioritizing microservices for migration
Data synchronization during migration
Managing reference data
User interfaces and web applications
Session handling and security
Test strategy
Building ecosystem capabilities
Migrate modules only if required
Target architecture
Internal layering of microservices

Orchestrating microservices
Integration with other systems
Managing shared libraries
Handling exceptions
Target implementation view
Implementation projects
Running and testing the project
Summary
5. Scaling Microservices with Spring Cloud
Reviewing microservices capabilities
Reviewing BrownField's PSS implementation
What is Spring Cloud?
Spring Cloud releases
Components of Spring Cloud
Spring Cloud and Netflix OSS
Setting up the environment for BrownField PSS
Spring Cloud Config
What's next?
Setting up the Config server
Understanding the Config server URL
Accessing the Config Server from clients
Handling configuration changes
Spring Cloud Bus for propagating configuration changes
Setting up high availability for the Config server
Monitoring the Config server health
Config server for configuration files
Completing changes to use the Config server
Feign as a declarative REST client
Ribbon for load balancing
Eureka for registration and discovery
Understanding dynamic service registration and discovery
Understanding Eureka
Setting up the Eureka server
High availability for Eureka
Zuul proxy as the API gateway
Setting up Zuul
High availability of Zuul
High availability of Zuul when the client is also a Eureka client
High availability when the client is not a Eureka client
Completing Zuul for all other services
Streams for reactive microservices
Summarizing the BrownField PSS architecture
Summary

6. Autoscaling Microservices
Reviewing the microservice capability model
Scaling microservices with Spring Cloud
Understanding the concept of autoscaling
The benefits of autoscaling
Different autoscaling models
Autoscaling an application
Autoscaling the infrastructure
Autoscaling in the cloud
Autoscaling approaches
Scaling with resource constraints
Scaling during specific time periods
Scaling based on the message queue length
Scaling based on business parameters
Predictive autoscaling
Autoscaling BrownField PSS microservices
The capabilities required for an autoscaling system
Implementing a custom life cycle manager using Spring Boot
Understanding the deployment topology
Understanding the execution flow
A walkthrough of the life cycle manager code
Running the life cycle manager
Summary
7. Logging and Monitoring Microservices
Reviewing the microservice capability model
Understanding log management challenges
A centralized logging solution
The selection of logging solutions
Cloud services
Off-the-shelf solutions
Best-of-breed integration
Log shippers
Log stream processors
Log storage
Dashboards
A custom logging implementation
Distributed tracing with Spring Cloud Sleuth
Monitoring microservices
Monitoring challenges
Monitoring tools
Monitoring microservice dependencies
Spring Cloud Hystrix for fault-tolerant microservices
Aggregating Hystrix streams with Turbine

Data analysis using data lakes
Summary
8. Containerizing Microservices with Docker
Reviewing the microservice capability model
Understanding the gaps in BrownField PSS microservices
What are containers?
The difference between VMs and containers
The benefits of containers
Microservices and containers
Introduction to Docker
The key components of Docker
The Docker daemon
The Docker client
Docker concepts
Docker images
Docker containers
The Docker registry
Dockerfile
Deploying microservices in Docker
Running RabbitMQ on Docker
Using the Docker registry
Setting up the Docker Hub
Publishing microservices to the Docker Hub
Microservices on the cloud
Installing Docker on AWS EC2
Running BrownField services on EC2
Updating the life cycle manager
The future of containerization – unikernels and hardened security
Summary
9. Managing Dockerized Microservices with Mesos and Marathon
Reviewing the microservice capability model
The missing pieces
Why cluster management is important
What does cluster management do?
Relationship with microservices
Relationship with virtualization
Cluster management solutions
Docker Swarm
Kubernetes
Apache Mesos
Nomad
Fleet
Cluster management with Mesos and Marathon

Diving deep into Mesos
The Mesos architecture
Marathon
Implementing Mesos and Marathon for BrownField microservices
Setting up AWS
Installing ZooKeeper, Mesos, and Marathon
Configuring ZooKeeper
Configuring Mesos
Running Mesos, Marathon, and ZooKeeper as services
Running the Mesos slave in the command line
Preparing BrownField PSS services
Deploying BrownField PSS services
Reviewing the deployment
A place for the life cycle manager
Rewriting the life cycle manager with Mesos and Marathon
The technology metamodel
Summary
10. The Microservices Development Life Cycle
Reviewing the microservice capability model
The new mantra of lean IT – DevOps
Reducing wastage
Automating every possible step
Value-driven delivery
Bridging development and operations
Meeting the trio – microservices, DevOps, and cloud
Cloud as the self-service infrastructure for Microservices
DevOps as the practice and process for microservices
Practice points for microservices development
Understanding business motivation and value
Changing the mindset from project to product development
Choosing a development philosophy
Design thinking
The start-up model
The Agile practice
Using the concept of Minimum Viable Product
Overcoming the legacy hotspot
Addressing challenges around databases
Establishing self-organizing teams
Building a self-service cloud
Building a microservices ecosystem
Defining a DevOps-style microservice life cycle process
Value-driven planning
Agile development

Continuous integration
Continuous testing
Continuous release
Continuous monitoring and feedback
Automating the continuous delivery pipeline
Development
Continuous integration
Automated testing
Different candidate tests for automation
Automated sanity tests
Regression testing
Automated functional testing
Automated acceptance testing
Performance testing
Real user flow simulation or journey testing
Automated security testing
Exploratory testing
A/B testing, canary testing, and blue-green deployments
Other nonfunctional tests
Testing in production
Antifragility testing
Target test environments
Continuous deployment
Monitoring and feedback
Automated configuration management
Microservices development governance, reference architectures, and libraries
Summary
3. Module 3
1. Microservices Architecture
Need for microservices
Monolithic software
Microservices in the real world
Microservice-oriented architectures
How is it better?
Shortcomings
Key design principles
Business units, no components
Smart services, dumb communication pipes
Decentralization
Technology alignment
How small is too small?
Key benefits
Resilience

Scalability
Technology heterogeneity
Replaceability
Independence
Why is replaceability important?
Easy to deploy
SOA versus microservices
Why Node.js?
API aggregation
The future of Node.js
Summary
2. Microservices in Node.js – Seneca and PM2 Alternatives
Need for Node.js
Installing Node.js, npm, Seneca, and PM2
Learning npm
Our first program – Hello World
Node.js threading model
Modular organization best practices
Javascript
SOLID design principles
Seneca – a microservices framework
Inversion of control done right
Pattern matching in Seneca
Patrun – a pattern-matching library
Reusing patterns
Writing plugins
Web server integration
Seneca as Express middleware
Data storage
PM2 – a task runner for Node.js
Single-threaded applications and exceptions
Using PM2 – the industry-standard task runner
Summary
3. From the Monolith to Microservices
First, there was the monolith
How to tackle organic growth?
How abstract is too abstract?
Then the microservices appeared
Disadvantages
Splitting the monolith
Problems splitting the monolith – it is all about the data
Organizational alignment
Summary

4. Writing Your First Microservice in Node.js
Micromerce – the big picture
Product Manager – the two-faced core
Fetching products
Fetching by category
Fetching by ID
Adding a product
Removing a product
Editing a product
Wiring everything up
Integrating with Express – how to create a REST API
The e-mailer – a common problem
How to send e-mails
Defining the interface
Setting up Mandrill
Hands on – integrating Mandrill in your microservice
The fallback strategy
The order manager
Defining the microservice – how to gather non-local data
The order manager – the code
Calling remote services
Resilience over perfection
The UI – API aggregation
Need for frontend microservice
The code
Service degradation – when the failure is not a disaster
Circuit breakers
Seneca – a simple puzzle that makes our lives easier
Seneca and promises
Debugging
Summary
5. Security and Traceability
Infrastructure logical security
SSH – encrypting the communications
Application security
Common threats – how to be up to date
Injection
Input validation
Cross-site scripting
Output encoding
Cross-site request forgery
Open redirects
Effective code reviews

- Traceability
  - Logging
  - Tracing requests
  - Auditing
  - HTTP codes
    - 1xx – informational
    - 2xx – success codes
    - 3xx – redirection
    - 4xx – client errors
    - 5xx – server errors
    - Why HTTP codes matter in microservices
- Summary

6. Testing and Documenting Node.js Microservices
- Functional testing
  - The pyramid of automated testing
    - Unit tests
    - Integration tests
    - End-to-end tests
    - How much testing is too much?
  - Testing microservices in Node.js
    - Chai
      - BDD-style interfaces
      - Assertions interface
    - Mocha
    - Sinon.JS – a mocking framework
    - Testing a real microservice
      - TDD – Test-driven development
      - Unit testing
      - End-to-end testing
    - Manual testing – the necessary evil
      - Building a proxy to debug our microservices
      - Postman
- Documenting microservices
  - Documenting APIs with Swagger
  - Generating a project from the Swagger definition
- Summary

7. Monitoring Microservices
- Monitoring services
  - Monitoring using PM2 and Keymetrics
    - Diagnosing problems
    - Monitoring application exceptions
    - Custom metrics
      - Simple metric

Counter
Average calculated values
Simian Army – the active monitoring from Spotify
Throughput and performance degradation
Summary
8. Deploying Microservices
Concepts in software deployment
Continuous integration
Continuous delivery
Deployments with PM2
PM2 – ecosystems
Deploying microservices with PM2
Configuring the server
Docker – a container for software delivery
Setting up the container
Installing Docker
Choosing the image
Running the container
Installing the required software
Saving the changes
Deploying Node.js applications
Automating Docker container creation
Node.js event loop – easy to learn and hard to master
Clustering Node.js applications
Load balancing our application
Health check on NGINX
Passive health check
Active health check
Summary
A. Bibliography
Index

Microservices: Building Scalable Software

Microservices: Building Scalable Software

Easily build and implement scalable microservices from scratch.

A course in three modules



BIRMINGHAM - MUMBAI

Microservices: Building Scalable Software

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: December 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN

www.packtpub.com

Credits

Authors

Sourabh Sharma

Rajesh RV

David Gonzalez

Reviewers

Guido Grazioli

Yogendra Sharma

Kishore Kumar Yekkanti

Content Development Editor

Onkar Wani

Graphics

Abhinash Sahu

Production Coordinator

Shraddha Falebhai

Preface

With the introduction of the cloud, enterprise application development moved from monolithic applications to small, lightweight, and process-driven components called microservices. Microservices architecture is a style of software architecture, which makes application development easier and offers great flexibility to utilize various resources optimally. They are the next big thing in designing scalable, easy-to-maintain applications. In today's world, many enterprises use microservices as the default standard for building large, service-oriented enterprise applications.

Implementing the microservice architecture in Spring Framework, Spring Boot, and Spring Cloud, helps you build modern, Internet-scale Java applications in no time. The Spring framework is a popular programming framework among developer community for many years. Spring Boot removed the need to have a heavyweight application container and provided a means to deploy lightweight, server-less applications, with ease.

This course is a hands-on guide to help you build enterprise-ready implementations of microservices. It explains the domain-driven design and its adoption in microservices. Teaching you how to build smaller, lighter, and faster services that can be implemented easily in a production environment. You will dive deep into Spring Boot, Spring Cloud, Docker, Mesos, and Marathon, to understand how to deploy autonomous services without the need for the application server and to manage resources effectively.

What this learning path covers

Module 1, *Mastering Microservices with Java*, This module teaches you how to build smaller, lighter, and faster services that can be implemented easily in a production environment. Giving you the understanding of the complete life cycle of enterprise app development, from designing and developing to deploying, testing, and security. The Module starts off with making you understand the core concepts and frameworks, you will then focus on the high-level design of large software projects. Gradually moving on to setting up the development environment and configuring it before implementing continuous integration to deploy your microservice architecture. At the end, you will know how to build smaller, lighter, and faster services that can be implemented easily in a production environment.

Module 2, *Spring Microservices*, The goal of this module is to enlighten you with a pragmatic approach and guidelines for implementing responsive microservices at scale. This module will dive deep into Spring Boot, Spring Cloud, Docker, Mesos, and Marathon. You will also understand how Spring Boot is used in deploying autonomous services without the need for a heavyweight application server. You will learn different Spring Cloud capabilities and also realize the use of Docker for containerization and of Mesos and Marathon for computing resource abstraction and cluster-wide control, respectively. In the end, you will have learned how to implement microservice architectures using the Spring framework, Spring Boot, and Spring Cloud.

Module 3, *Developing Microservices with Node.js*, This module is a hands-on guide to start writing microservices using Node.js and the most modern frameworks, especially Seneca and PM2. You will learn how to design, build, test, and deploy microservices using the best practices. Also, how to make the right level of compromise in order to avoid over-designing and get the business requirements aligned with the technical solutions..

What you need for this learning path

Module 1:

For this module, you can use any operating system (Linux, Windows, or Mac) with a minimum of 2 GB RAM. You will also require NetBeans with Java, Maven, Spring Boot, Spring Cloud, Eureka Server, Docker, and CI/CD app. For Docker containers, you may need a separate VM or a cloud host with preferably 16 GB or more RAM.

Module 2:

Chapter 2, Building Microservices with Spring Boot, introduces Spring Boot, which

requires the following software components to test the code:

- JDK 1.8
- Spring Tool Suite 3.7.2 (STS)
- Maven 3.3.1
- Spring Framework 4.2.6.RELEASE
- Spring Boot 1.3.5.RELEASE
- spring-boot-cli-1.3.5.RELEASE-bin.zip
- RabbitMQ 3.5.6
- FakeSMTP

Chapter 5, Scaling Microservices with Spring Cloud, you will learn about the Spring Cloud project. This requires the following software components in addition to the previously mentioned ones:

- Spring Cloud Brixton.RELEASE

Chapter 7, Logging and Monitoring Microservices, we will take a look at how

centralized logging can be implemented for microservices. This requires the

following software stack:

- Elasticsearch 1.5.2
- kibana-4.0.2-darwin-x64
- Logstash 2.1.2

Chapter 8, Containerizing Microservices with Docker, we will demonstrate how we

can use Docker for microservices deployments. This requires the following software

components:

- Docker version 1.10.1

- Docker Hub

Chapter 9, Managing Dockerized Microservices with Mesos and Marathon, uses Mesos and Marathon to deploy dockerized microservices into an autoscalable cloud. The following software components are required for this purpose:

- Mesos version 0.27.1
- Docker version 1.6.2
- Marathon version 0.15.3

Module 3:

In order to follow the module, you will need to install Node.js, PM2 (it is a package that is installed through npm), and MongoDB. We will also need an editor. It is recommended to use Atom, but any general purpose editor should be enough.

Who this learning path is for

This course is intended for Java and Spring developers, DevOps engineers, and system administrators who are familiar with microservice architecture and have a good understanding of the core elements and microservice applications but now want to delve into effectively implementing microservices at the enterprise level.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at http://www.packtpub.com. If you purchased this course elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at https://github.com/PacktPublishing/Microservices-Building-Scalable-Software. We also have other code bundles from our rich catalog of books, videos, and courses available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part 1. Module 1

Mastering Microservices with Java

Master the art of implementing microservices in your production environment with ease

Chapter 1. A Solution Approach

As a prerequisite, I believe you have a basic understanding of microservices and software architecture. If not, I would recommend you Google them and find one of the many resources that explains and describes them in detail. It will help you to understand the concepts and book thoroughly.

After reading this book, you could implement microservices for on premise or cloud production deployment and learn the complete life cycle from design, development, testing, and deployment with continuous integration and deployment. This book is specifically written for practical use and to ignite your mind as a solution architect. Your learning will help you to develop and ship products for any type on premise, including SaaS, PaaS, and so on. We'll primarily use the Java and Java-based framework tools such as Spring Boot and Jetty, and we will use Docker as container.

Note

From this point onwards, µServices will be used for microservices except in quotes.

In this chapter, you will learn the eternal existence of µServices, and how it has evolved. It highlights the large problems that premise and cloud-based products face and how µServices deals with it. It also explains the common problems encountered during the development of SaaS, enterprise, or large applications and their solutions.

In this chapter, we will learn the following topics:

- µServices and a brief background
- Monolithic architecture
- Limitation of monolithic architecture
- The benefits and flexibility microservices offers
- µServices deployment on containers such as Docker

Evolution of µServices

Martin Fowler explains:

> *"The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on "µServices" as the most appropriate name."*

Let's get some background on the way it has evolved over the years. Enterprise architecture evolved more from historic mainframe computing, through client-server architecture (2-tier to n-tier) to ***service-oriented architecture (SOA)****.*

The transformation from SOA to µServices is not a standard defined by any industry organization, but a practical approach practiced by many organizations. SOA eventually evolved to become µServices.

Adrian Cockcroft, former Netflix Architect, describes it as:

> *"Fine grain SOA. So microservice is SOA with emphasis on small ephemeral components."*

Similarly, the following quote from Mike Gancarz (a member that designed the X windows system), which defines one of the paramount percepts of UNIX philosophy, suits the µService paradigm as well:

> *"Small is beautiful."*

µServices shares many common characteristics with SOA, such as focus on services and how one service decouples from another. SOA evolved around monolithic application integration by exposing API that was mostly **Simple Object Access Protocol** (**SOAP**) based. Therefore, middleware such as **Enterprise Service Bus** (**ESB**) is very important for SOA. µServices is less complex, and even though it may use the message bus it is only used for message transport and it does not contain any logic.

Tony Pujals defined µServices beautifully:

> *"In my mental model, I think of self-contained (as in containers) lightweight processes communicating over HTTP, created and deployed with relatively small effort and ceremony, providing narrowly-focused APIs to their consumers."*

Monolithic architecture overview

µServices is not something new, it has been around for many years. Its recent rise is owing to its popularity and visibility. Before µServices became popular, there was primarily monolithic architecture that was being used for developing on premise and cloud applications.

Monolithic architecture allows the development of different components such as presentation, application logic, business logic, and **data access objects** (**DAO**), and then you either bundle them together in **enterprise archive** (**EAR**)/**web archive** (**WAR**), or store them in a single directory hierarchy (for example, Rails, NodeJS, and so on).

Many famous applications such as Netflix have been developed using µServices architecture. Moreover, eBay, Amazon, and Groupon have evolved from monolithic architecture to a µServices architecture.

Now, that you have had an insight into the background and history of µServices, let's discuss the limitations of a traditional approach, namely monolithic app development, and compare how µServices would address them.

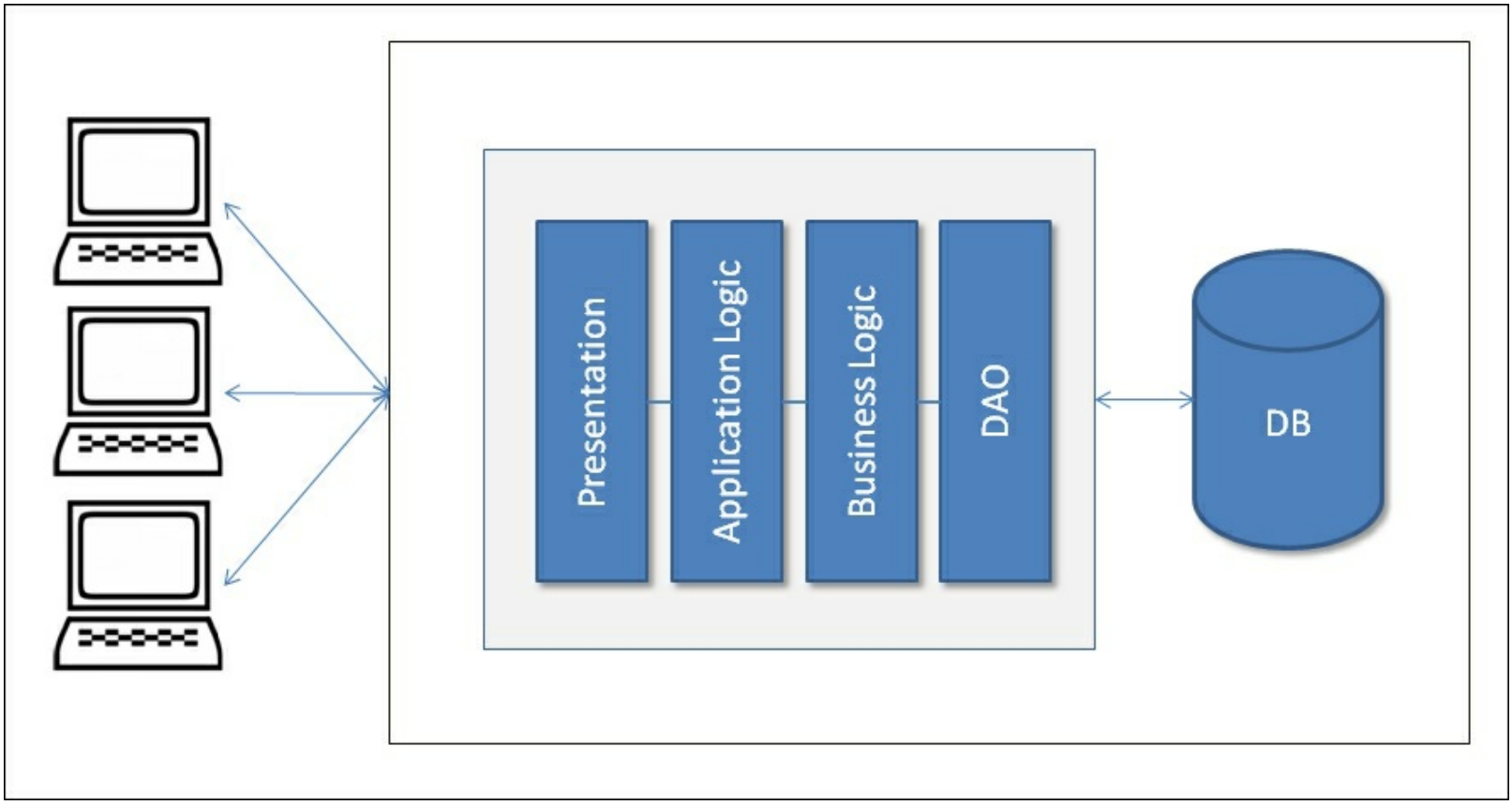
Limitation of monolithic architecture versus its solution with µServices

As we know, change is eternal. Humans always look for better solutions. This is how µServices became what it is today and it may evolve further in the future. Today, organizations are using agile methodologies to develop applications; it is a fast paced development environment and is also on a much larger scale after the invention of cloud and distributed technologies. Many argue that monolithic architecture could also serve a similar purpose and be aligned with agile methodologies, but µServices still provides a better solution to many aspects of production-ready applications.

To understand the design differences between monolithic and µServices, let's take an example of a restaurant table-booking application. This app may have many services such as customers, bookings, analytics and so on, as well as regular components such as presentation and database.

We'll explore three different designs here – traditional monolithic design, monolithic design with services and µServices design.

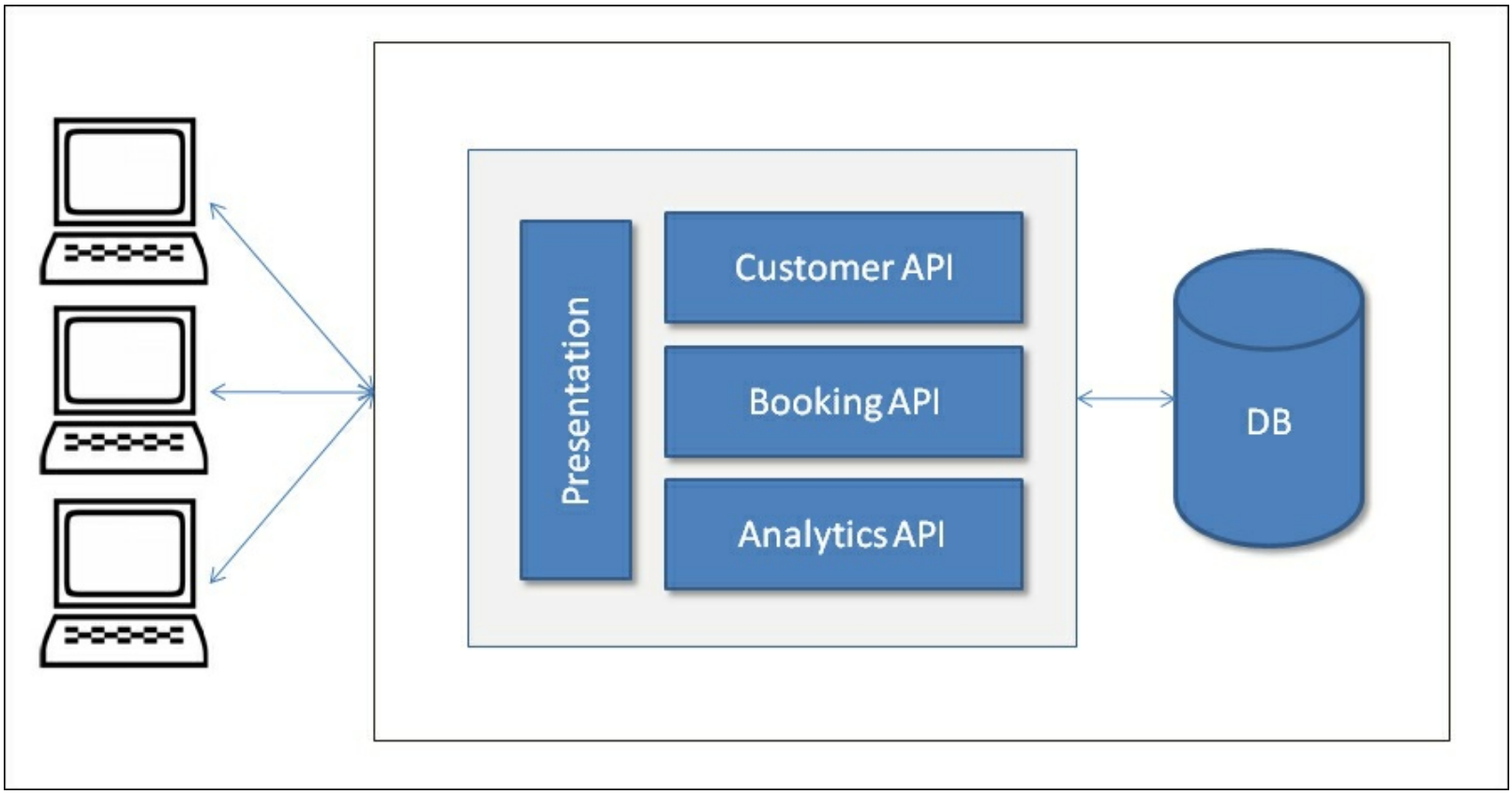
The following diagram explains the traditional monolithic application design. This design was widely used before SOA became popular:



Traditional monolithic design

In traditional monolithic design, everything is bundled in the same archive such as presentation code, application logic and business logic code, and DAO and related code that interacts with the database files or another source.

After SOA, applications started being developed based on services, where each component provides the services to other components or external entities. The following diagram depicts the monolithic application with different services; here services are being used with a presentation component. All services, the presentation component, or any other components are bundled together:

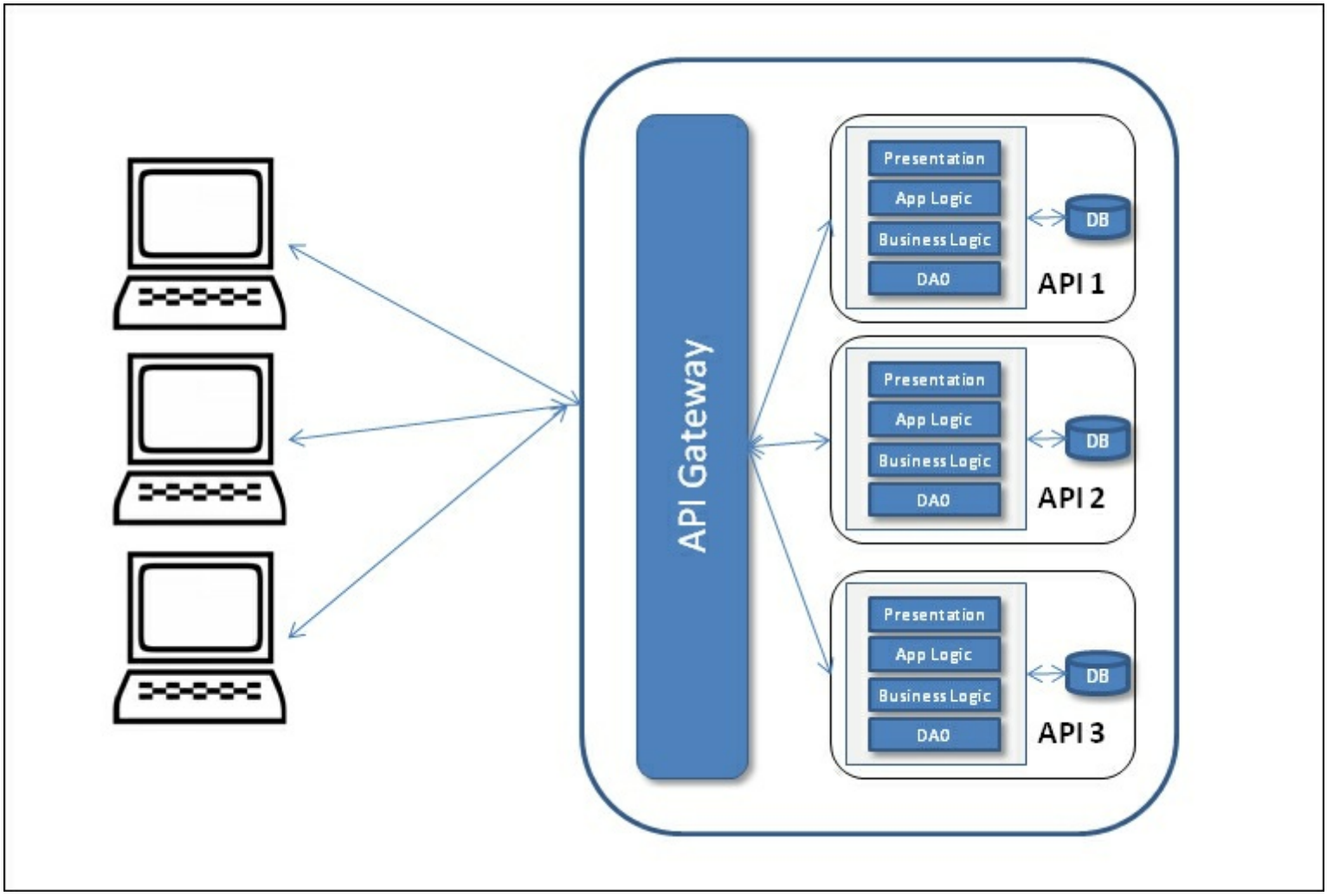


Monolithic design with services

The following third design depicts the µServices. Here, each component represents autonomy. Each component could be developed, built, tested, and deployed independently. Here, even the application UI component could also be a client and consume the µServices. For the purpose of our example, the layer designed is used within µService.

The API gateway provides the interface where different clients can access the individual services and solve the following problems:

- What to do when you want to send different responses to different clients for the same service. For example, a booking service could send different responses to a mobile client (minimal information) and a desktop client (detailed information) providing different details and something different again to a third-party client.
- A response may require fetching information from two or more services:



Microservices design

After observing all the sample design diagrams, which are very high-level designs, you might find out that in monolithic design, the components are bundled together and tightly coupled.

All the services are part of the same bundle. Similarly, in the second design figure, you can see a variant of the first figure where all services could have their own layers and form different APIs, but, as shown in the figure, these are also all bundled together.

Conversely, in µServices, design components are not bundled together and have loose coupling. Each service has its own layers and DB and is bundled in a separate archive. All these deployed services provide their specific API such as Customers, Bookings, or Customer. These APIs are ready to consume. Even the UI is also deployed separately and designed using µService. For this reason, it provides various advantages over its monolithic counterpart. I would still remind you that there are some exceptional cases where monolithic app development is highly successful, like Etsy, and peer-to-peer e-commerce web applications.

One dimension scalability

Monolithic applications, which are large when scaled, scale everything as all the components are bundled together. For example, in the case of a restaurant table reservation application, even if you would like to scale the table-booking service, it would scale the whole application; it cannot scale the table-booking service separately. It does not utilize the resource optimally.

In addition, this scaling is one-dimensional. Running more copies of the application provides scale with increasing transaction volume. An operation team could adjust the number of application copies that were using a load-balancer based on the load in a server farm or a cloud. Each of these copies would access the same data source, therefore increasing the memory consumption, and the resulting I/O operations make caching less effective.

µServices gives the flexibility to scale only those services where scale is required and it allows optimal utilization of the resources. As we mentioned previously, when it is needed, you can scale just the table-booking service without affecting any of the other components. It also allows two-dimensional scaling; here we can not only increase the transaction volume but also the data volume using caching (Platform scale).

A development team can then focus on the delivery and shipping of new features, instead of worrying about the scaling issues (Product scale).

µServices could help you scale platform, people, and product dimensions as we have seen previously. People scaling here refers to an increase or decrease in team size depending on µServices' specific development and focus needs.

µService development using RESTful web service development makes it scalable in the sense that the server-end of REST is stateless; this means that there is not much communication between servers, which makes it horizontally scalable.

Release rollback in case of failure

Since, monolithic applications are either bundled in the same archive or contained in a single directory, they prevent the deployment of code modularity. For example, many of you may have experienced the pain of delaying rolling out the whole release due to the failure of one feature.

To resolve these situations, µServices gives us flexibility to rollback only those features that have failed. It's a very flexible and productive approach. For example, let's assume you are the member of an online shopping portal development team and want to develop an app based on µServices. You can divide your app based on different domains such as products, payments, cart and so on, and package all these components as separate packages. Once you have deployed all these packages separately, these would act as single components that can be developed, tested and deployed independently, and called µService.

Now, let's see how that helps you. Let's say that after a production release launching new features, enhancements, and bug fixes, you find flaws in the payment service that need an immediate fix. Since the architecture you have used is based on µServices, you can rollback the payment service instead of rolling back the whole release, if your application architecture allows, or apply the fixes to the µServics payment service without affecting the other services. This not only allows you to handle failure properly, but also helps to deliver the features/fixes swiftly to customer.

Problems in adopting new technologies

Monolithic applications are mostly developed and enhanced based on the technologies primarily used during the initial development of a project or a product. It makes it very difficult to introduce new technology at a later stage of the development or once the product is in a mature state (for example, after a few years). In addition, different modules in the same project that depend on different versions of the same library make this more challenging.

Technology is improving year on year. For example, your system might be designed in Java and then, a few years later, you want to develop a new service in Ruby on rails or NodeJS because of a business need or to utilize the advantages of new technologies. It would be very difficult to utilize the new technology in an existing monolithic application.

It is not just about code-level integration but also about testing and deployment. It is possible to adopt a new technology by re-writing the entire application, but it is time consuming and a risky thing to do.

On the other hand, because of its component-based development and design, µServices gives us the flexibility to use any technology, new or old, for its development. It does not restrict you to using specific technologies, it gives a new paradigm to your development and engineering activities. You can use Ruby on Rails, NodeJS or any other technology at any time.

So, how is it achieved? Well, it's very simple. µServices-based application code does not bundle into a single archive and is not stored in a single directory. Each µService has its own archive and is deployed separately. A new service could be developed in an isolated environment and could be tested and deployed without any technology issues. As you know, µServices also owns its own separate processes; it serves its purpose without any conflict such as shared resources with tight coupling, and processes remain independent.

Since a µService is by definition a small, self-contained function, it provides a low-risk opportunity to try a new technology. That is definitely not the case where monolithic systems are concerned.

You can also make your Microservice available as open source software so it can be used by others, and if required it may interoperate with a closed source proprietary one, which is not possible with monolithic applications.

Alignment with Agile practices

There is no question that monolithic applications can be developed using agile practices and these are being developed. **Continuous Integration** (**CI**) and **Continuous Deployment** (**CD**) could be used, but, the question is – does it use agile practices effectively? Let's examine the following points:

- For example, when there is a high probability of having stories dependent on each other, and there could be various scenarios, a story could be taken up until the dependent story is not complete
- The build takes more time as the code size increases
- The frequent deployment of a large monolithic application is a difficult task to achieve
- You would have to redeploy the whole application even if you updated a single component
- Redeployment may cause problems to already running components, for example a job scheduler may change whether components impact it or not
- The risk of redeployment may increase if a single changed component does not work properly or if it needs more fixes
- UI developers always need more redeployment, which is quite risky and time-consuming for large monolithic applications

The preceding issues can be tackled very easily by µServices, for example, UI developers may have their own UI component that can be developed, built, tested, and deployed separately. Similarly, other µServices might also be deployable independently and because of their autonomous characteristics, the risk of system failure is reduced. Another advantage for development purposes is that UI developers can make use of the JSON object and mock Ajax calls to develop the UI, which can be taken up in an isolated manner. After development completes, developers can consume the actual APIs and test the functionality. To summarize, you could say that µServices development is swift and it aligns well with the incremental needs of businesses.

Ease of development – could be done better

Generally, large monolithic application code is the toughest to understand for developers, and it takes time before a new developer can become productive. Even loading the large monolithic application into IDE is troublesome, and it makes IDE slower and the developer less productive.

A change in a large monolithic application is difficult to implement and takes more time due to a large code base, and there will be a high risk of bugs if impact analysis is not done properly and thoroughly. Therefore, it becomes a prerequisite for developers to do thorough impact analysis before implementing changes.

In monolithic applications, dependencies build up over time as all components are bundled together. Therefore, risk associated with code change rises exponentially as code changes (number of modified lines of code) grows.

When a code base is huge and more than 100 developers are working on it, it becomes very difficult to build products and implement new features because of the previously mentioned reason. You need to make sure that everything is in place, and that everything is coordinated. A well-designed and documented API helps a lot in such cases.

Netflix, the on-demand Internet streaming provider, had problems getting their application developed with around 100 people. Then, they used a cloud and broke up the app into separate pieces. These ended up being microservices. Microservices grew from the desire for speed and agility and to deploy teams independently.

Micro-components are made loosely coupled thanks to their exposed API, which can be continuously integration tested. With µServices' continuous release cycle, changes are small and developers can rapidly exploit them with a regression test, then go over them and fix the eventual defects found, reducing the risk of a deployment. This results in higher velocity with a lower associated risk.

Owing to the separation of functionality and single responsibility principle, µServices makes teams very productive. You can find a number of examples online where large projects have been developed with minimum team sizes such as eight to ten developers.

Developers can have better focus with smaller code and resultant better feature implementation that leads to a higher empathic relationship with the users of the product. This conduces better motivation and clarity in feature implementation. Empathic relationship with the users allows a shorter feedback loop, and better and speedy prioritization of the feature pipeline. Shorter feedback loop makes defects detection also faster.

Each µServices team works independently and new features or ideas can be implemented without being coordinated with larger audiences. The implementation of end-point failures handling is also easily achieved in the µServices design.

Recently, at one of the conferences, a team demonstrated how they had developed a µServices-based transport-tracking application including iOS and Android apps within 10 weeks, which had Uber-type tracking features. A big consulting firm gave a seven months estimation for the same app to his client. It shows how µServices is aligned with agile methodologies and CI/CD.

Microservices build pipeline

Microservices could also be built and tested using the popular CI/CD tools such as Jenkins, TeamCity, and so on. It is very similar to how a build is done in a monolithic application. In microservices, each microservice is treated like a small application.

For example, once you commit the code in the repository (SCM), CI/CD tools triggers the build process:

- Cleaning code
- Code compilation
- Unit test execution
- Building the application archives
- Deployment on various servers such as Dev, QA, and so on
- Functional and integration test execution
- Creating image containers
- Any other steps

Then, release-build triggers that change the SNAPSHOT or RELEASE version in `pom.xml` (in case of Maven) build the artifacts as described in the normal build trigger. Publish the artifacts to the `artifacts` repository. Tag this version in the repository. If you use the container image then build the container image.

Deployment using a container such as Docker

Owing to the design of µServices, you need to have an environment that provides flexibility, agility and smoothness for continuous integration and deployment as well as for shipment. µServices deployments need speed, isolation management and an agile life cycle.

Products and software can also be shipped using the concept of an intermodal-container model. An intermodal-container is a large standardized container, designed for intermodal freight transport. It allows cargo to use different modes of transport – truck, rail, or ship without unloading and reloading. This is an efficient and secure way of storing and transporting stuff. It resolves the problem of shipping, which previously had been a time consuming, labor-intensive process, and repeated handling often broke fragile goods.

Shipping containers encapsulate their content. Similarly, software containers are starting to be used to encapsulate their contents (products, apps, dependencies, and so on).

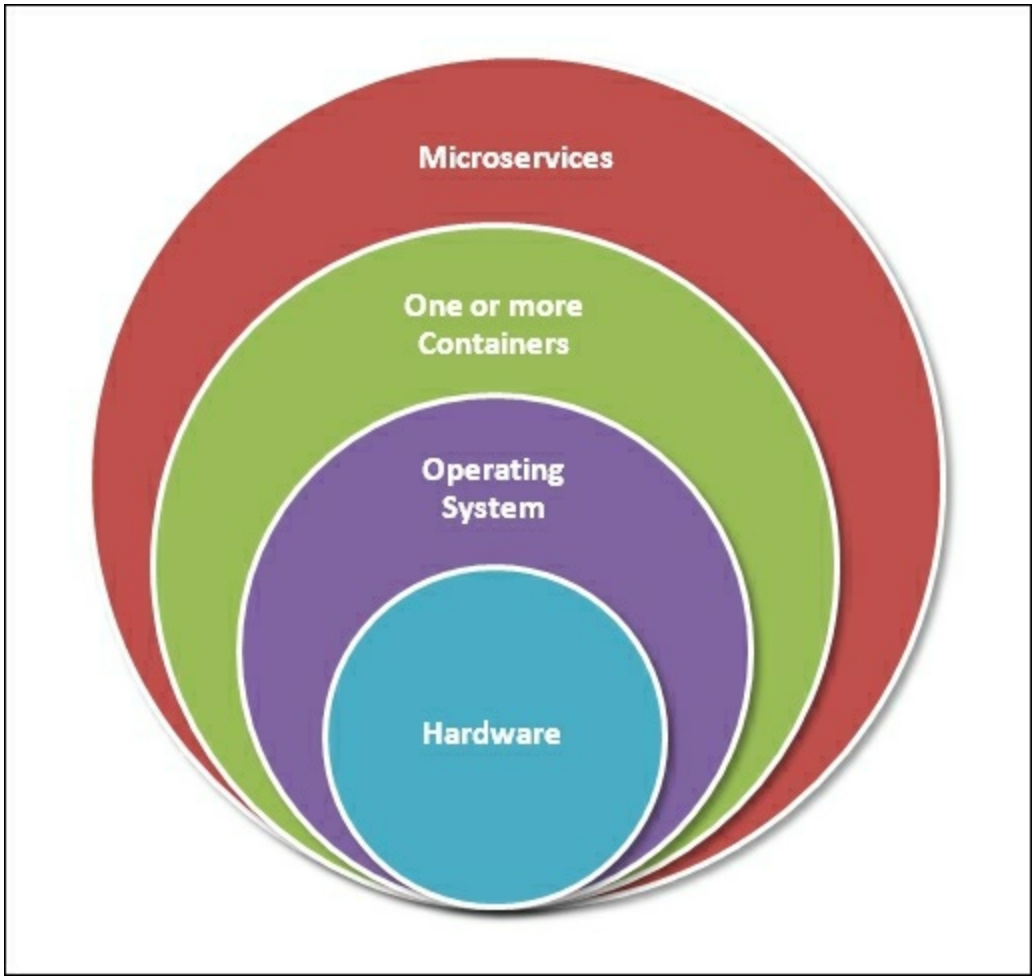
Previously, **virtual machines** (**VMs**) were used to create software images that could be deployed where needed. Later, containers such as Docker became more popular as they were compatible with both traditional virtual stations systems and cloud environments. For example, it is not practical to deploy more than a couple of VMs on a developer's laptop. Building and booting a VM machine is usually I/O intensive and consequently slow.

Containers

A container (for example, Linux containers) provides a lightweight runtime environment consisting of the core features of virtual machines and the isolated services of operating systems. This makes the packaging and execution of µServices easy and smooth.

As the following diagram shows, a container runs as an application (µService) within the operating system. The OS sits on top of the hardware and each OS could have multiple containers, with a container running the application.

A container makes use of an operating system's kernel interfaces such as cnames and namespaces that allow multiple containers to share the same kernel while running in complete isolation to one another. This gives the advantage of not having to complete an OS installation for each usage; the result being that it removes the overhead. It also makes optimal use of the hardware.



Layer diagram for containers

Docker

Container technology is one of the fastest growing technologies today and Docker leads this segment. Docker is an open source project and was launched in 2013. Ten thousand developers tried it after its interactive tutorial launched in August 2013. It was downloaded 2.75 million times by the time of the launch of its 1.0 release in June 2013. Many large companies have signed the partnership agreement with Docker such as Microsoft, Red Hat, HP, OpenStack and service providers such as Amazon web services, IBM, and Google.

As we mentioned earlier, Docker also makes use of the Linux kernel features, such as cgroups and namespaces to ensure resource isolation and packaging of the application with its dependencies. This packaging of dependencies enables an application to run as expected across different Linux operating systems/distributions; supporting a level of portability. Furthermore this portability allows developers to develop an application in any language and then easily deploy it from a laptop to a test or production server.

Note

Docker runs natively on Linux. However, you can also run Docker on Windows and Mac OS using

VirtualBox and boot2docker.

Containers are comprised of just the application and its dependencies including the basic operating system. This makes it lightweight and efficient in terms of resource utilization.. Developers and system administrators get interested in container's portability and efficient resource utilization.

Everything in a Docker container executes natively on the host and uses the host kernel directly. Each container has its own user namespace.

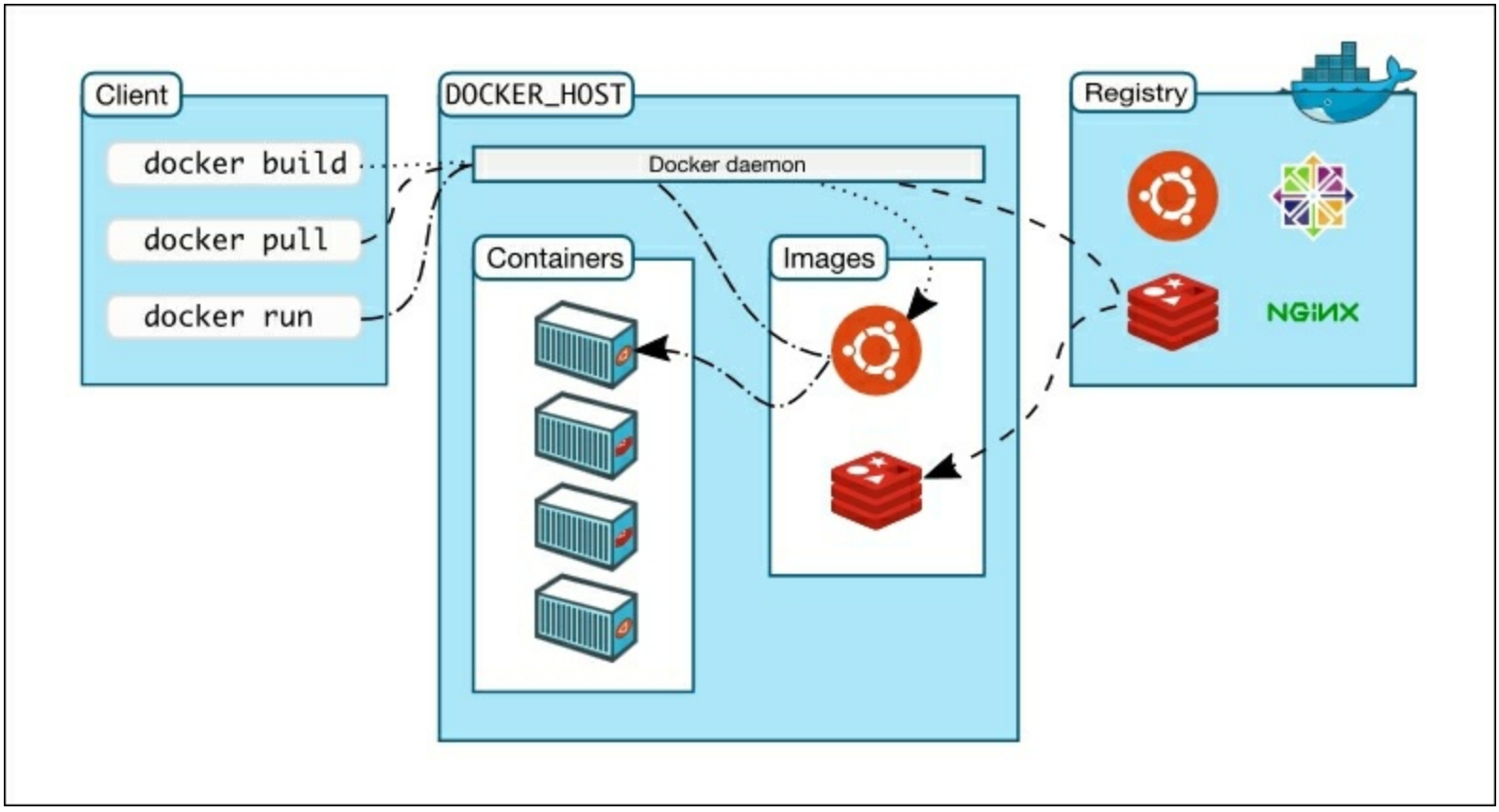
Docker's architecture

As specified on Docker documentation, Docker architecture uses client-server architecture. As shown in the following figure (sourced from Docker's website), the Docker client is primary a user interface that is used by an end user; clients communicate back and forth with a Docker daemon. The Docker daemon does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and the daemon can run on the same system, or different machines. The Docker client and daemon communicate via sockets or through a RESTful API. Docker registries are public or private Docker image repositories from which you upload or download images, for example Docker Hub (hub.docker.com) is a public Docker registry.

The primary components of Docker are a Docker image and a Docker container.

Docker image

A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache web server and your web application installed. Docker images are a build component of Docker. Images are used to create Docker containers. Dockers provides a simple way to build new images or update existing images. You can also use images created by others.



Docker's architecture

Docker container

A Docker container is created from a Docker image. Docker works so that the container can only see its own processes, and have its own filesystem layered onto a host filesystem and a networking stack, which pipes to the host-networking stack. Docker containers can be run, started, stopped, moved or deleted.

Deployment

µServices deployment with Docker deals with three parts:

1. Application packaging, for example, `jar`
2. Building Docker image with `jar` and dependencies using a Docker instruction file, the Dockerfile and command `docker build`. It helps to repeatedly create the image.
3. Docker container execution from this newly built image using command `docker run`.

The preceding information will help you to understand the basics of Docker. You will learn more about Docker and its practical usage in Chapter 5, *Deployment and Testing*. Source and reference: https://docs.docker.com.

Summary

In this chapter, you have learned or rehearsed the high-level design of large software projects from traditional monolithic to μServices applications. You were also introduced to a brief history of μServices, the limitation of monolithic applications, and the benefits and flexibility that microservices offers. I hope this chapter helped you to understand the common problems faced in a production environment by monolithic applications and how microservices can resolve such problem. You were also introduced to lightweight and efficient Docker containers, and saw how containerization is an excellent way to simplify microservices deployment.

In the next chapter, you will get to know about setting up the development environment from IDE, and other development tools, to different libraries We will deal with creating basic projects and setting up Spring Boot configuration to build and develop our first microservice. Here, we will use Java 8 as the language and Spring Boot for our project.

Chapter 2. Setting Up the Development Environment

This chapter focuses on the development environment setup and configurations. If you are familiar with the tools and libraries, you could skip this chapter and continue with Chapter 3, *Domain-Driven Design* where you could explore the domain driven design.

This chapter will cover the following topics:

- Spring Boot configuration
- Sample REST program
- Build setup
- REST API testing using the Postman Chrome extension
- NetBeans – installation and setup

This book will use only the open source tools and frameworks for examples and code. The book will also use Java 8 as its programming language, and the application framework will be based on the Spring framework. This book makes use of Spring Boot to develop microservices.

NetBeans Integrated Development Environment (**IDE**) that provides state of the art support for both Java and JavaScript, is sufficient for our needs. It has evolved a lot over the years and has built-in support for most of the technologies used by this book, such as Maven, Spring Boot and so on. Therefore, I would recommend you to use NetBeans IDE. You are, however free to use any IDE.

We will use Spring Boot to develop the REST services and microservices. Opting for the most popular of Spring frameworks, Spring Boot, or its subset Spring Cloud in this book was a conscious decision. Because of this, we don't need to write applications from scratch and it provides default configuration for most of the stuff for Cloud applications. A Spring Boot overview is provided in Spring Boot's configuration section. If you are new to Spring Boot, this would definitely help you.

We will use Maven as our build tool. As with the IDE, you can use whichever build tool you want, for example Gradle or Ant. We will use the embedded Jetty as our web server but another alternative is to use an embedded Tomcat web server. We will also use the Postman extension of Chrome for testing our REST services.

We will start with Spring Boot Configurations. If you are new to NetBeans or are facing issues in setting up the environment, you can refer to the NetBeans IDE installation section explained in the last section; otherwise you can skip that section altogether.

Spring Boot configuration

Spring Boot is an obvious choice to develop state of the art production-ready applications specific to Spring. Its website also states its real advantages:

> *"Takes an opinionated view of building production-ready Spring applications. Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible."*

Spring Boot overview

Spring Boot is an amazing Spring tool created by Pivotal and released in April 2014 (GA). It was developed based on request of SPR-9888 (https://jira.spring.io/browse/SPR-9888) with the title *Improved support for 'containerless' web application architectures*.

You must be wondering why containerless? Because, today's cloud environment or PaaS provides most of the features offered by container-based web architectures such as reliability, management or scaling. Therefore, Spring Boot focuses on making itself an ultra light container.

Spring Boot is preconfigured to make production-ready web applications very easily. Spring Initializer (http://start.spring.io) is a page where you can select build tools such as Maven or Gradle, project metadata such as group, artifact and dependencies. Once, you feed the required fields you can just click on the **Generate Project** button, which will give you the Spring Boot project that you can use for your production application.

On this page, the default packaging option is `jar`. We'll also use jar packaging for our microservices development. The reason is very simple: it makes microservices development easier. Just think how difficult it would be to manage and create an infrastructure where each microservice runs on its own server instance.

Josh Long shared in his talk in one of the Spring IOs:

> *"It is better to make Jar, not War."*

Later, we will use the Spring Cloud that is a wrapper on top of Spring Boot.

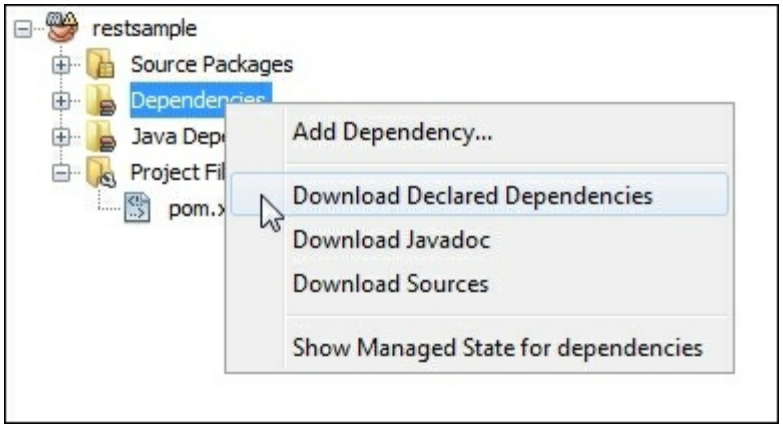
Adding Spring Boot to the rest sample

At the time of writing the book, Spring Boot 1.2.5 release version was available. You can use the latest released version. Spring Boot uses Spring 4 (4.1.7 release).

Open the `pom.xml` (available under **restsample** | **Project Files**) to add Spring Boot to your rest sample project:

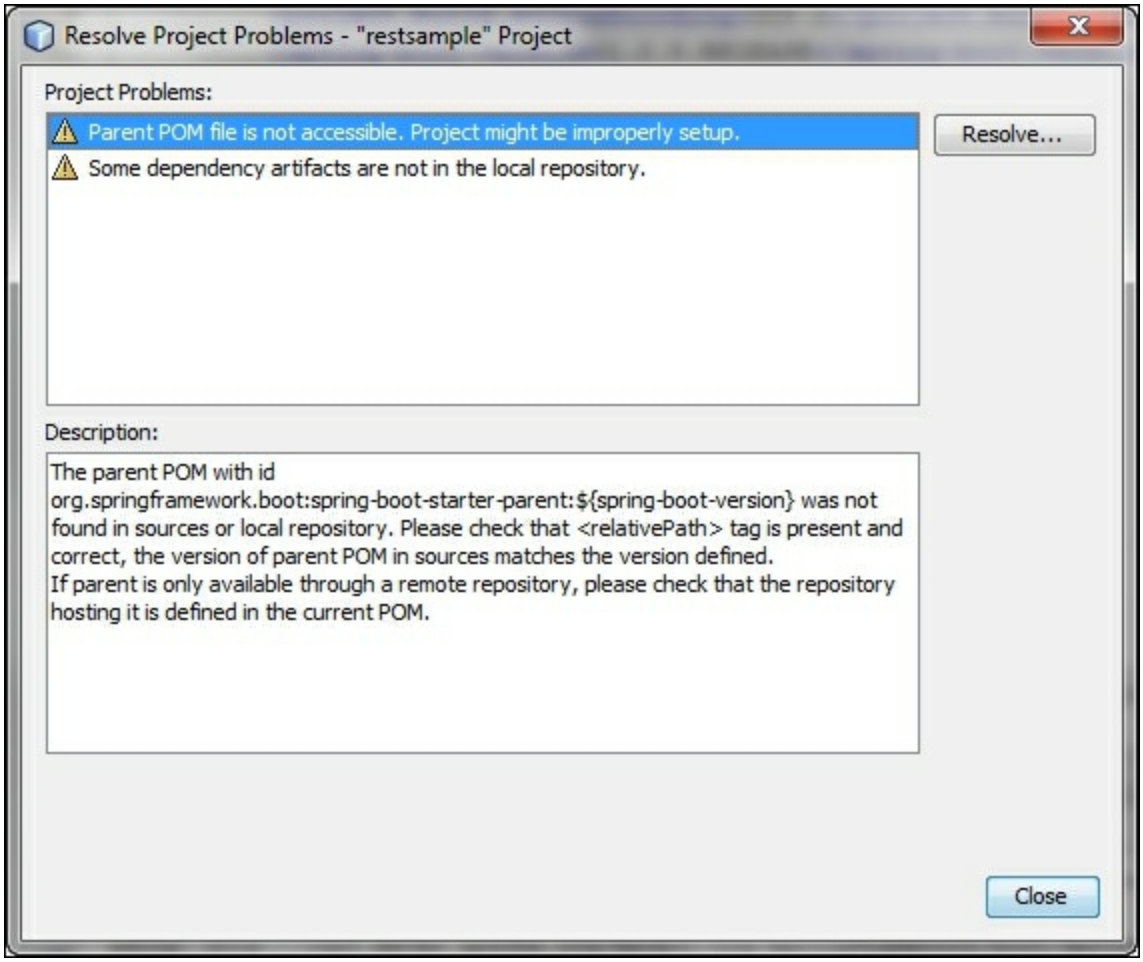
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packtpub.mmj</groupId>
    <artifactId>restsample</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
    </parent>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <spring-boot-version>1.2.5.RELEASE</spring-boot-version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>${spring-boot-version}</version>
        </dependency>
    </dependencies>
</project>
```

If you are adding these dependencies for the first time, you need to download the dependencies by right clicking on the `Dependencies` folder under **restsample** project in the **Projects** pane shown as follows:



Download Maven Dependencies in NetBeans

Similarly, to resolve the project problems, right-click on the NetBeans project **restsample** and opt for the **Resolve Project Problems…**. It will open the dialog shown as follows. Click on the **Resolve...** button to resolve the issues:



Resolve project problems dialog

Note

If you are using Maven behind the proxy, then update the proxies settings in `<NetBeans Installation Directory>\java\maven\conf\settings.xml`. You may need to restart the NetBeans IDE

The preceding steps will download all the required dependencies from a remote Maven repository if the declared dependencies and transitive dependencies are not available in a local Maven repository. If you are downloading the dependencies for the first time, then it may take a bit of time, depending on your Internet speed.

Adding a Jetty-embedded server

Spring Boot by default provides Apache Tomcat as an embedded application container. This book will use the Jetty-embedded application container in the place of Apache Tomcat. Therefore, we need to add a Jetty application container dependency to support the Jetty web server.

Jetty also allows you to read keys or trust stores using classpath that is, you don't need to keep these stores outside the JAR files. If you use Tomcat with SSL, then you will need to access the key store or trust store directly from the filesystem but you can't do that using the classpath. The result is that you can't read a key store or a trust store within a JAR file because Tomcat requires that the key store (and trust store if you're using one) is directly accessible on the filesystem.

This limitation doesn't apply to Jetty, which allows the reading of keys or trust stores within a JAR file:

```
<dependencies>
<dependency>
       <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-web</artifactId>
           <exclusions>
             <exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
               </exclusion>
            </exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
</dependencies>
```

Sample REST program

You will use a simple approach to building a stand-alone application. It packages everything into a single, executable JAR file, driven by a `main()` method. Along the way, you use Spring's support for embedding the Jetty servlet container as the HTTP runtime, instead of deploying it to an external instance. Therefore, we would create the executable JAR in place of the war that needs to be deployed on external web servers.

Now, as you are ready with Spring Boot in NetBeans IDE, you could create your sample web service. You will create a Math API that performs simple calculations and generates the result as JSON.

Let's discuss how we can call and get responses from REST services.

The service will handle GET requests for /calculation/sqrt or /calculation/power and so on. The GET request should return a `200 OK` response with JSON in the body that represents the square root of given number. It should look something like this:

```
{
  "function": "sqrt",
  "input": [
    "144"
  ],
  "output": [
    "12.0"
  ]
}
```

The `input` field is the input parameter for the square root function, and the content is the textual representation of the result.

You could create a resource representation class to model the representation by using **Plain Old Java Object** (**POJO**) with fields, constructors, setters, and getters for the input, output, and function data:

```
package com.packtpub.mmj.restsample.model;

import java.util.List;

public class Calculation {

    String function;
    private List<String> input;
    private List<String> output;

    public Calculation(List<String> input, List<String> output, String function)
{
        this.function = function;
        this.input = input;
        this.output = output;
    }
```

```
    public List<String> getInput() {
        return input;
    }

    public void setInput(List<String> input) {
        this.input = input;
    }

    public List<String> getOutput() {
        return output;
    }

    public void setOutput(List<String> output) {
        this.output = output;
    }

    public String getFunction() {
        return function;
    }

    public void setFunction(String function) {
        this.function = function;
    }

}
```

Writing the REST controller class

Roy Fielding defined and introduced the term **REST**, **Representational State Transfer** in his doctoral dissertation. REST is a style of software architecture for a distributed hypermedia system such as WWW. RESTful refers to those systems that conform to REST architecture properties, principles, and constraints.

Now, you'll create a REST controller to handle the calculation resource. The controller handles the HTTP requests in the Spring RESTful web service implementation.

@RestController

`@RestController` is a class-level annotation used for the `resource` class introduced in Spring 4. It is a combination of `@Controller` and `@ResponseBody`, and because of it, class returns a domain object instead of a view.

In the following code, you can see that the `CalculationController` class handles GET requests for `/calculation` by returning a new instance of the `calculation` class.

We will implement two URLs for a calculation resource – the square root (`Math.sqrt()` function) as `/calculations/sqrt` URL, and power (`Math.pow()` function) as `/calculation/power` URL.

@RequestMapping

`@RequestMapping` annotation is used at class-level to map the `/calculation` URI to `CalculationController` class that is, it ensures that the HTTP request to `/calculation` is mapped to the `CalculationController` class. Based on the path defined using the annotation `@RequestMapping` of the URI (postfix of `/calculation`, for example, `/calculation/sqrt/144`), it would be mapped to respective methods. Here, the request mapping `/calculation/sqrt` is mapped to the `sqrt()` method and `/calculation/power` is mapped to the `pow()` method.

You might have also observed that we have not defined what request method (GET/POST/PUT, and so on) these methods would use. The `@RequestMapping` annotation maps all the HTTP request methods by default. You could use specific methods by using the method property of `RequestMapping`. For example, you could write a `@RequestMethod` annotation in the following way to use the `POST` method:

```
@RequestMapping(value = "/power", method = POST)
```

For passing the parameters along the way, the sample demonstrates both request parameters and path parameters using annotations `@RequestParam` and `@PathVariable` respectively.

@RequestParam

`@RequestParam` is responsible for binding the query parameter to the parameter of the controller's method. For example, the QueryParam base and exponent are bound to parameters `b` and `e` of method

`pow()` of `CalculationController` respectively. Both of the query parameters of the `pow()` method are required since we are not using any default value for them. Default values for query parameters could be set using the `defaultValue` property of `@RequestParam` for example `@RequestParam(value="base", defaultValue="2")`, here, if the user does not pass the query parameter base, then the default value `2` would be used for the base.

If no `defaultValue` is defined, and the user doesn't provide the request parameter, then `RestController` returns the HTTP status code 400 with the message **400 Required String parameter base is not present**. It always uses the reference of the first required parameter if more than one of the request parameters is missing:

```
{
  "timestamp": 1464678493402,
  "status": 400,
  "error": "Bad Request",
  "exception":
"org.springframework.web.bind.MissingServletRequestParameterException",
  "message": "Required String parameter 'base' is not present",
  "path": "/calculation/power/"
}
```

@PathVariable

`@PathVariable` helps you to create the dynamic URIs. `@PathVariable` annotation allows you to map Java parameters to a path parameter. It works with `@RequestMapping` where placeholder is created in URI then the same placeholder name is used either as a `PathVariable` or a method parameter, as you can see in the `CalculationController` class's method `sqrt()`. Here, the value placeholder is created inside the `@RequestMapping` and the same value is assigned to the value of the `@PathVariable`.

Method `sqrt()` takes the parameter in the URI in place of the request parameter. For example, `http://localhost:8080/calculation/sqrt/144`. Here, the `144` value is passed as the path parameter and this URL should return the square root of 144 that is, `12`.

To use the basic check in place, we use the regular expression `"^-?+\\d+\\.?+\\d*$"` to allow only valid numbers in parameters. If non-numeric values are passed, the respective method adds an error message in the output key of the JSON.

Tip

`CalculationController` also uses the regular expression `.+` in the `path` variable (`path` parameter) to allow the decimal point(`.`) in numeric values - `/path/{variable:.+}`. Spring ignores anything after the last dot. Spring default behavior takes it as a file extension.

There are other alternatives such as adding a slash at the end (`/path/{variable}/`) or overriding the method `configurePathMatch()` of `WebMvcConfigurerAdapter` by setting the `useRegisteredSuffixPatternMatch` to `true`, using `PathMatchConfigurer` (available in Spring

4.0.1+).

```
package com.packtpub.mmj.restsample.resources;

package com.packtpub.mmj.restsample.resources;

import com.packtpub.mmj.restsample.model.Calculation;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/calculation")
public class CalculationController {

    private static final String PATTERN = "^-?+\\d+\\.?+\\d*$";

    @RequestMapping("/power")
    public Calculation pow(@RequestParam(value = "base") String b,
@RequestParam(value = "exponent") String e) {
        List<String> input = new ArrayList();
        input.add(b);
        input.add(e);
        List<String> output = new ArrayList();
        String powValue = "";
        if (b != null && e != null && b.matches(PATTERN) && e.matches(PATTERN))
{
            powValue = String.valueOf(Math.pow(Double.valueOf(b),
Double.valueOf(e)));
        } else {
            powValue = "Base or/and Exponent is/are not set to numeric value.";
        }
        output.add(powValue);
        return new Calculation(input, output, "power");
    }

    @RequestMapping(value = "/sqrt/{value:.+}", method = GET)
    public Calculation sqrt(@PathVariable(value = "value") String aValue) {
        List<String> input = new ArrayList();
        input.add(aValue);
        List<String> output = new ArrayList();
        String sqrtValue = "";
        if (aValue != null && aValue.matches(PATTERN)) {
            sqrtValue = String.valueOf(Math.sqrt(Double.valueOf(aValue)));
        } else {
            sqrtValue = "Input value is not set to numeric value.";
        }
        output.add(sqrtValue);
        return new Calculation(input, output, "sqrt");
    }
```

```
}
```

Here, we are exposing only the `power` and `sqrt` functions for the `Calculation` resource using URI `/calculation/power` and `/calculation/sqrt`.

Tip

Here, we are using sqrt and power as a part of the URI, which we have used for demonstration purposes only. Ideally, these should have been passed as value of a request parameter "function"; or something similar based on endpoint design formation.

One interesting thing here is that due to Spring's HTTP message converter support, the `Calculation` object gets converted to JSON automatically. You don't need to do this conversion manually. If Jackson 2 is on the classpath, Spring's `MappingJackson2HttpMessageConverter` converts the `Calculation` object to JSON.

Making a sample REST app executable

Create a class `RestSampleApp` with the annotation `SpringBootApplication`. The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application.

We will annotate the `RestSampleApp` class with the `@SpringBootApplication` that adds all of the following tags implicitly:

- The `@Configuration` annotation tags the class as a source of Bean definitions for the application context.
- The `@EnableAutoConfiguration` annotation indicates that Spring Boot is to start adding beans based on classpath settings, other beans, and various property settings.
- The `@EnableWebMvc` annotation is added if Spring Boot finds the spring-webmvc on the classpath. It treats the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
- The `@ComponentScan` annotation tells Spring to look for other components, configurations, and services in the given package:

```
package com.packtpub.mmj.restsample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestSampleApp {

    public static void main(String[] args) {
        SpringApplication.run(RestSampleApp.class, args);
    }
}
```

This web application is 100 percent pure Java and you didn't have to deal with configuring any plumbing or infrastructure using XML; instead it uses the Java annotation, that is made even simpler by Spring Boot. Therefore, there wasn't a single line of XML except `pom.xml` for Maven. There wasn't even a `web.xml` file.

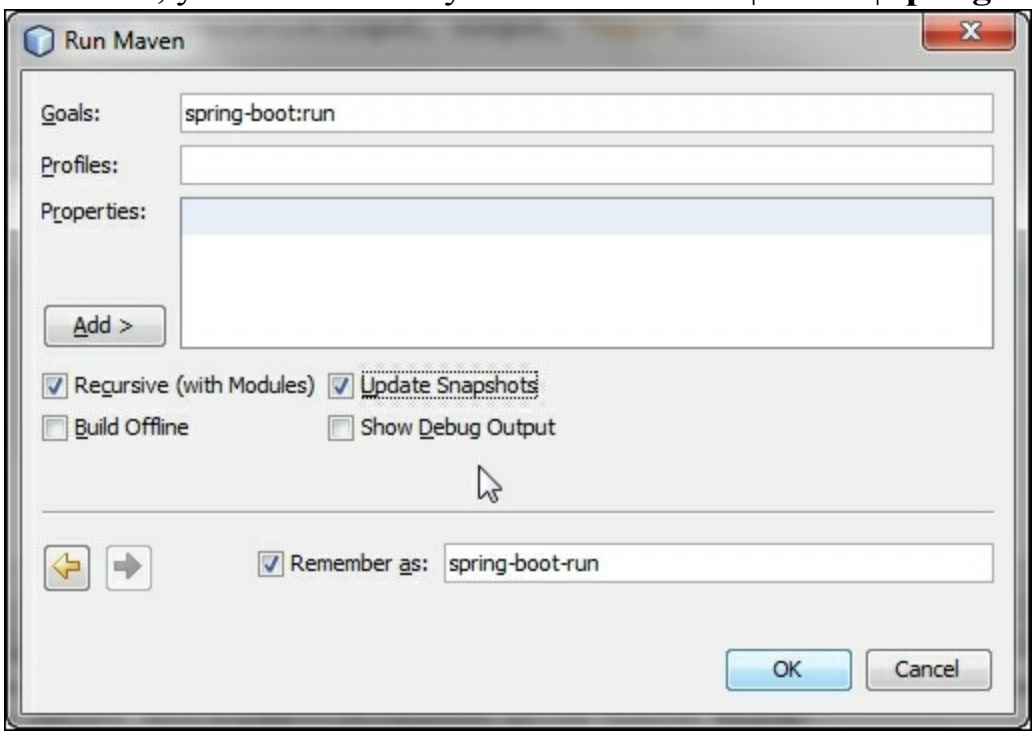
Setting up the application build

Whatever `pom.xml` we have used until now is enough to execute our sample REST service. This service would package the code into a JAR. To make this JAR executable we need to opt for the following options:

Running the Maven tool

Here, we use the Maven tool to execute the generated JAR, steps for the same are as follows:

1. Right-click on the `pom.xml`.
2. Select **run-maven** | **Goals…** from the pop-up menu. It will open the dialog. Type `spring-boot:run` in the **Goals** field. We have used the released version of Spring Boot in the code. However, if you are using the snapshot release, you can check the **Update Snapshots** checkbox. To use it in the future, type `spring-boot-run` in the **Remember as** field.
3. Next time, you could directly click **run-maven** | **Goals** | **spring-boot-run** to execute the project:



Run Maven dialog

4. Click on **OK** to execute the project.

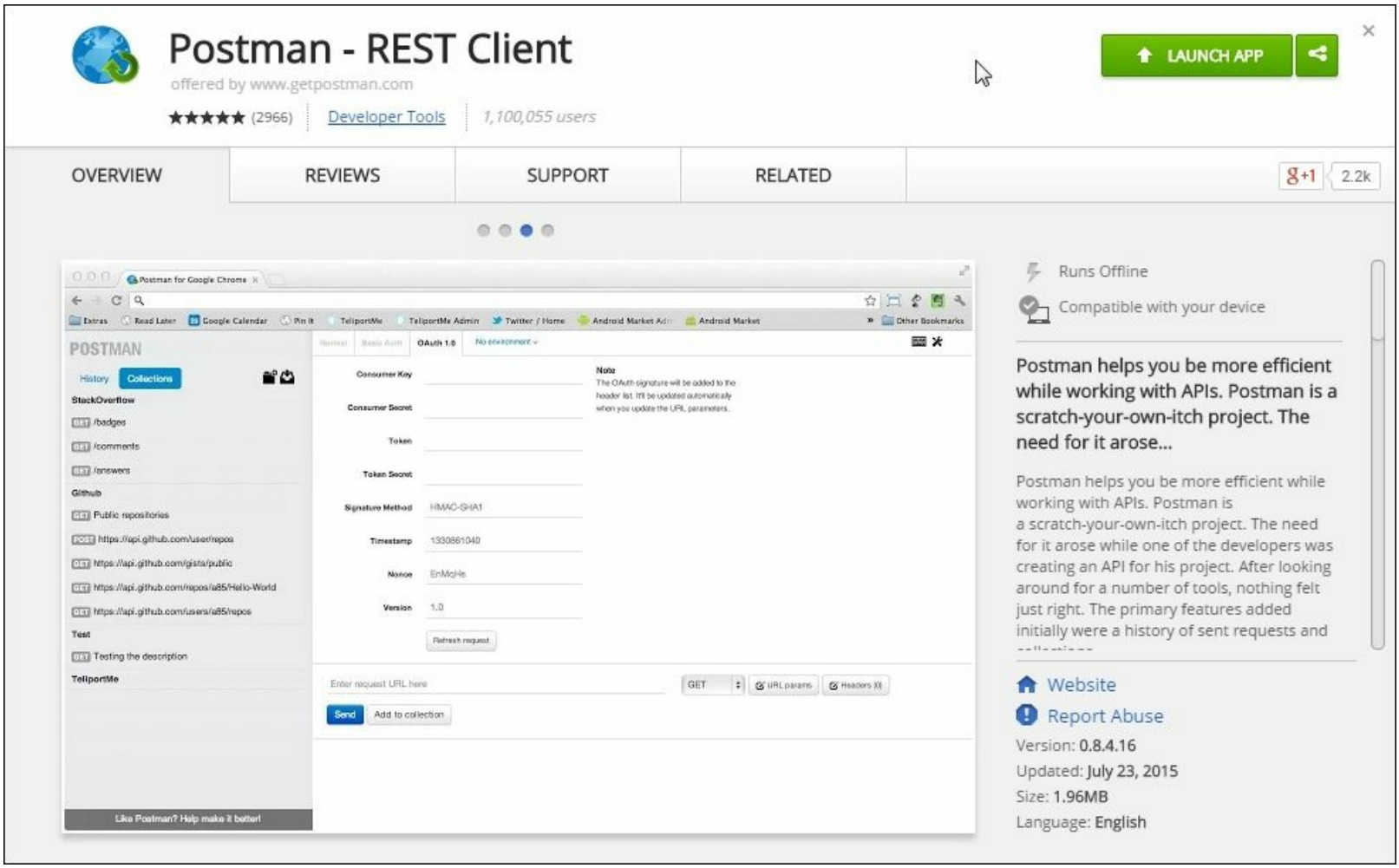
Executing with the Java command

To build the JAR, perform the `mvn clean package` Maven goal. It creates the JAR file in a `target` directory, then, the JAR can be executed using the command:

```
java -jar target/restsample-1.0-SNAPSHOT.jar
```

REST API testing using the Postman Chrome extension

This book uses the Postman – REST Client extension for Chrome to test our REST service. We use the 0.8.4.16 version that can be downloaded using https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm. This extension is no longer searchable but you can add it to your Chrome using the given link. You can also use the Postman Chrome app or any other REST Client to test your sample REST application:



Postman – Rest Client Chrome extension

Let's test our first REST resource once you have the Postman – REST Client installed. We start the Postman – REST Client from either the start menu or from a shortcut.

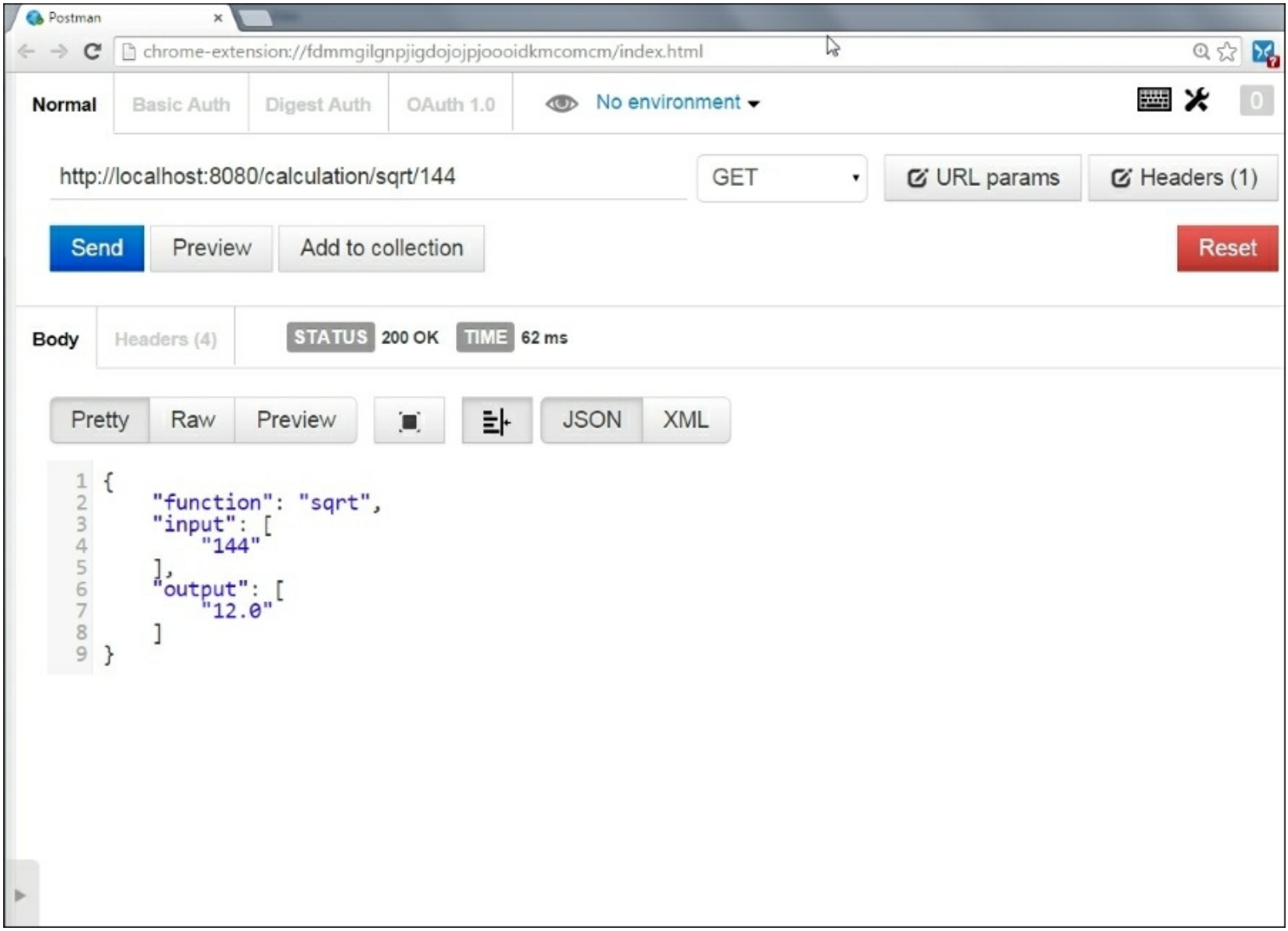
Tip

By default the embedded web server starts on port 8080. Therefore, we need to use the `http://localhost:8080/<resource>` URL for accessing the sample REST application. Example:

`http://localhost:8080/calculation/sqrt/144`.

Once it's started, you can type the Calculation REST URL for `sqrt` and value `144` as the path parameter. You could see it in the following image. This URL is entered in the URL (Enter request URL here) input field of the Postman extension. By default, the request method is GET. We use the default value for the request method, as we have also written our RESTful service to serve the request GET method.

Once you are ready with your input data as mentioned above, you can submit the request by clicking the **Send** button. You can see in the following image that the response code `200` is returned by your sample rest service. You can find the **Status** label in the following image to see the **200 OK** code. A successful request also returns the JSON data of the Calculation Resource, shown in the `Pretty` tab in the screenshot. The returned JSON shows the `sqrt`, value of the function key. It also displays `144` and `12.0` as the input and output lists respectively:



Calculation (sqrt) resource test with Postman

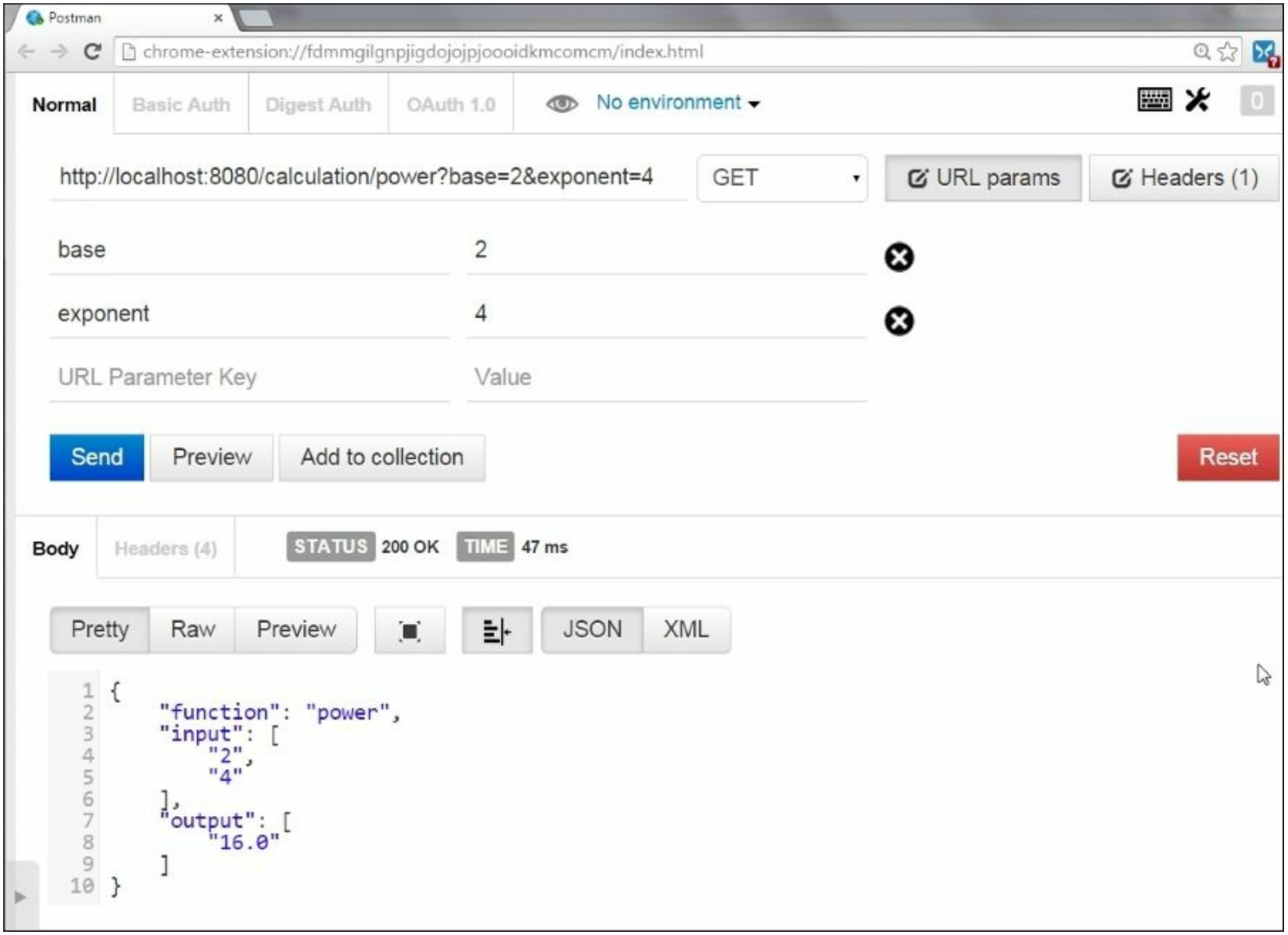
Similarly, we also test our sample REST service for the calculating `power` function. We input the following data in the Postman extension:

- **URL**: `http://localhost:8080/calculation/power?base=2&exponent=4`
- **Request method**: `GET`

Here, we are passing the request parameters base and exponent with values of `2` and `4` respectively. It returns the following JSON with a response status of **200** as shown in the following screenshot.

Returned JSON:

```
{
    "function": "power",
    "input": [
        "2",
        "4"
    ],
    "output": [
        "16.0"
    ]
}
```



Calculation (power) resource test with Postman

Some more positive test scenarios

In the following table, all the URLs start with `http://localhost:8080`:

URL	Output JSON

`/calculation/sqrt/12344.234`	{
	"function": "sqrt",
	"input": [
	"12344.234"
	],
	"output": [
	"111.1046083652699"
	]
	}

`/calculation/sqrt/-9344.34`
 	
`Math.sqrt` function's special scenario:
• If the argument is `NaN` or less than zero, then the result is `NaN`
	{
	"function": "sqrt",
	"input": [
	"-9344.34"
	],
	"output": [
	"NaN"
	]
	}

`/calculation/power?base=2.09&exponent=4.5`	{
	"function": "power",
	"input": [
	"2.09",
	"4.5"
	],
	"output": [
	"27.58406626826615"
	]
	}

`/calculation/power?base=-92.9&exponent=-4`	{
	"function": "power",
	"input": [
	"-92.9",
	"-4"
	],
	"output": [
	"1.3425706351762353E-8"
	]
	}

Negative test scenarios

Similarly, you could also perform some negative scenarios as shown in the following table. In this table, all the URLs start with `http://localhost:8080`:

URL	Output JSON

/calculation/power?base=2a&exponent=4	{
	"function": "power",
	"input": [
	"2a",
	"4"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}

/calculation/power?base=2&exponent=4b	{
	"function": "power",
	"input": [
	"2",
	"4b"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}

/calculation/power?base=2.0a&exponent=a4	{
	"function": "power",
	"input": [
	"2.0a",
	"a4"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}

/calculation/sqrt/144a	{
	"function": "sqrt",
	"input": [
	"144a"
	],
	"output": [
	"Input value is not set to numeric value."
	]
	}

/calculation/sqrt/144.33$	{
	"function": "sqrt",
	"input": [
	"144.33$"
	],
	"output": [
	"Input value is not set to numeric value."
	]
	}

NetBeans IDE installation and setup

NetBeans IDE is free and open source, and has a big community of users. You can download the NetBeans IDE from https://netbeans.org/downloads/, its official website.

At the time of writing this book, version 8.0.2 was the latest available version. As shown in the following screenshot, please download all the supported NetBeans bundles, as we'll use Javascript also:

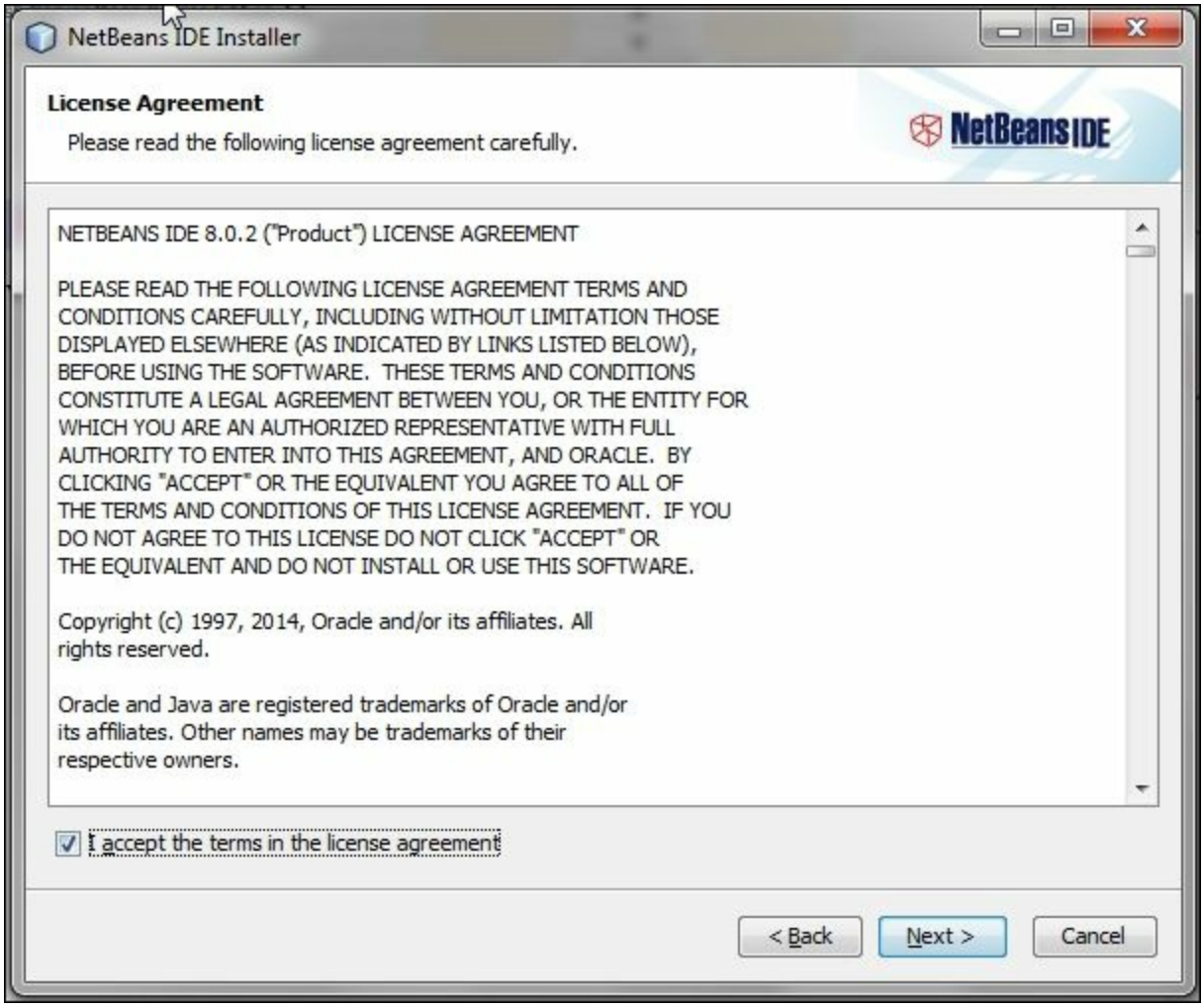


NetBeans bundles

After downloading the the installation, execute the installer file. Accept the license agreement as shown in the following screenshot, and follow the rest of the steps to install the NetBeans IDE. Glassfish Server and Apache Tomcat are optional.

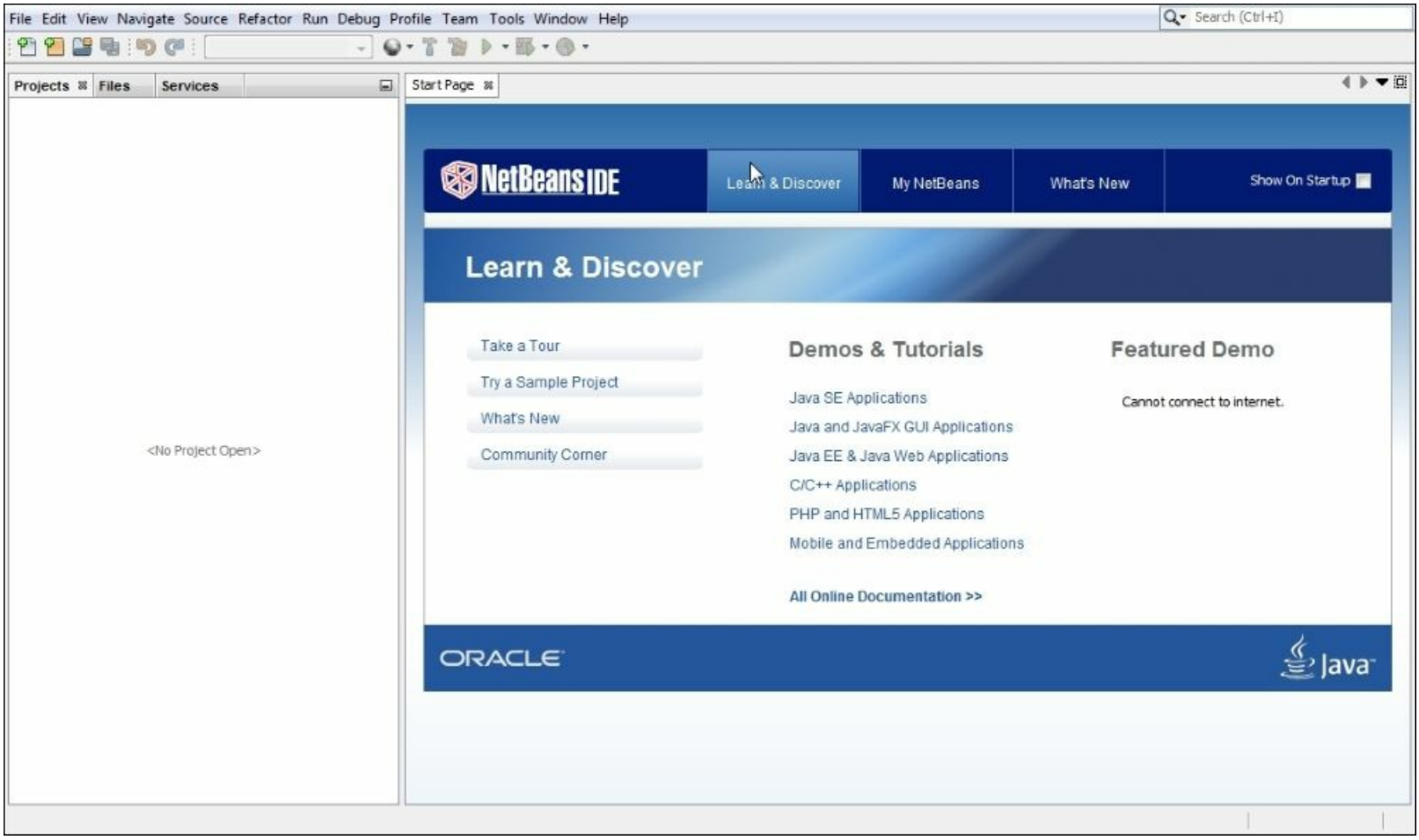
Note

JDK 7 or a later version is required for installing and running the All NetBeans Bundles. You can download a standalone JDK 8 from http://www.oracle.com/technetwork/java/javase/downloads/index.html.



NetBeans Bundles

Once NetBeans the IDE is installed, start the NetBeans IDE. NetBeans IDE should look as follows:



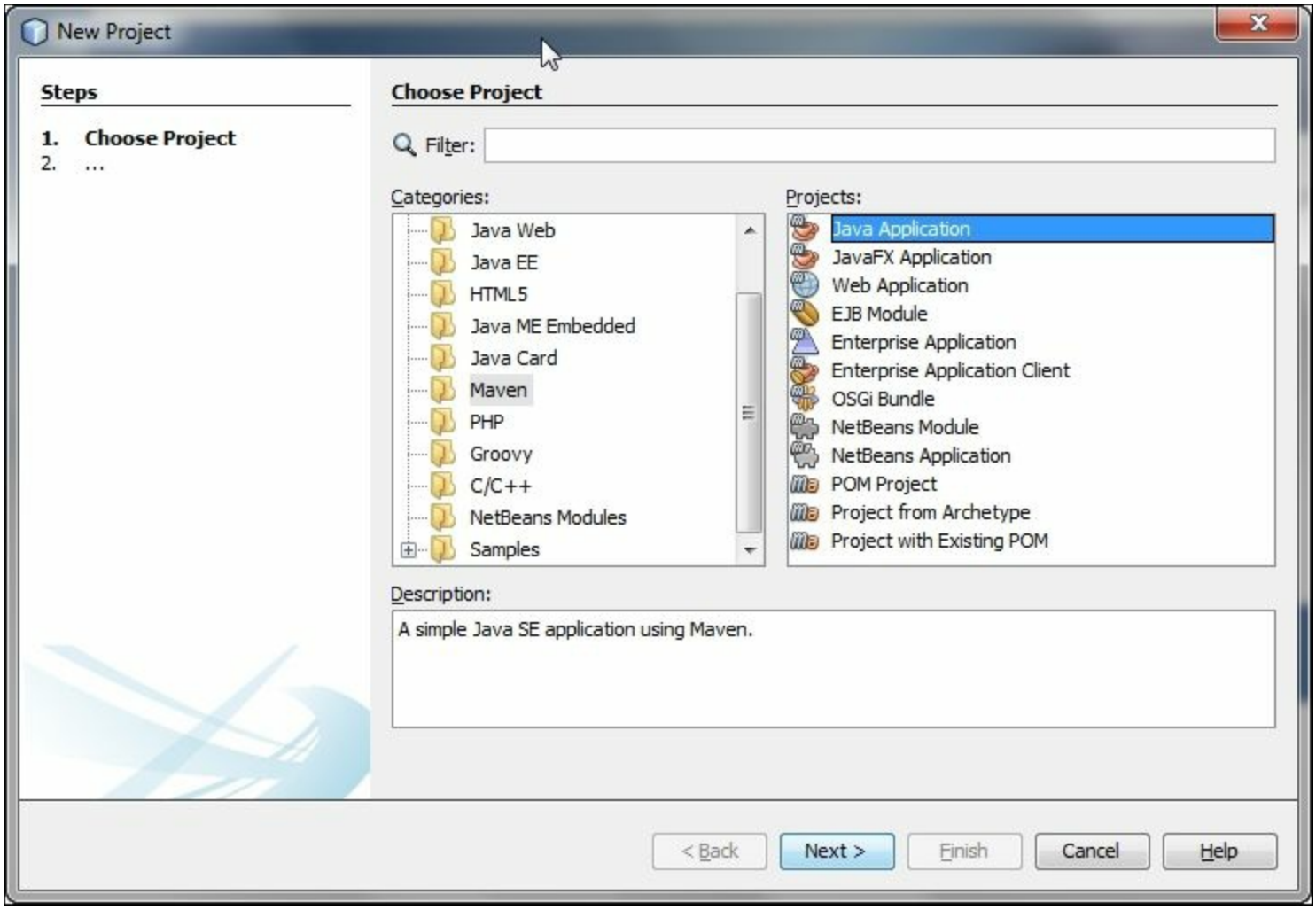
NetBeans start page

Maven and Gradle are both Java build tools. They add dependent libraries to your project, compile your code, set properties, build archives, or do many more related activities. Spring Boot or the Spring Cloud support both Maven and Gradle build tools. However, in this book we'll use the Maven build tool. Please feel free to use Gradle if you prefer.

Maven is already available in NetBeans IDE. Now, we can start a new Maven project to build our first REST app.

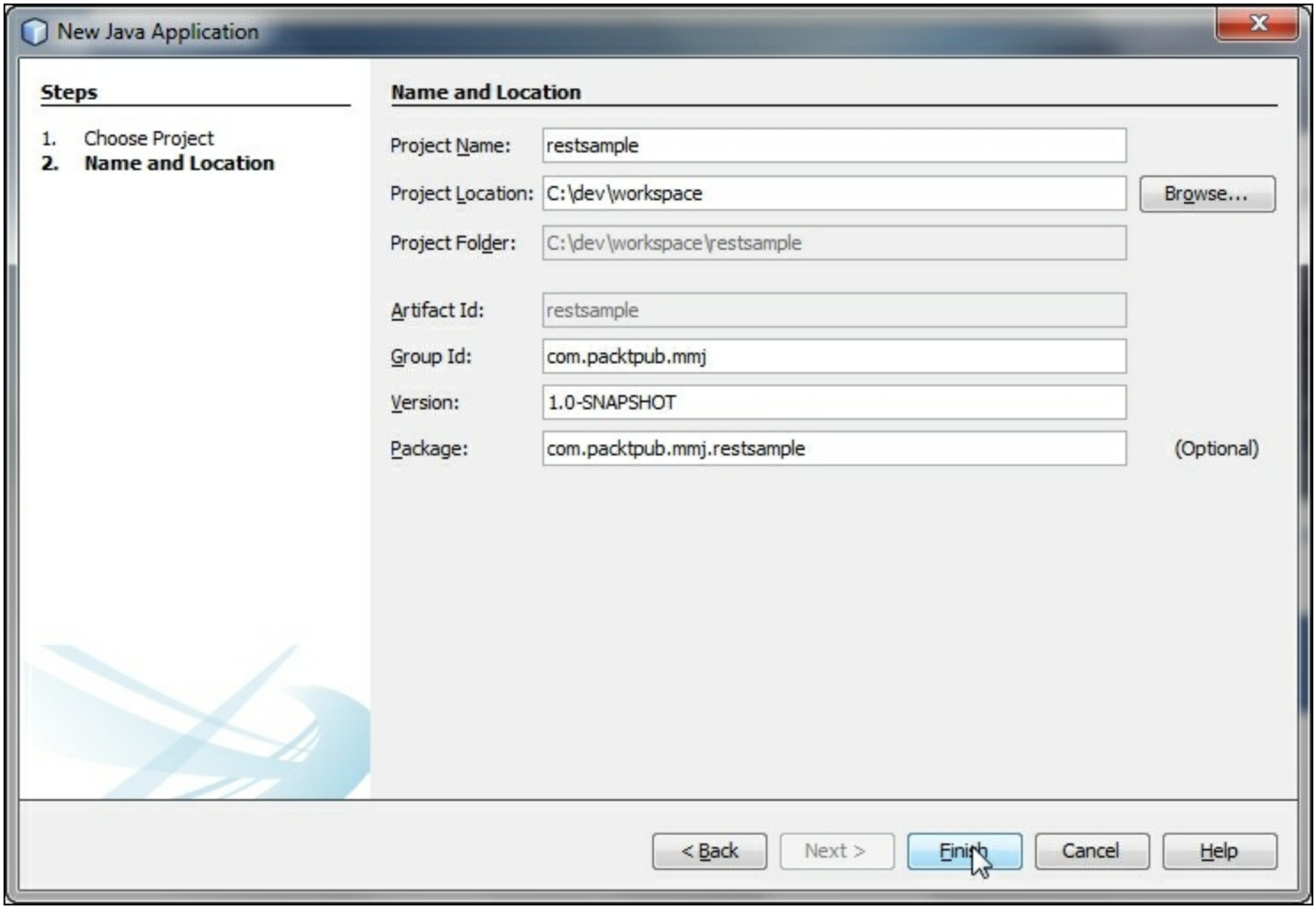
Steps for creating a new empty Maven project:

1. Click on **New Project** (*Ctrl* + *Shift* + *N*) under the **File** menu. It will open the new project wizard.
2. Select **Maven** in the **Categories** list. Then, select **Java Application** in the **Projects** list (as shown in following screenshot). Then, click on the **Next** button:



New Project Wizard

3. Now, enter the the project name as `restsample`. Also, enter the other properties as shown in the following screenshot. Click on **Finish** once all the mandatory fields are entered:



NetBeans Maven project properties

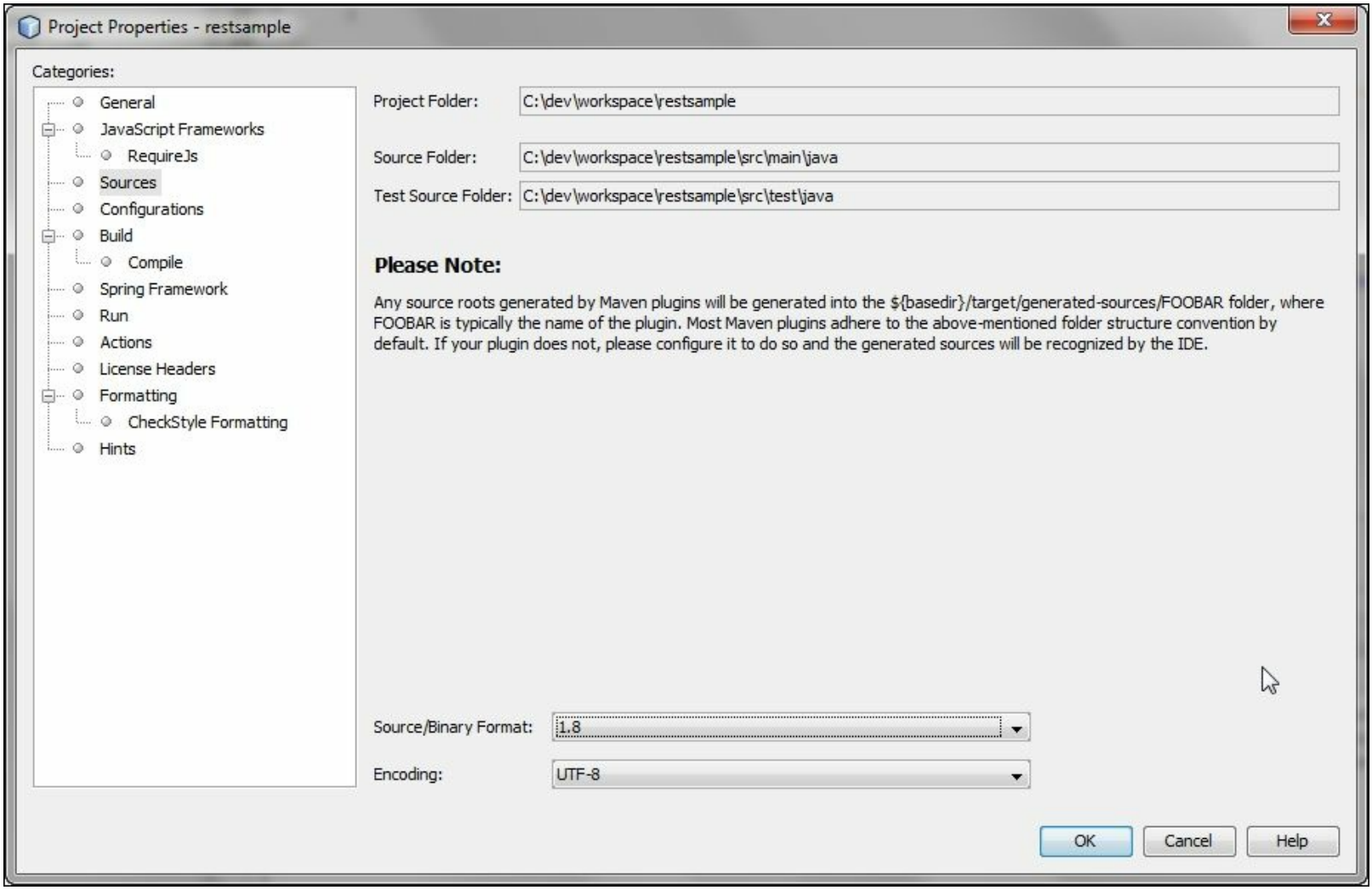
Tip

Aggelos Karalias has developed a helpful plugin for NetBeans IDE offering autocomplete support for Spring Boot configuration properties available at https://github.com/keevosh/nb-springboot-configuration-support. You can download it from his project page at http://keevosh.github.io/nb-springboot-configuration-support/.

You could also use Spring Tool Suite IDE (https://spring.io/tools) from Pivotal instead of NetBeans IDE. It's a customized all-in-one Eclipse-based distribution that makes application development easy.

After finishing all the the preceding steps, NetBeans will display a newly created Maven project. You will use this project for creating the sample rest application using Spring Boot.

To use Java 8 as a source, set the **Source/Binary Format** to **1.8** as shown in the following screenshot:



NetBeans Maven project properties

References

- **Spring Boot**: http://projects.spring.io/spring-boot/
- **Download NetBeans**: https://netbeans.org/downloads
- **Representational State Transfer** (**REST**): Chapter 5 of Roy Thomas Fielding Ph.D. dissertation "Architectural Styles and the Design of Network-based Software Architectures" - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- **REST**: https://en.wikipedia.org/wiki/Representational_state_transfer
- **Maven**: https://www.apache.org/
- **Gradle**: http://gradle.org/

Summary

In this chapter, you have explored various aspects of setting up a development environment such as NetBeans IDE setup and installation, Maven configuration, Spring Boot configuration and so on.

You have also learned how to make use of Spring Boot to develop a sample REST service application. We learned how powerful Spring Boot is – it eases development so much that you only have to worry about the actual code, and not about the boilerplate code or configurations that you write. We have also packaged our code into a JAR with an embedded application container Jetty. It allows it to run and access the web application without worrying about the deployment.

In the next chapter, you will learn the **domain-driven design** (**DDD**) using a sample project that can be used across the rest of the chapters. We'll use the sample project **online table reservation system** (**OTRS**) to go through various phases of microservices development and understand the DDD. After completing Chapter 3, *Domain-Driven Design* you will learn the fundamentals of DDD. You will understand how to practically use the DDD by design sample services. You will also learn to design the domain models and REST services on top of it.

Chapter 3. Domain-Driven Design

This chapter sets the tone for rest of the chapters by referring to one sample project. The sample project will be used to explain different microservices concepts from here onwards. This chapter uses this sample project to drive through different combinations of functional and domain services or apps to explain the **domain-driven design** (**DDD**). It will help you to learn the fundamentals of DDD and its practical usage. You will also learn the concepts of designing domain models using REST services.

This chapter covers the following topics:

- Fundamentals of DDD
- How to design an application using DDD
- Domain models
- A sample domain model design based on DDD

A good software design is as much the key to the success of a product or services as the functionalities offered by it. It carries equal weight to the success of product; for example, Amazon.com provides the shopping platform but its architecture design makes it different from other similar sites and contributes to its success. It shows how important a software or architecture design is for the success of a product/service. DDD is one of the software design practices and we'll explore it with various theories and practical examples.

DDD is a key design practice that helps to design the microservices of the product that you are developing. Therefore, we'll first explore DDD before jumping into microservices development. DDD uses multilayered architecture as one of its building blocks. After learning this chapter, you will understand the importance of DDD for microservices development.

Domain-driven design fundamentals

An enterprise or cloud application solves business problems and other real world problems. These problems cannot be resolved without knowledge of the domain. For example, you cannot provide a software solution for a financial system such as online stock trading if you don't understand the stock exchanges and their functioning. Therefore, having domain knowledge is a must for solving problems. Now, if you want to offer a solution using software or apps, you need to design it with the help of domain knowledge. When we combine the domain and software design, it offers a software design methodology known as DDD.

When we develop software to implement real world scenarios offering the functionalities of a domain, we create a model of the domain. A model is an abstraction or a blueprint of the domain.

Note

Eric Evans coined the term DDD in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, published in 2004.

Designing this model is not rocket science but it does take a lot of effort, refining and input from domain experts. It is the collective job of software designers, domain experts, and developers. They organize information, divide it into smaller parts, group them logically and create modules. Each module can be taken up individually and can be divided using a similar approach. This process can be followed until we reach the unit level or we cannot divide it any further. A complex project may have more of such iterations and similarly a simple project could have just a single iteration of it.

Once a model is defined and well documented, it can move onto the next stage – code design. So, here we have a software design – a Domain Model and code design – and code implementation of the Domain Model. The Domain Model provides a high level of architecture of a solution (software/app) and the code implementation gives the domain model a life, as a working model.

DDD makes design and development work together. It provides the ability to develop software continuously while keeping the design up to date based on feedback received from the development. It solves one of the limitations offered by Agile and Waterfall methodologies making software maintainable including design and code, as well as keeping app minimum viable.

Design-driven development involves a developer from the initial stage and all meetings where software designers discuss the domain with domain experts in the modeling process. It gives developers the right platform to understand the domain and provides the opportunity to share early feedback of the Domain Model implementation. It removes the bottleneck that appears in later stages when stockholders waits for deliverables.

Building blocks

This section explains the ubiquitous language used and why it is required, the different patterns to be used in model-driven design and the importance of multilayered architecture.

Ubiquitous language

As we have seen, designing a model is the collective effort of software designers, domain experts, and developers and, therefore, it requires a common language to communicate. DDD makes it necessary to use common language and the use of ubiquitous language. Domain models use ubiquitous language in their diagrams, descriptions, presentations, speeches, and meetings. It removes the misunderstanding, misinterpretation and communication gap among them.

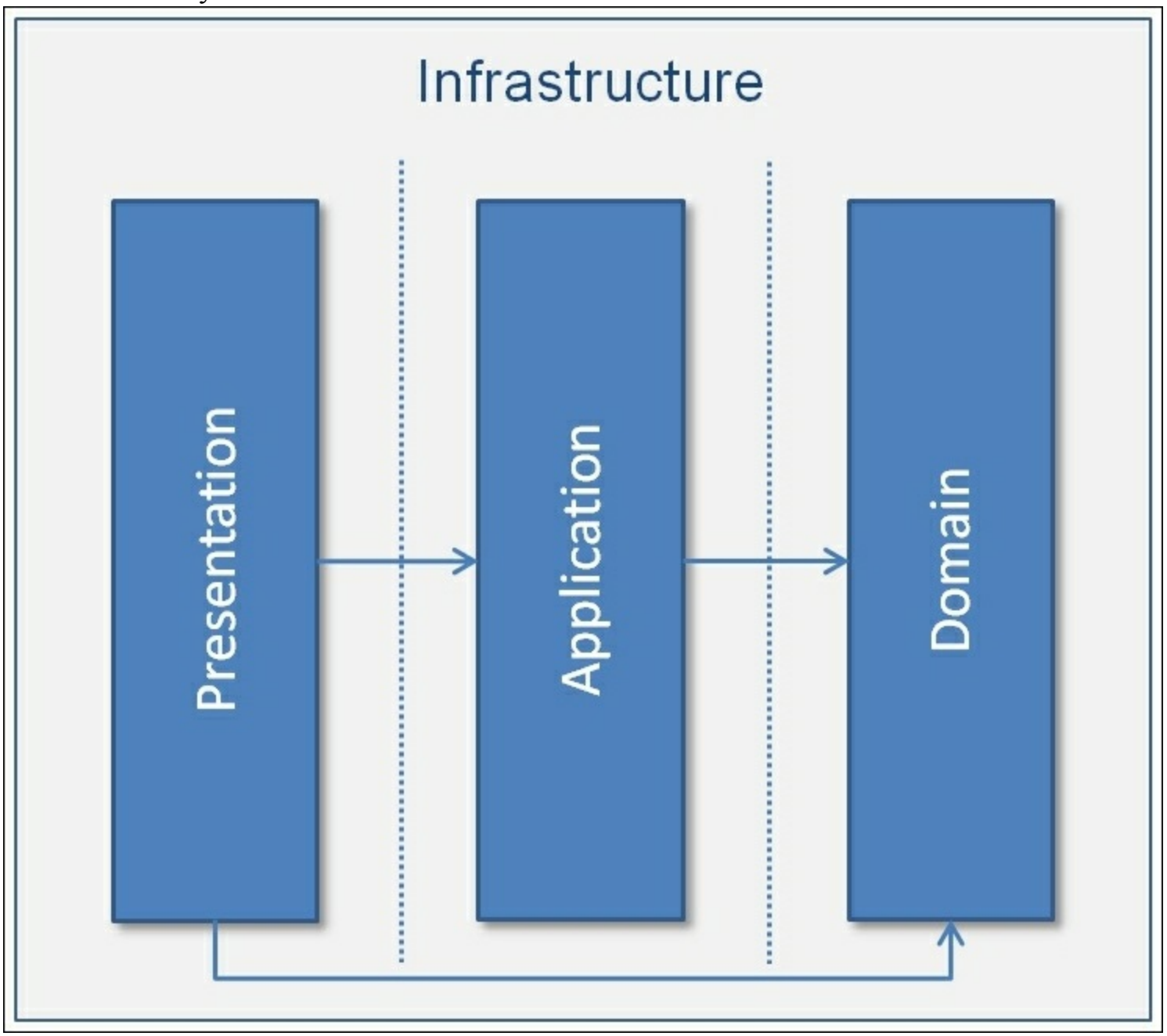
Unified Model Language (**UML**) is widely used and very popular when creating models. It also carries few limitations, for example when you have thousands of classes drawn of a paper, it's difficult to represent class relationships and also understand their abstraction while taking a meaning out of it. Also UML diagrams do not represent the concepts of a model and what objects are supposed to do.

There are other ways to communicate the domain model such as – documents, code, and so on.

Multilayered architecture

Multilayered architecture is a common solution for **DDD**. It contains four layers:

1. Presentation layer or **User Interface** (**UI**).
2. Application layer.
3. Domain layer.
4. Infrastructure layer.



Layered architecture

You can see here that only the domain layer is responsible for the domain model and others are related to other components such as UI, app logic and so on. This layered architecture is very

important. It keeps domain-related code separate from other layers.

In this multilayered architecture, each layer contains its respective code and it helps to achieve loose coupling and avoid mixing code from different layers. It also help the product/service's long term maintainability and the ease of enhancements as the change of one layer code does not impact on other components if the change is intended for the respective layer only. Each layer can be switched with another implementation easily with multitier architecture.

Presentation layer

This layer represents the UI and provides the user interface for the interaction and information display. This layer could be a web application, mobile app or a third-party application consuming your services.

Application layer

This layer is responsible for application logic. It maintains and coordinates the overall flow of the product/service. It does not contain business logic or UI. It may hold the state of application objects like tasks in progress. For example, your product REST services would be the part of this application layer.

Domain layer

The domain layer is a very important layer as it contains the domain information and business logic. It holds the state of the business object. It persists the state of the business objects, and communicates these persisted states to the infrastructure layer.

Infrastructure layer

This layer provides support to all the other layers and is responsible for communication among the other layers. It contains the supporting libraries that are used by the other layers. It also implements the persistence of business objects.

To understand the interaction of the different layers, let us take an example of table booking at a restaurant. The end user places a request for a table booking using UI. UI passes the request to the application layer. The application layer fetches the domain objects such as the restaurant, the table with a date and so on from the domain layer. The domain layers fetch these existing persisted objects from the infrastructure and invoke relevant methods to make the booking and persists them back to infrastructure layer. Once, domain objects are persisted, application layer shows the booking confirmation to end user.

Artifacts of domain-driven design

There are different artifacts used in DDD to express, create, and retrieve domain models.

Entities

There are certain categories of objects that are identifiable and remain same throughout the states of the product/services. These objects are NOT defined by its attributes, but by its identity and thread of continuity. These are known as entities.

It sounds pretty simple but carries complexity. You need to understand how we can define the entities. Let's take an example for table booking system, if we have a `restaurant` class with attributes such as restaurant name, address, phone number, establishment data, and so on. We can take two instances of the `restaurant` class that are not identifiable using the restaurant name, as there could be other restaurants with the same name. Similarly, if we go by any other single attributes we will not find any attributes that can singularly identify a unique restaurant. If two restaurants have all the same attribute values, these are the same and are interchangeable with each other. Still, these are not the same entities as both have different references (memory addresses).

Conversely, let's take a class of US citizen. Each citizen has his own social security number. This number is not only unique but remains unchanged throughout the life of the citizen and assures continuity. This citizen object would exist in the memory, would be serialized, and would be removed from the memory and stored in the database. It even exists after the person is dead. It will be kept in the system as long as system exists. A citizen's social security number remains the same irrespective of its representation.

Therefore, creating entities in a product means creating identity. So, now give an identity to any restaurant in the previous example, then either use a combination of attributes such as restaurant name, establishment date and street, or add an identifier such as `restaurant_id` to identify it. This is the basic rule that two identifiers cannot be same. Therefore, when we introduce an identifier for any entity we need to be sure of it.

There are different ways to create a unique identity for objects described as follows:

- Using the primary key in a table.
- Using an automated generated ID by a domain module. A domain program generates the identifier and assigns it to objects that are being persisted among different layers.
- A few real life objects carry user-defined identifiers themselves. For example each country has its own country codes for dialing ISD calls.
- An attribute or combination of attributes can also be used for creating an identifier as explained for the preceding `restaurant` object.

Entities are very important for domain models, therefore, they should be defined from the initial stage of the modeling process. When an object can be identified by its identifier and not by its attributes, a class representing these objects should have a simple definition and care should be taken with the life

cycle continuity and identity. It's imperative to say that it is a requirement of identifying objects in this class that have the same attribute values. A defined system should return a unique result for each object if queried. Designers should take care that the model must define what it means to be the same thing.

Value objects

Entities have traits such as, identity, a thread of continuity, and attributes that do not define their identity. **Value objects** (**VOs**) just have attributes and no conceptual identity. A best practice is to keep value Objects as immutable objects. If possible, you should even keep entity objects immutable too.

Entity concepts may bias you to keep all objects as entities, a uniquely identifiable object in the memory or database with life cycle continuity, but there has to be one instance for each object. Now, let's say you are creating customers as entity objects. Each customer object would represent the restaurant guest and this cannot be used for booking orders for other guests. This may create millions of customer entity objects in the memory if millions of customers are using the system. There are not only millions of uniquely identifiable objects that exist in the system, but each object is being tracked. Both tracking and creating identity is complex. A highly credible system is required to create and track these objects, which is not only very complex but also resource heavy. It may result in system performance degradation. Therefore, it is important to use value objects instead of using entities. The reasons are explained in the next few paragraphs.

Applications don't always needs to have an identifiable customer object and be trackable. There are cases when you just need to have some or all attributes of the domain element. These are the cases when value objects can be used by the application. It makes things simple and improves the performance.

Value objects can be created and destroyed easily, owing to the absence of identity. This simplifies the design – it makes value objects available for garbage collection if no other object has referenced them.

Let's discuss the value object's immutability. Value objects should be designed and coded as immutable. Once they are created they should never be modified during their life-cycle. If you need a different value of the VO or any of its objects, then simply create a new value object, but don't modify the original value object. Here, immutability carries all the significance from **object-oriented programming** (**OOP**). A value object can be shared and used without impacting on its integrity if and only if it is immutable.

Frequently asked questions

1. Can a value object contain another value object?

   Yes, it can
2. Can a value object refer to another value object or entity?

Yes, it can

3. Can I create a value object using the attributes of different value objects or entities?

   Yes, you can

Services

While creating the domain model you may encounter various situations, where behavior may not be related to any object specifically. These behaviors can be accommodated in service objects.

Ubiquitous language helps you to identify different objects, identities or value objects with different attributes and behaviors during the process of domain modeling. During the course of creating the domain model, you may find different behaviors or methods that do not belong to any specific object. Such behaviors are important so cannot be neglected. You can also not add them to entities or value objects. It would spoil the object to add behavior that does not belong to it. Keep in mind, that behavior may impact on various objects. The use of object-oriented programming makes it possible to attach to some objects; this is known as a service.

Services are common in technical frameworks. These are also used in domain layers in DDD. A service object does not have any internal state, the only purpose of it is to provide a behavior to the domain. Service objects provides behaviors that cannot be related with specific entities or value objects. Service objects may provide one or more related behaviors to one or more entities or value objects. It is a practice to define the services explicitly in the domain model.

While creating the services, you need to tick all the following points:

- Service objects' behavior performs on entities and value objects but it does not belong to entities or value objects
- Service objects' behavior state is not maintained and hence they are stateless
- Services are part of the domain model

Services may exist in other layers also. It is very important to keep domain layer services isolated. It removes the complexities and keeps the design decoupled.

Lets take an example where a restaurant owner wants to see the report of his monthly table booking. In this case, he will log in as an admin and click the **Display Report** button after providing the required input fields such as duration.

Application layers pass the request to the domain layer that owns the report and templates objects, with some parameters such as report ID and so on. Reports get created using the template and data is fetched from either the database or other sources. Then the application layer passes through all the parameters including the report ID to business layer. Here, a template needs to be fetched from the database or other source to generate the report based on the ID. This operation does not belong to either the report object or the template object. Therefore a service object is used that performs this operation to retrieve the required template from the DB.

Aggregates

Aggregate domain pattern is related to the object's life cycle and defines ownership and boundaries.

When, you reserve a table in your favorite restaurant online, using any app, you don't need to worry about the internal system and process that takes places to book your reservation such as searching the available restaurants, then the available tables during the given date, time, and so on and so forth. Therefore, you can say that a reservation app is an aggregate of several other objects and works as a root for all the other objects for a table reservation system.

This root should be an entity that binds collections of objects together. It is also called the aggregate root. This root object does not pass any reference of inside objects to external worlds and protects the changes performed in internal objects.

We need to understand why aggregators are required. A domain model can contains large numbers of domain objects. The bigger the application functionalities and size and the more complex its design, the greater number of objects will be there. A relationship exists among these objects. Some may have a many-to-many relationship, a few may have a one-to-many relationship and others may have a one-to-one relationship. These relationships are enforced by the model implementation in the code or in the database that ensures that these relationships among the objects are kept intact. Relationships are not just unidirectional, they can also be bi-directional. They can also increase in complexity.

The designer's job is to simplify these relationships in the model. Some relationships may exist in a real domain, but may not be required in the domain model. Designers need to ensure that such relationships do not exist in the domain model. Similarly, multiplicity can be reduced by these constraints. One constraint may do the job where many objects satisfy the relationship. It is also possible that a bidirectional relationship could be converted into a unidirectional relationship.

No matter how much simplification you input, you may still end up with relationships in the model. These relationships need to be maintained in the code. When one object is removed, the code should remove all the references of this object from other places. For example, a record removal from one table needs to be addressed wherever it has references in the form of foreign keys and such to keep the data consistent and maintain its integrity. Also invariants (rules) need to be forced and maintained whenever data changes.

Relationships, constraints, and invariants bring a complexity that requires an efficient handling in code. We find the solution by using the aggregate represented by the single entity known as the root that is associated with the group of objects that maintains consistency with respect to data changes.

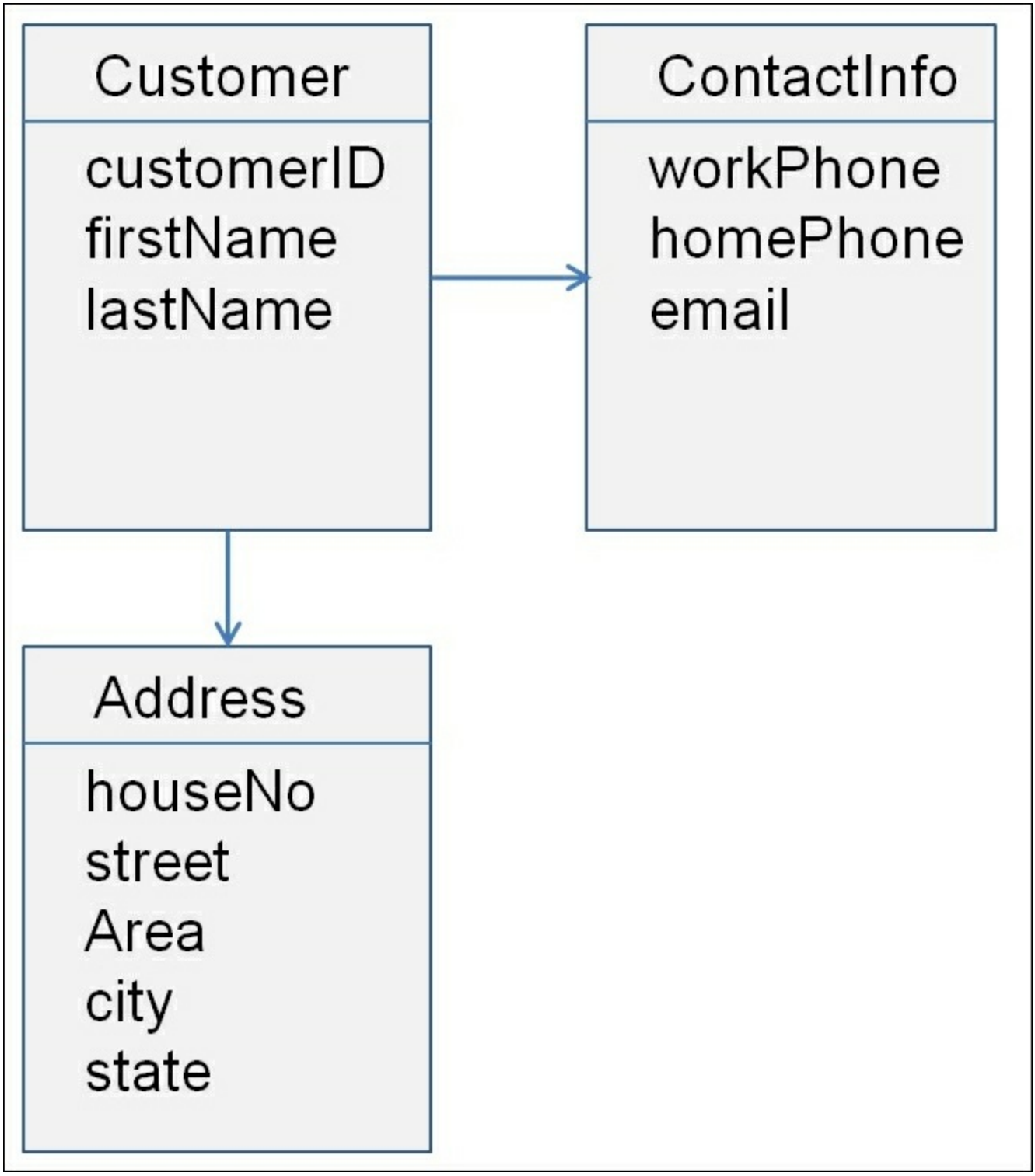
This root is the only object that is accessible from outside, so this root element works as a boundary gate that separates the internal objects from the external world. Roots can refer to one or more inside objects and these inside objects can have references to other inside objects that may or may not have relationships with the root. However, outside objects can also refer to the root and not to any inside objects.

An aggregate ensures data integrity and enforces the invariant. Outside objects cannot make any change to inside objects they can only change the root. However, they can use the root to make a change inside the object by calling exposed operations. The root should pass the value of inside objects to outside objects if required.

If an aggregate object is stored in the database then the query should only return the aggregate object. Traversal associations should be used to return the object when it is internally linked to the aggregate root. These internal objects may also have references to other aggregates.

An aggregate root entity holds its global identity and hold local identities inside their entities.

An easy example of an aggregate in the table booking system is the customer. Customers can be exposed to external objects and their root object contains their internal object address and contact information. When requested, the value object of internal objects like address can be passed to external objects:



The customer as an aggregate

Repository

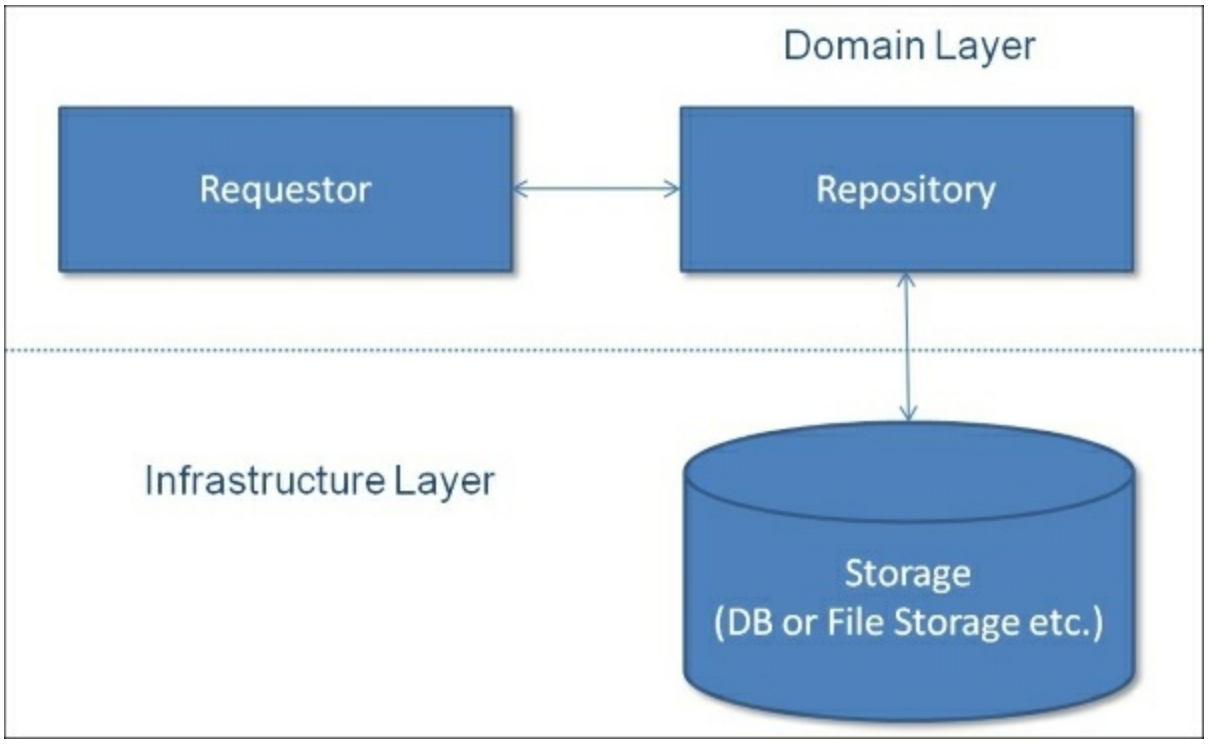
In a domain model, at a given point in time, many domain objects may exist. Each object may have its own life cycle from the creation of objects to their removal or persistence. Whenever any domain operation needs a domain object, it should retrieve the reference of the requested object efficiently. It would be very difficult if you didn't maintain all the available domain objects in a central object that carries the references of all the objects and is responsible for returning the requested object

reference. This central object is known as the repository.

The repository is a point that interacts with infrastructures such as the database or file system. A repository object is the part of the domain model that interacts with storage such as database, external sources, and so on to retrieve the persisted objects. When a request is received by the repository for an object's reference, it returns the existing object's reference. If the requested object does not exist in the repository then it retrieves the object from storage. For example, if you need a customer, you would query the repository object to provide the customer with ID `31`. The repository would provide the requested customer object if it is already available within the repository, and if not would query the persisted stores such as the database, fetch it and provide its reference.

The main advantage of using the repository is having a consistent way to retrieve objects where the requestor does not need to interact directly with the storage such as the database.

A repository may query objects from various storage types such as one or more databases, filesystems or factory repositories and so on. In such cases, a repository may have strategies that also point to different sources for different object types or categories:



Repository object flow

As shown in the repository object flow diagram, the repository interacts with the infrastructure layer and this interface is part of the domain layer. The requestor may belong to a domain layer or an application layer. The repository helps the system to manage the life cycle of domain objects.

Factory

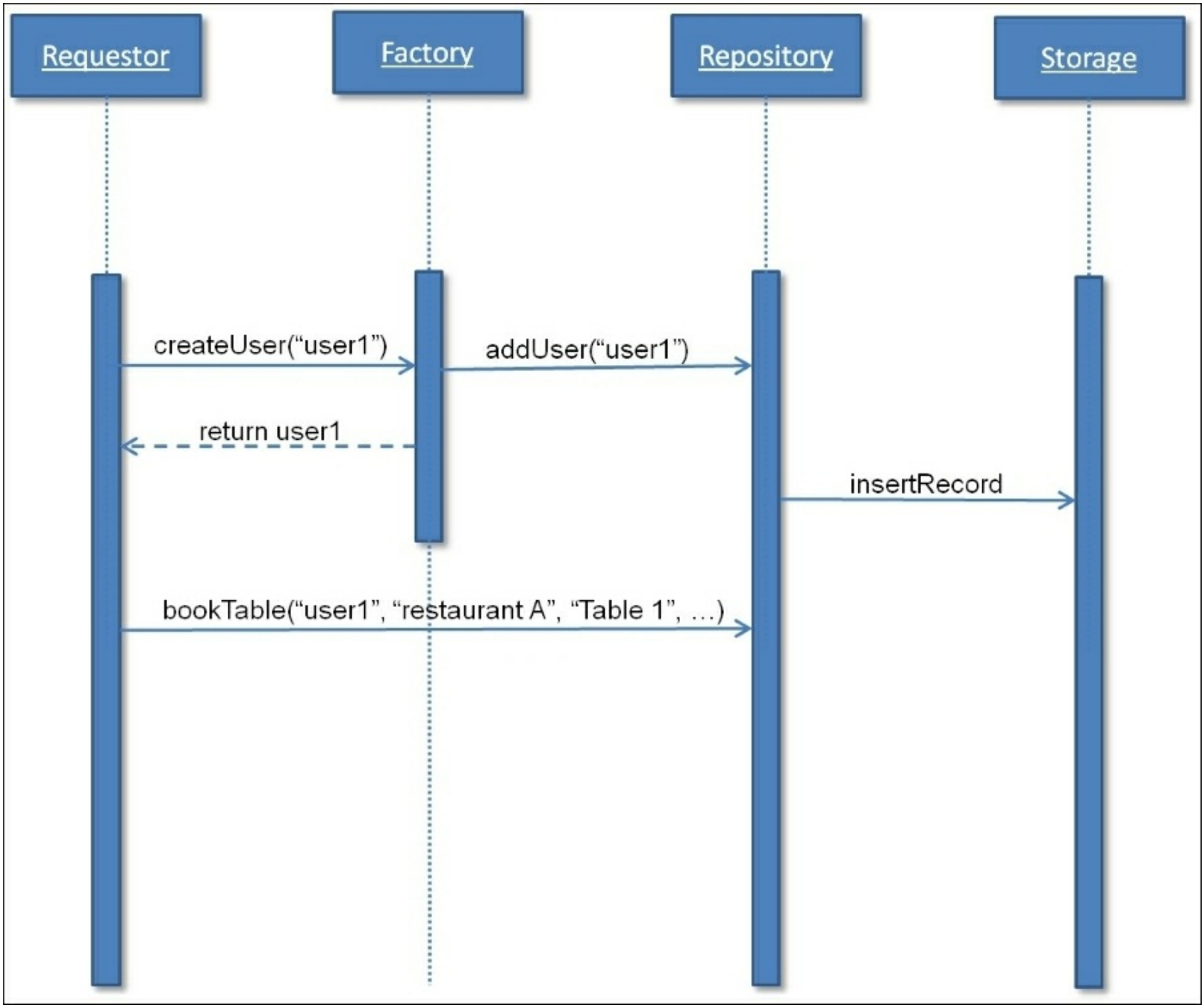
The factory is required when a simple constructor is not enough to create the object. It helps to create complex objects or an aggregate that involves the creation of other related objects.

A factory is also a part of the life cycle of domain objects as it is responsible for creating them. Factories and repositories are in some way related to each other as both refer to domain objects. The factory refers to newly created objects whereas the repository returns the already existing objects either from in the memory or from external storages.

Let us see how control flows using a user creation process app. Let's say that a user signs up with a username `user1`. This user creation first interacts with the factory, which creates the name `user1` and then caches it in the domain using the repository which also stores it in the storage for persistence.

When the same user logs in again, the call moves to the repository for a reference. This uses the storage to load the reference and pass it to the requestor.

The requestor may then use this `user1` object to book the table in a specified restaurant and at a specified time. These values are passed as parameters and a table booking record is created in storage using the repository:



Repository object flow

The factory may use one of the object oriented programming patterns such as the factory or abstract factory pattern for object creation.

Modules

Modules are the best way of separating related business objects. These are best suited to large projects where the size of domain objects is bigger. For the end user, it makes sense to divide the domain model into modules and set the relationship between these modules. Once you understand the modules and their relationship, you start to see the bigger picture of the domain model, and it is easier to drill down further and understand the model.

Modules also help in code that is highly cohesive or that maintains low coupling. Ubiquitous language can be used to name these modules. For the table booking system, we could have different modules such as user-management, restaurants and tables, analytics and reports, and reviews, and so on.

Strategic design and principles

An enterprise model is usually very large and complex. It may be distributed among different departments in an organization. Each department may have a separate leadership team, so working and designing together can create difficulty and coordination issues. In such scenarios, maintaining the integrity of the domain model is not an easy task.

In such cases, working on a unified model is not the solution and large enterprise models need to be divided into different submodels. These submodels contain the predefined accurate relationship and contract in minute detail. Each submodel has to maintain the defined contracts without any exception.

There are various principles that could be followed to maintain the integrity of the domain model, and these are listed as follows:

- Bounded context
- Continuous integration
- Context map
    - Shared kernel
    - Customer-supplier
    - Conformist
    - Anticorruption layer
    - Separate ways
    - Open host service
    - Distillation

Bounded context

When you have different submodels, it is difficult to maintain the code when all submodels are combined. You need to have a small model that can be assigned to a single team. You might need to collect the related elements and group them. Context keeps and maintains the meaning of the domain term defined for its respective submodel by applying this set of conditions.

These domain terms defines the scope of the model that creates the boundaries of the context.

Bounded context seems very similar to the module that you learned about in the previous section. In fact, module is part of the bounded context that defines the logical frame where a submodel takes place and is developed. Whereas, the module organizes the elements of the domain model and is visible in design document and the code.

Now, as a designer you would have to keep each submodel well-defined and consistent. In this way you can refactor the each model independently without affecting the other submodels. This gives the software designer the flexibility to refine and improve it at any point in time.

Now look at the table reservation example. When you started designing the system, you would have seen that the guest would visit the app and would request a table reservation in a selected restaurant, date, and time. Then, there is backend system that informs the restaurant about the booking information, and similarly, the restaurant would keep their system updated with respect to table bookings, given that tables can also be booked by the restaurant themselves. So, when you look at the systems finer points, you can see two domains models:

- The online table reservation system
- The offline restaurant management system

Both have their own bounded context and you need to make sure that the interface between them works fine.

Continuous integration

When you are developing, the code is scattered among many teams and various technologies. This code may be organized into different modules and has applicable bounded context for respective submodels.

This sort of development may bring with it a certain level of complexity with respect to duplicate code, a code break or maybe broken-bounded context. It happens not only because of the large size of code and domain model, but also because of other factors such as changes in team members, new members or not having a well documented model to name just a few of them.

When systems are designed and developed using DDD and Agile methodologies, domain models are not designed fully before coding starts and the domain model and its elements get evolved over a period of time with continuous improvements and refinement happening over the time.

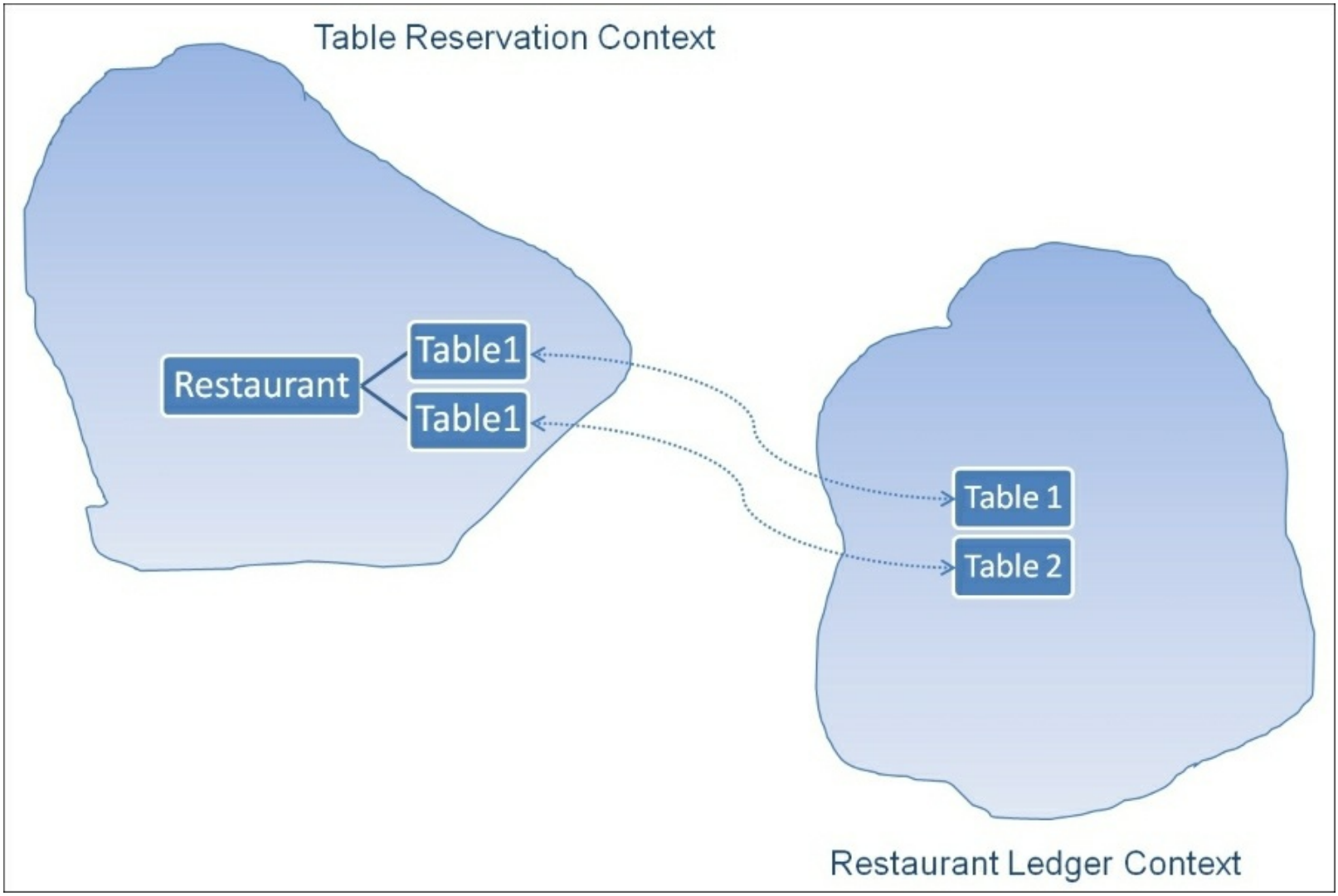
Therefore, integration continues and this is currently one of the key reasons for development today, so it plays a very important role. In continuous integration, code is merged frequently to avoid any breaks and issues with the domain model. Merged code not only gets deployed but it is also tested on a regular basis. There are various continuous integration tools available in the market that merge, build, and deploy the code at scheduled times. Organizations, these days, put more emphasis on the automation of continuous integration. Hudson, TeamCity, and Jenkins CI are a few of the popular tools available today for continuous integration. Hudson and Jenkins CI are open source tools and TeamCity is a proprietary tool.

Having a test suite attached to each build confirms the consistency and integrity of the model. A test suite defines the model from a physical point of view, whereas UML does it logically. It tells you about any error or unexpected outcome that requires a code change. It also helps to identify errors and anomalies in a domain model early.

Context map

The context map helps you to understand the overall picture of a large enterprise application. It shows how many bounded contexts are present in the enterprise model and how they are interrelated. Therefore we can say that any diagram or document that explains the bounded contexts and relationship between them is called a context map.

Context maps helps all team members, whether they are in the same team or in different team, to understand the high-level enterprise model in the form of various parts (bounded context or submodels) and relationships. This gives individuals a clearer picture about the tasks one performs and may allow him to raise any concern/question about the model's integrity:



Context map example

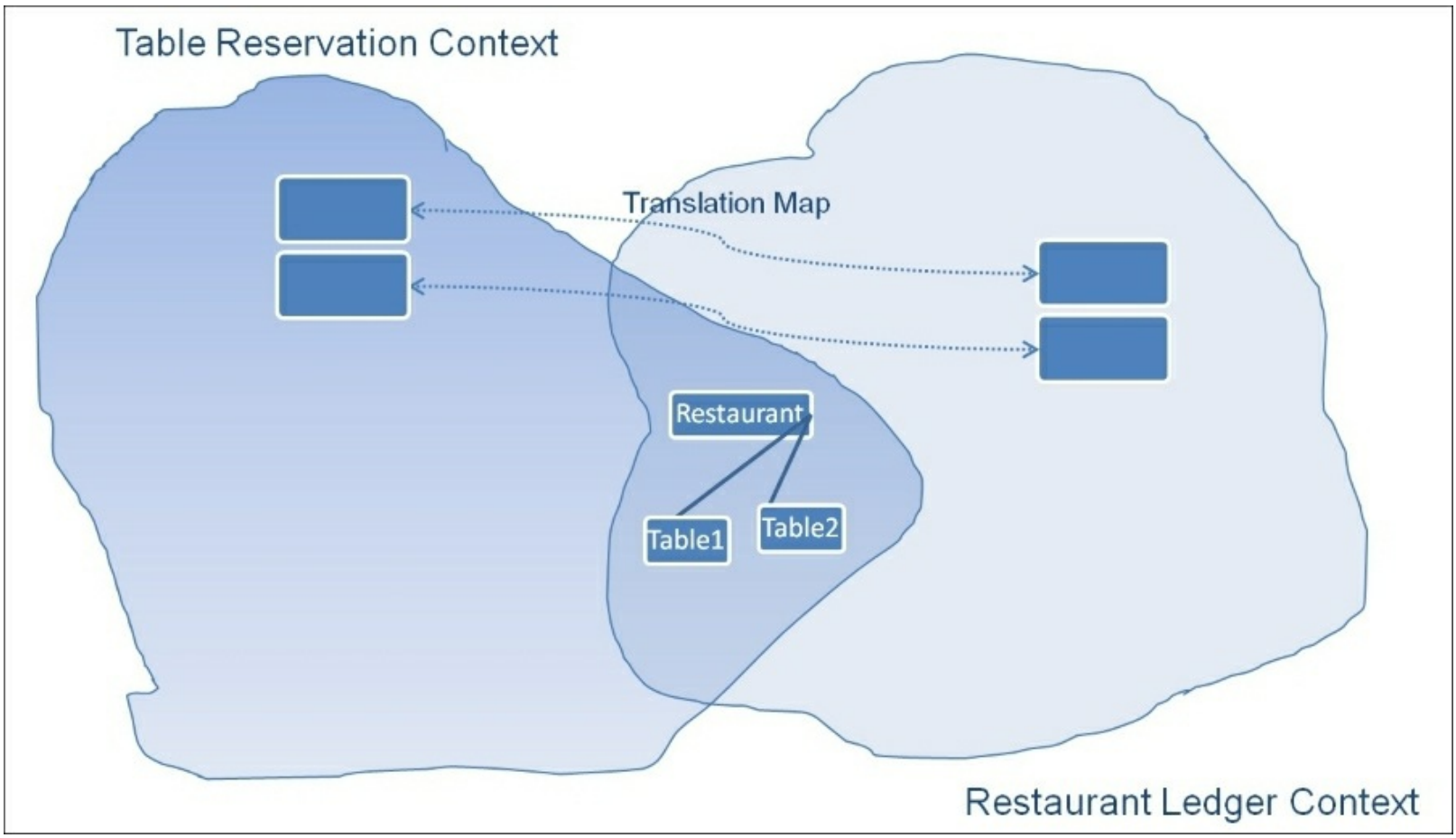
The context map example diagram is a sample of a context map. Here, **Table1** and **Table2** both appear in the **Table Reservation Context** and also in the **Restaurant Ledger Context**. The interesting thing is that **Table1** and **Table2** have their own respective concept in each bounded context. Here, ubiquitous language is used to name the bounded context as table reservation and

restaurant ledger.

In the following section, we will explore a few patterns that can be used to define the communication between different contexts in the context map.

Shared kernel

As the name suggests, one part of the bounded context is shared with the other's bounded context. As you can see below the **Restaurant** entity is being shared between the **Table Reservation Context** and the **Restaurant Ledger Context**:



Shared kernel

Customer-supplier

The customer-supplier pattern represents the relationship between two bounded contexts when the output of one bounded context is required for the other bounded context that is, one supplies the information to the other (known as the customer) who consumes the information.

In a real world example, a car dealer could not sell cars until the car manufacturer delivers them. Hence, in this domain-model, the car manufacturer is the supplier and the dealer is the customer. This relationship establishes a customer-supplier relationship because the output (car) of one bounded context (car-manufacturer) is required by the other bounded context (dealer).

Here, both customer and supplier teams should meet regularly to establish a contract and form the right protocol to communicate with each other.

Conformist

This pattern is similar to that of the customer and the supplier, where one needs to provide the contract and information while the other needs to use it. Here, instead of bounded context, actual teams are involved in having an upstream/downstream relationship.

Moreover, upstream teams do not provide for the needs of the downstream team because of their lack of motivation. Therefore, it is possible that the downstream team may need to plan and work on items which will never be available. To resolve such cases, either the customer team could develop their own models if the supplier provides information that is not worth enough. If the supplier provided information is really of worth or of partial worth, then the customer can use the interface or translators that can be used to consume the supplier-provided information with the customer's own models.

Anticorruption layer

The anticorruption layer remains part of a domain and it is used when a system needs data from external systems or from their own legacy systems. Here, anticorruption is the layer that interacts with external systems and uses external system data in the domain model without affecting the integrity and originality of the domain model.

For the most part, a service can be used as an anticorruption layer that may use a facade pattern with an adapter and translator to consume external domain data within the internal model. Therefore, your system would always use the service to retrieve the data. The service layer can be designed using the façade pattern. This would make sure that it would work with the domain model to provide the required data in a given format. The service could then also use the adapter and translator patterns that would make sure that whatever format and hierarchy the data is sent in, by external sources, the service would be provided in a desired format and the hierarchy would use adapters and translators.

Separate ways

When you have a large enterprise application and a domain where different domains have no common elements and it's made of large submodels that can work independently, this still works as a single application for an end user.

In such cases, a designer could create separate models that have no relationship and develop a small application on top of them. These small applications become a single application when merged together.

An employer's Intranet application that offers various small applications such as those that are HR-related, issue trackers, transport or intra-company social networks, is one such application where a designer could use the separate ways pattern.

It would be very challenging and complex to integrate applications that were developed using separate models. Therefore, you should take care before implementing this pattern.

Open host service

A translation layer is used when two submodels interact with each other. This translation layer is used when you integrate models with an external system. This works fine when you have one submodel that uses this external system. The open host service is required when this external system is being used by many submodels to remove the extra and duplicated code because then you need to write a translation layer for each submodel external system.

An open host service provides the services of an external system using a wrapper to all sub-models.

Distillation

As you know, distillation is the process of purifying liquid. Similarly, in DDD, distillation is the process that filters out the information that is not required, and keeps only the meaningful information. It helps you to identify the core domain and the essential concepts for your business domain. It helps you to filter out the generic concepts until you get the code domain concept.

Core domain should be designed, developed and implemented with the highest attention to detail, using the developers and designers, as it is crucial for the success of the whole system.

In our table reservation system example, which is not a large, or a complex domain application, it is not difficult to identify the core domain. The core domain here exists to share the real-time accurate vacant tables in the restaurants and allows the user to reserve them in a hassle free process.

Sample domain service

Let us create a sample domain service based on our table reservation system. As discussed in this chapter, the importance of an efficient domain layer is the key to successful products or services. Projects developed based on the domain layer are more maintainable, highly cohesive, and decoupled. They provide high scalability in terms of business requirement change and have a low impact on the design of other layers.

Domain-driven development is based on domain, hence it is not recommended that you use a top-down approach where the UI would be developed first, followed by the rest of the layers and finally the persistence layer, or a bottom-up approach where the persistence layer like the DB is designed first and then the rest of the layers, with the UI at last.

Having a domain model developed first, using the patterns described in this book, gives clarity to all team members functionality wise and an advantage to the software designer to build a flexible, maintainable and consistent system that helps the organization to launch a world class product with less maintenance costs.

Here, you will create a restaurant service that provides the feature to add and retrieve restaurants. Based on implementation, you can add other functionalities such as finding restaurants based on cuisine or on rating.

Start with the entity. Here, the restaurant is our entity as each restaurant is unique and has an identifier. You can use an interface or set of interfaces to implement the entity in our table reservation system. Ideally, if you go by the interface segregation principle, you will use a set of interfaces rather than a single interface.

Note

The **Interface Segregation Principle** (**ISP**): clients should not be forced to depend upon interfaces that they do not use.

Entity implementation

For the first interface you could have an abstract class or interface that is required by all the entities. For example if we consider ID and name, attributes would be common for all entities. Therefore, you could use the abstract class `Entity` as an abstraction of entity in your domain layer:

```
public abstract class Entity<T> {

    T id;
    String name;

}
```

Based on that you can also have another abstract class that inherits `Entity`, an abstract class:

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    public BaseEntity(T id, String name) {
        super.id = id;
        super.name = name;

    }
    ... (getter/setter and other relevant code)
}
```

Based on the preceding abstractions, we could create the `Restaurant` entity for restaurant management.

Now since we are developing the table reservation system, `Table` is another important entity in terms of the domain model. So, if we go by the aggregate pattern, `restaurant` would work as a root, and table would be internal to the `Restaurant` entity. Therefore, the `Table` entity would always be accessible using the `Restaurant` entity.

You can create the `Table` entity using the following implementation, and you can add attributes as you wish. For demonstration purpose only, basic attributes are used:

```
public class Table extends BaseEntity<BigInteger> {

    private int capacity;

    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
```

```
    public int getCapacity() {
        return capacity;
    }
}
```

Now, we can implement the aggregator `Restaurant` shown as follows. Here, only basic attributes are used. You could add as many you want or may add other features also:

```
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();
    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }
}
```

Repository implementation

Now, we can implement the repository pattern as learned in this chapter. To start with, you will first create the two interfaces `Repository` and `ReadOnlyRepository`. `ReadOnlyRepository` will be used to provide abstraction for read only operations whereas `Repository` abstraction will be used to perform all types of operations:

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

Based on this interface, we could create the abstraction of the repository that would do additional operations such as adding, removing, and updating:

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

Repository abstraction as defined previously could be implemented in a way that suits you to persist your objects. The change in persistence code, that is a part of infrastructure layer, won't impact on your domain layer code as the contract and abstraction are defined by the domain layer. The domain layer uses the abstraction classes and interfaces that remove the use of direct concrete class and provides the loose coupling. For demonstration purpose, we could simple use the map that remains in the memory to persist the objects:

```
public interface RestaurantRepository<Restaurant, String> extends
Repository<Restaurant, String> {

    boolean ContainsName(String name);
}

public class InMemRestaurantRepository implements
RestaurantRepository<Restaurant, String> {

    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
    }

    @Override
```

```
    public boolean ContainsName(String name) {
        return entities.containsKey(name);
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getName(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getName())) {
            entities.put(entity.getName(), entity);
        }
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Entity get(String id) {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }

}
```

Service implementation

In the same way as the preceding approach, you could divide the abstraction of domain service into two parts: main service abstraction and read only service abstraction:

```
public abstract class ReadOnlyBaseService<TE, T> {

    private Repository<TE, T> repository;

    ReadOnlyBaseService(Repository<TE, T> repository) {
        this.repository = repository;
    }
    ...
}
```

Now, we could use this `ReadOnlyBaseService` to create the `BaseService`. Here, we are using the dependency inject pattern via a constructor to map the concrete objects with abstraction:

```
public abstract class BaseService<TE, T> extends ReadOnlyBaseService<TE, T> {
    private Repository<TE, T> _repository;

    BaseService(Repository<TE, T> repository) {
        super(repository);
        _repository = repository;
    }

    public void add(TE entity) throws Exception {
        _repository.add(entity);
    }

    public Collection<TE> getAll() {
        return _repository.getAll();
    }
}
```

Now, after defining the service abstraction services, we could implement the `RestaurantService` in the following way:

```
public class RestaurantService extends BaseService<Restaurant, BigInteger> {

    private RestaurantRepository<Restaurant, String> restaurantRepository;

    public RestaurantService(RestaurantRepository repository) {
        super(repository);
        restaurantRepository = repository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurantRepository.ContainsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a product with
the name - %s", restaurant.getName()));
        }
```

```
        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Restaurant name cannot be null or empty
string.");
        }
        super.add(restaurant);
    }
}
```

Similarly, you could write the implementation for other entities. This code is a basic implementation and you might add various implementations and behaviors in the production code.

Summary

In this chapter, you have learned the fundamentals of DDD. You have also explored multilayered architecture and different patterns one can use to develop software using DDD. By this time, you might be aware that the domain model design is very important for the success of the software. At the end, there is also one domain service implementation shown using the restaurant table reservation system.

In the next chapter, you will learn how to use the design is used to implement the sample project. The explanation of the design of this sample project is derived from the last chapter and the DDD will be used to build the microservices. This chapter not only covers the coding, but also the different aspects of the microservices such as build, unit-testing, and packaging. At the end of the next chapter, the sample microservice project will be ready for deployment and consumption.

Chapter 4. Implementing a Microservice

This chapter takes you from the design stage to the implementation of our sample project – an **Online Table Reservation System** (**OTRS**). Here, you will use the same design explained in the last chapter and enhance it to build the µService. At the end of this chapter, you will not only have learned to implement the design, but also learned the different aspects of µServices – building, testing, and packaging. Although the focus is on building and implementing the Restaurant µService, you can use the same approach to build and implement other µServices used in the OTRS.

In this chapter, we will cover the following topics:

- OTRS overview
- Developing and implementing µService
- Testing

We will use the domain-driven design key concepts demonstrated in the last chapter. In the last chapter, you saw how domain-driven design is used to develop the domain model using core Java. Now, we will move from a sample domain implementation to a Spring framework-driven implementation. You'll make use of Spring Boot to implement the domain-driven design concepts and transform them from core Java to a Spring framework-based model.

In addition, we'll also use the Spring Cloud, which provides a cloud-ready solution. Spring Cloud also uses Spring Boot, which allows you to use an embedded application container relying on Tomcat or Jetty inside your service, which is packages as a JAR or as a WAR. This JAR is executed as a separate process, a µService that would serve and provide the response to all requests and, point to endpoints defined in the service.

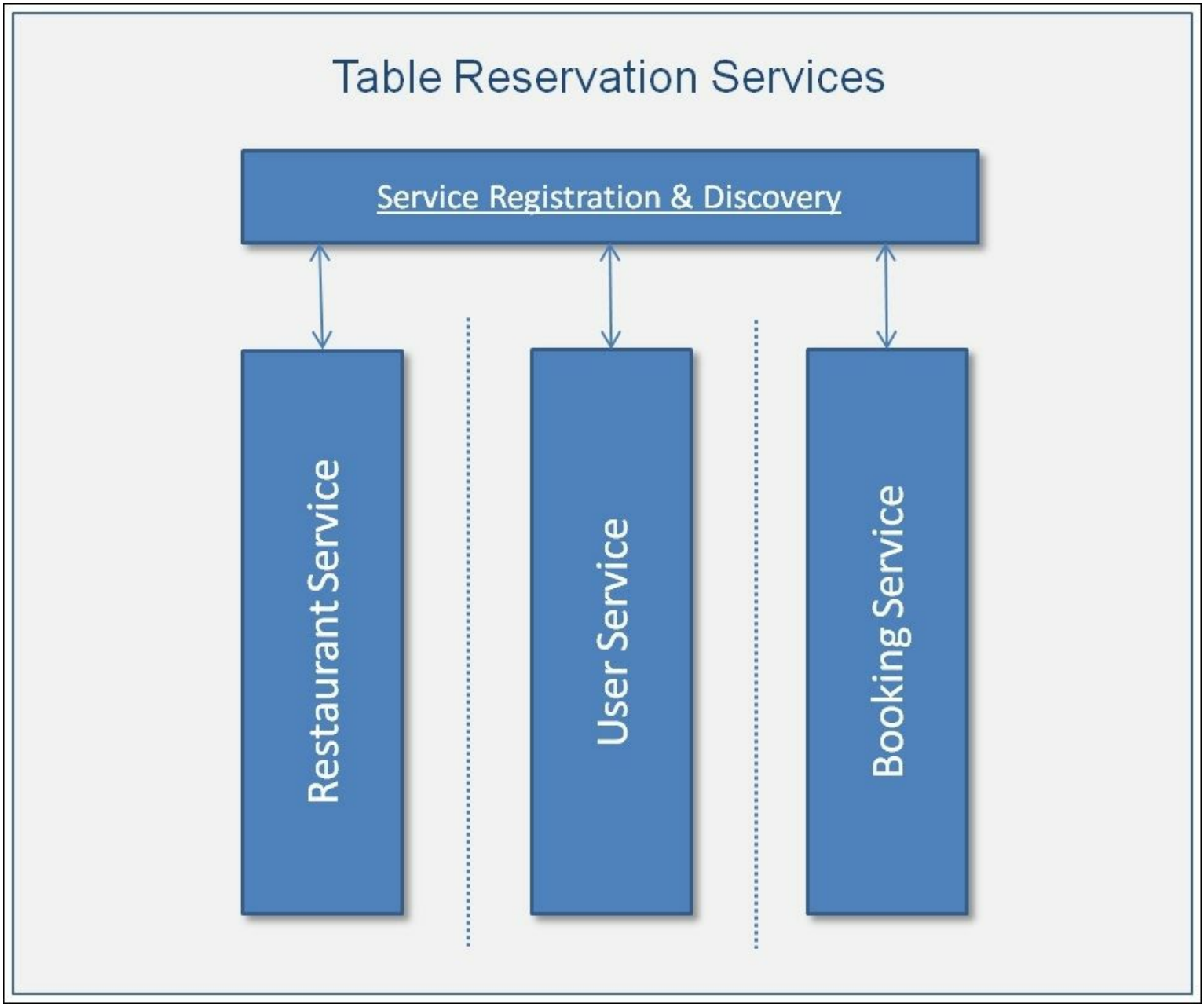
Spring Cloud can also be integrated easily with Netflix Eureka, a service registry and discovery component. The OTRS will use it for registration and the discovery of µServices.

OTRS overview

Based on µService principles, we need to have separate µServices for each functionality that can function independently. After looking at the OTRS, we can easily divide the OTRS into three main µServices – Restaurant service, Booking service, and User service. There can be other µServices that can be defined in the OTRS. Our focus is on these three µServices. The idea is to make them independent, including having their own separate databases.

We can summarize the functionalities of these services as follows:

- **Restaurant service**: This service provides the functionality for the Restaurant resource – **create, read, update, delete** (**CRUD**) operation and searching based on criteria. It provides the association between restaurants and tables. Restaurant would also provide the access to the Table entity.
- **User service**: This service, as the name suggests, allows the end user to perform CRUD operations on User entities.
- **Booking service**: This makes use of the Restaurant service and User service to perform CRUD operations on booking. It would use the Restaurant searching, its associated tables lookup and allocation based on table availability for a specified time duration. It creates the relationship between the Restaurant/Table and the User.



Different µServices, Registration and Discovery

The preceding diagram shows how each µService works independently. This is the reason µServices can be developed, enhanced, and maintained separately, without affecting others. These services can each have their own layered architecture and database. There is no restriction to use same technologies, frameworks, and languages to develop these services. At any given point in time, you can also introduce new µServices. For example, for accounting purposes, we can introduce an accounting service that can be exposed to Restaurant for book keeping. Similarly, analytics and reporting are other services that can be integrated and exposed.

For demonstration purposes, we will only implement the three services shown in the preceding diagram.

Developing and implementing µServices

We will use the domain-driven implementation and approach described in the last chapter to implement the µServices using Spring Cloud. Let's revisit the key artifacts:

- **Entities**: These are categories of objects that are identifiable and remain the same throughout the states of the product/services. These objects are NOT defined by their attributes, but by their identities and threads of continuity.

  Entities have traits such as identity, a thread of continuity, and attributes that do not define their identity. **Value Objects** (**VO**) just have the attributes and no conceptual identity. A best practice is to keep Value Objects as immutable objects. In the Spring framework, entities are pure POJOs, therefore we'll also use them as VO.
- **Services**: These are common in technical frameworks. These are also used in the Domain layer in domain-driven design. A Service object does not have an internal state; the only purpose of it is to provide the behavior to the domain. Service objects provide behaviors that cannot be related with specific entities or value objects. Service objects may provide one or more related behaviors to one or more entities or value objects. It is a best practice to define the Services explicitly in the domain model.
- **Repository object**: A Repository object is a part of the domain model that interacts with storage, such as databases, external sources and so on, to retrieve the persisted objects. When a request is received by the repository for an object reference, it returns the existing object reference. If the requested object does not exist in the repository, then it retrieves the object from storage.
- Each OTRS µService API represents a RESTful web service. The OTRS API uses HTTP verbs such as GET, POST, and so on, and a RESTful endpoint structure. Request and response payloads are formatted as JSON. If required, XML can also be used.

Restaurant µService

The Restaurant µService will be exposed to the external world using REST endpoints for consumption. We'll find the following endpoints in the Restaurant µService example. One can add as many endpoints as per the requirements:

Endpoint	GET /v1/restaurants/<Restaurant-Id>	
Parameters		
Name	**Description**	
Restaurant_Id	Path parameter that represents the unique restaurant associated with this ID	
Request		
Property	**Type**	**Description**
None		
Response		
Property	**Type**	**Description**
Restaurant	Restaurant object	Restaurant object that is associated with the given ID

Endpoint	GET /v1/restaurants/	
Parameters		
Name	Description	
None		
Request		
Property	**Type**	**Description**
Name	String	Query parameter that represents the name, or substring of the name, of the restaurant
Response		

Property	Type	Description
`Restaurants`	Array of restaurant objects	Returns all the restaurants whose names contain the given name value

`Endpoint`	`POST /v1/restaurants/`	
Parameters		
Name	**Description**	
None		
Request		
Property	**Type**	**Description**
`Restaurant`	Restaurant object	A JSON representation of the restaurant object
Response		
Property	**Type**	**Description**
`Restaurant`	Restaurant object	A newly created Restaurant object

Similarly, we can add various endpoints and their implementations. For demonstration purposes, we'll implement the preceding endpoints using Spring Cloud.

Controller class

The Restaurant Controller uses the `@RestController` annotation to build the restaurant service endpoints. We have already gone through the details of `@RestController` in Chapter 2, *Setting Up the Development Environment*. `@RestController` is a class-level annotation that is used for resource classes. It is a combination of `@Controller` and `@ResponseBody`. It returns the domain object.

API versioning

As we move forward, I would like to share with you that we are using the `v1` prefix on our REST endpoint. That represents the version of the API. I would also like to brief you on the importance of API versioning. Versioning APIs is important, because APIs change over time. Your knowledge and experience improves with time, which leads to changes to your API. A change of API may break existing client integrations.

Therefore, there are various ways of managing API versions. One of these is using the version in path or some use the HTTP header. The HTTP header can be a custom request header or an Accept header to represent the calling API version. Please refer to *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing, https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices, for more information.

```
@RestController
@RequestMapping("/v1/restaurants")
public class RestaurantController {

    protected Logger logger =
Logger.getLogger(RestaurantController.class.getName());

    protected RestaurantService restaurantService;

    @Autowired
    public RestaurantController(RestaurantService restaurantService) {
        this.restaurantService = restaurantService;
    }

    /**
     * Fetch restaurants with the specified name. A partial case-insensitive
     * match is supported. So <code>http://.../restaurants/rest</code> will find
     * any restaurants with upper or lower case 'rest' in their name.
     *
     * @param name
     * @return A non-null, non-empty collection of restaurants.
     */
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Collection<Restaurant>>
findByName(@RequestParam("name") String name) {

logger.info(String.format("restaurant-service findByName() invoked:{} for {} ",
restaurantService.getClass().getName(), name));
        name = name.trim().toLowerCase();
        Collection<Restaurant> restaurants;
        try {
            restaurants = restaurantService.findByName(name);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised findByName REST Call",
ex);
            return new ResponseEntity< Collection< Restaurant>>
(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return restaurants.size() > 0 ? new ResponseEntity< Collection<
Restaurant>>(restaurants, HttpStatus.OK)
                : new ResponseEntity< Collection< Restaurant>>
(HttpStatus.NO_CONTENT);
    }

    /**
     * Fetch restaurants with the given id.
     * <code>http://.../v1/restaurants/{restaurant_id}</code> will return
```

```
     * restaurant with given id.
     *
     * @param retaurant_id
     * @return A non-null, non-empty collection of restaurants.
     */
    @RequestMapping(value = "/{restaurant_id}", method = RequestMethod.GET)
    public ResponseEntity<Entity> findById(@PathVariable("restaurant_id") String
id) {

       logger.info(String.format("restaurant-service findById() invoked:{} for
{} ", restaurantService.getClass().getName(), id));
        id = id.trim();
        Entity restaurant;
        try {
            restaurant = restaurantService.findById(id);
        } catch (Exception ex) {
            logger.log(Level.SEVERE, "Exception raised findById REST Call", ex);
            return new ResponseEntity<Entity>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return restaurant != null ? new ResponseEntity<Entity>(restaurant,
HttpStatus.OK)
                : new ResponseEntity<Entity>(HttpStatus.NO_CONTENT);
    }

    /**
     * Add restaurant with the specified information.
     *
     * @param Restaurant
     * @return A non-null restaurant.
     * @throws RestaurantNotFoundException If there are no matches at all.
     */
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<Restaurant> add(@RequestBody RestaurantVO
restaurantVO) {

        logger.info(String.format("restaurant-service add() invoked: %s for %s",
restaurantService.getClass().getName(), restaurantVO.getName());

        Restaurant restaurant = new Restaurant(null, null, null);
        BeanUtils.copyProperties(restaurantVO, restaurant);
        try {
            restaurantService.add(restaurant);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised add Restaurant REST Call
"+ ex);
            return new ResponseEntity<Restaurant>
(HttpStatus.UNPROCESSABLE_ENTITY);
        }
        return new ResponseEntity<Restaurant>(HttpStatus.CREATED);
    }
}
```

Service classes

`RestaurantController` uses `RestaurantService`. `RestaurantService` is an interface that defines CRUD and some search operations and is defined as follows:

```
public interface RestaurantService {

    public void add(Restaurant restaurant) throws Exception;

    public void update(Restaurant restaurant) throws Exception;

    public void delete(String id) throws Exception;

    public Entity findById(String restaurantId) throws Exception;

    public Collection<Restaurant> findByName(String name) throws Exception;

    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>>
name) throws Exception;
}
```

Now, we can implement the `RestaurantService` we have just defined. It also extends the `BaseService` you created in the last chapter. We use `@Service` Spring annotation to define it as a service:

```
@Service("restaurantService")
public class RestaurantServiceImpl extends BaseService<Restaurant, String>
        implements RestaurantService {

    private RestaurantRepository<Restaurant, String> restaurantRepository;

    @Autowired
    public RestaurantServiceImpl(RestaurantRepository<Restaurant, String>
restaurantRepository) {
        super(restaurantRepository);
        this.restaurantRepository = restaurantRepository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Restaurant name cannot be null or empty
string.");
        }

        if (restaurantRepository.containsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a product with
the name - %s", restaurant.getName()));
        }

        super.add(restaurant);
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        return restaurantRepository.findByName(name);
```

```
    }

    @Override
    public void update(Restaurant restaurant) throws Exception {
        restaurantRepository.update(restaurant);
    }

    @Override
    public void delete(String id) throws Exception {
        restaurantRepository.remove(id);
    }

    @Override
    public Entity findById(String restaurantId) throws Exception {
        return restaurantRepository.get(restaurantId);
    }

    @Override
    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>>
name) throws Exception {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
}
```

Repository classes

The `RestaurantRepository` interface defines two new methods: the `containsName` and `findByName` methods. It also extends the `Repository` interface:

```
public interface RestaurantRepository<Restaurant, String> extends
Repository<Restaurant, String> {

    boolean containsName(String name) throws Exception;

    Collection<Restaurant> findByName(String name) throws Exception;
}
```

The `Repository` interface defines three methods: `add`, `remove`, and `update`. It also extends the `ReadOnlyRepository` interface:

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

The `ReadOnlyRepository` interface definition contains the `get` and `getAll` methods, which return Boolean values, Entity, and collection of Entity respectively. It is useful if you want to expose only a read-only abstraction of the repository:

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

Spring framework makes use of the `@Repository` annotation to define the repository bean that implements the repository. In the case of `RestaurantRepository`, you can see that a map is used in place of the actual database implementation. This keeps all entities saved in memory only. Therefore, when we start the service, we find only two restaurants in memory. We can use JPA for database persistence. This is the general practice for production-ready implementations:

```
@Repository("restaurantRepository")
public class InMemRestaurantRepository implements
RestaurantRepository<Restaurant, String> {
    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Big-O Restaurant", "1", null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("O Restaurant", "2", null);
        entities.put("2", restaurant);
    }

    @Override
    public boolean containsName(String name) {
        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
            //Exception Handler
        }
        return false;
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getId())) {
```

```
            entities.put(entity.getId(), entity);
        }
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        Collection<Restaurant> restaurants = new ArrayList();
        int noOfChars = name.length();
        entities.forEach((k, v) -> {
            if (v.getName().toLowerCase().contains(name.subSequence(0,
noOfChars))) {
                restaurants.add(v);
            }
        });
        return restaurants;
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Entity get(String id) {
        return entities.get(id);
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
}
```

Entity classes

The `Restaurant` entity, which extends `BaseEntity`, is defined as follows:

```
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();

    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, name: {}, capacity: {}", this.getId(),
this.getName(), this.getCapacity()));
        return sb.toString();
    }

}
```

Note

Since, we are using POJO classes for our entity definitions, we do not need to create a Value object in many cases. The idea is that the state of the object should not be persisted across.

The `Table` entity, which extends `BaseEntity`, is defined as follows:

```
public class Table extends BaseEntity<BigInteger> {

    private int capacity;

    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public int getCapacity() {
        return capacity;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, name: {}", this.getId(),
this.getName()));
        sb.append(String.format("Tables: {}" +
Arrays.asList(this.getTables())));
        return sb.toString();
    }

}
```

The `Entity` abstract class is defined as follows:

```
public abstract class Entity<T> {

    T id;
    String name;
```

```
    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

The `BaseEntity` abstract class is defined as follows. It extends the `Entity` abstract class:

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    private boolean isModified;
    private String name;

    public BaseEntity(T id, String name) {
        this.id = id;
        this.name = name;
    }

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public boolean isIsModified() {
        return isModified;
    }

    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
}
```

Booking and user services

We can use the `RestaurantService` implementation to develop the Booking and User services. The User service can offer the endpoint related to the User resource with respect to CRUD operations. The Booking service can offer the endpoints related to the Booking resource with respect to CRUD operations and the availability of table slots. You can find the sample code of these services on the Packt website.

Registration and Discovery service (Eureka service)

Spring Cloud provides state-of-the-art support to *Netflix Eureka*, a service registry and discovery tool. All services executed by you get listed and discovered by Eureka service, which it reads from the Eureka client Spring configuration inside your service project.

It needs a Spring Cloud dependency as shown here and a startup class with the `@EnableEurekaApplication` annotation in `pom.xml`:

Maven dependency:

```
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
```

Startup class:

The startup class **App** would run the Eureka service seamlessly by just using the `@EnableEurekaApplication` class annotation:

```
package com.packtpub.mmj.eureka.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Tip

Use `<start-class>com.packtpub.mmj.eureka.service.App</start-class>` under the `<properties>` tag in the `pom.xml` project.

Spring configurations:

Eureka Service also needs the following Spring configuration for Eureka Server configuration (`src/main/resources/application.yml`):

```
server:
  port: ${vcap.application.port:8761}   # HTTP port

eureka:
```

```
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

Similar to Eureka Server, each OTRS service should also contain the Eureka Client configuration, so that a connection between Eureka Server and the client can be established. Without this, the registration and discovery of services is not possible.

Eureka Client: your services can use the following spring configuration to configure Eureka Server:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Execution

To see how our code works, we need to first build it and then execute it. We'll use Maven *clean package* to build the service JARs.

Now to execute these service JARs, simply execute the following command from the service home directory:

```
    java -jar target/<service_jar_file>
```

For example:

```
  java -jar target/restaurant-service.jar
    java -jar target/eureka-service.jar
```

Testing

To enable testing, add the following dependency in `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

To test the `RestaurantController`, the following files have been added:

- `RestaurantControllerIntegrationTests`, which uses the `@SpringApplicationConfiguration` annotation to pick the same configuration that Spring Boot uses:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = RestaurantApp.class)
public class RestaurantControllerIntegrationTests extends
        AbstractRestaurantControllerTests {

}
```

- An abstract class to write our tests:

```
public abstract class AbstractRestaurantControllerTests {

    protected static final String RESTAURANT = "1";
    protected static final String RESTAURANT_NAME = "Big-O Restaurant";

    @Autowired
    RestaurantController restaurantController;

    @Test
    public void validResturantById() {
        Logger.getGlobal().info("Start validResturantById test");
        ResponseEntity<Entity> restaurant =
restaurantController.findById(RESTAURANT);

        Assert.assertEquals(HttpStatus.OK, restaurant.getStatusCode());
        Assert.assertTrue(restaurant.hasBody());
        Assert.assertNotNull(restaurant.getBody());
        Assert.assertEquals(RESTAURANT, restaurant.getBody().getId());
        Assert.assertEquals(RESTAURANT_NAME,
restaurant.getBody().getName());
        Logger.getGlobal().info("End validResturantById test");
    }

    @Test
    public void validResturantByName() {
        Logger.getGlobal().info("Start validResturantByName test");
        ResponseEntity<Collection<Restaurant>> restaurants =
restaurantController.findByName(RESTAURANT_NAME);
        Logger.getGlobal().info("In validAccount test");
```

```
        Assert.assertEquals(HttpStatus.OK, restaurants.getStatusCode());
        Assert.assertTrue(restaurants.hasBody());
        Assert.assertNotNull(restaurants.getBody());
        Assert.assertFalse(restaurants.getBody().isEmpty());
        Restaurant restaurant = (Restaurant) restaurants.getBody().toArray()
[0];
        Assert.assertEquals(RESTAURANT, restaurant.getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.getName());
        Logger.getGlobal().info("End validResturantByName test");
    }

    @Test
    public void validAdd() {
        Logger.getGlobal().info("Start validAdd test");
        RestaurantVO restaurant = new RestaurantVO();
        restaurant.setId("999");
        restaurant.setName("Test Restaurant");

        ResponseEntity<Restaurant> restaurants =
restaurantController.add(restaurant);
        Assert.assertEquals(HttpStatus.CREATED,
restaurants.getStatusCode());
        Logger.getGlobal().info("End validAdd test");
    }
}
```

- Finally, `RestaurantControllerTests`, which extends the previously created abstract class and also creates the `RestaurantService` and `RestaurantRepository` implementations:

```
public class RestaurantControllerTests extends
AbstractRestaurantControllerTests {

    protected static final Restaurant restaurantStaticInstance = new
Restaurant(RESTAURANT,
            RESTAURANT_NAME, null);

    protected static class TestRestaurantRepository implements
RestaurantRepository<Restaurant, String> {

        private Map<String, Restaurant> entities;

        public TestRestaurantRepository() {
            entities = new HashMap();
            Restaurant restaurant = new Restaurant("Big-O Restaurant", "1",
null);
            entities.put("1", restaurant);
            restaurant = new Restaurant("O Restaurant", "2", null);
            entities.put("2", restaurant);
        }

        @Override
        public boolean containsName(String name) {
            try {
                return this.findByName(name).size() > 0;
```

```
            } catch (Exception ex) {
                //Exception Handler
            }
            return false;
        }

        @Override
        public void add(Restaurant entity) {
            entities.put(entity.getId(), entity);
        }

        @Override
        public void remove(String id) {
            if (entities.containsKey(id)) {
                entities.remove(id);
            }
        }

        @Override
        public void update(Restaurant entity) {
            if (entities.containsKey(entity.getId())) {
                entities.put(entity.getId(), entity);
            }
        }

        @Override
        public Collection<Restaurant> findByName(String name) throws
Exception {
            Collection<Restaurant> restaurants = new ArrayList();
            int noOfChars = name.length();
            entities.forEach((k, v) -> {
                if (v.getName().toLowerCase().contains(name.subSequence(0,
noOfChars))) {
                    restaurants.add(v);
                }
            });
            return restaurants;
        }

        @Override
        public boolean contains(String id) {
            throw new UnsupportedOperationException("Not supported yet.");
//To change body of generated methods, choose Tools | Templates.
        }

        @Override
        public Entity get(String id) {
            return entities.get(id);
        }
        @Override
        public Collection<Restaurant> getAll() {
            return entities.values();
        }
    }
```

```
    protected TestRestaurantRepository testRestaurantRepository = new
TestRestaurantRepository();
    protected RestaurantService restaurantService = new
RestaurantServiceImpl(testRestaurantRepository);

    @Before
    public void setup() {
        restaurantController = new RestaurantController(restaurantService);

    }
}
```

References

- *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing: https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices
- *Spring Cloud*: http://cloud.spring.io/
- *Netflix Eureka*: https://github.com/netflix/eureka

Summary

In this chapter, we have learned how the domain-driven design model can be used in a µService. After running the demo application, we can see how each µService can be developed, deployed, and tested independently. You can create µServices using Spring Cloud very easily. We have also explored how one can use the Eureka registry and Discovery component with Spring Cloud.

In the next chapter, we will learn to deploy µServices in containers such as Docker. We will also understand µService testing using REST Java clients and other tools.

Chapter 5. Deployment and Testing

This chapter will explain how to deploy microservices in different forms, from standalone to containers such as Docker. It will also demonstrate how Docker can be used to deploy our sample project on a cloud service such as AWS. Before implementing Docker, first we'll explore other factors about microservices, such as load balancing and Edge Server. You will also come to understand microservice testing using different REST clients such as RestTemplate, Netflix Feign, and so on.

In this chapter, we will cover the following topics:

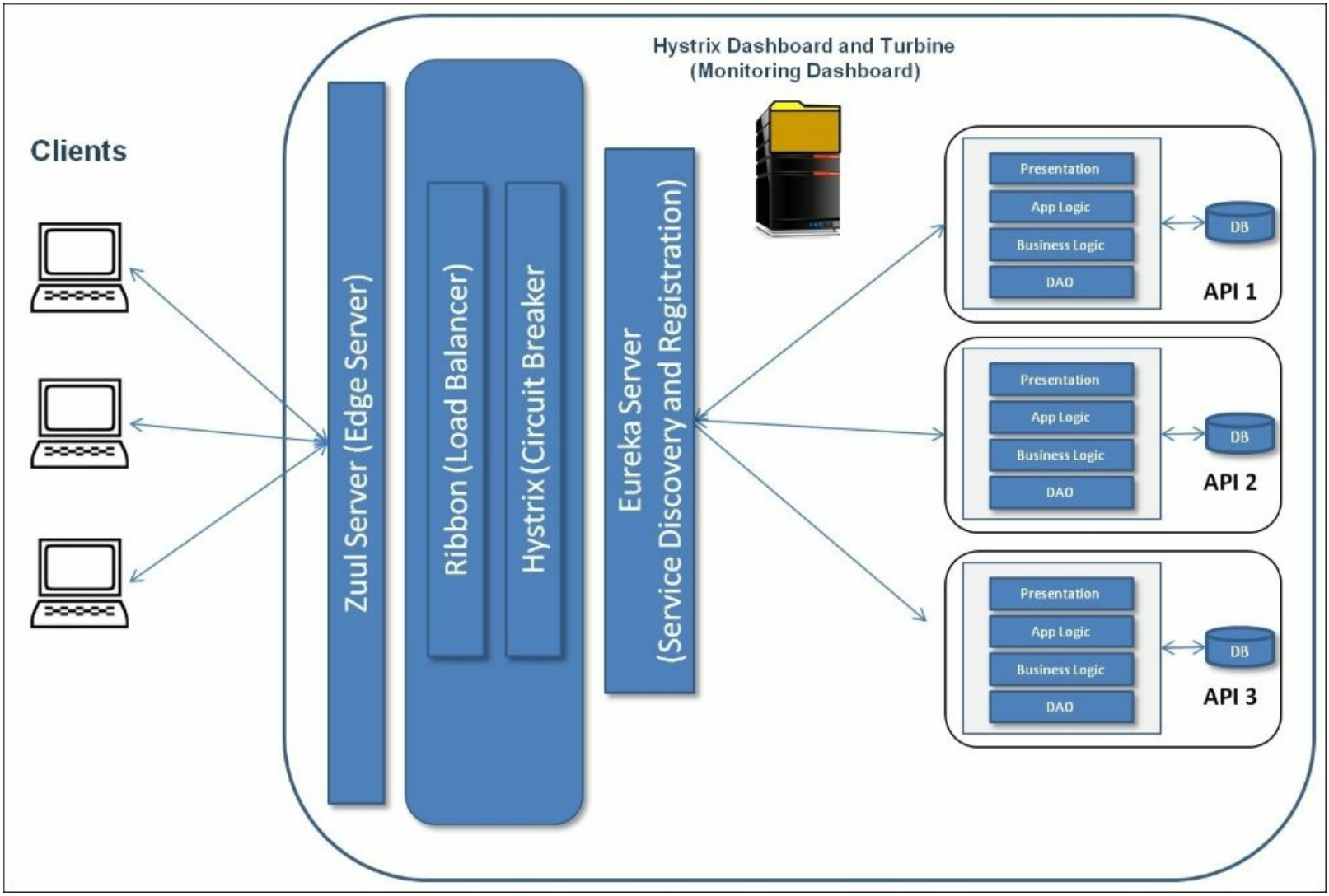
- An overview of microservice architecture using Netflix OSS
- Load balancing microservices
- Edge Server
- Circuit breakers and monitoring
- Microservice deployment using containers
- Microservice integration testing using Docker containers

An overview of microservice architecture using Netflix OSS

Netflix are pioneers in microservice architecture. They were the first to successfully implement microservice architecture on a large scale. They also helped increase its popularity and contributed immensely to microservices by open sourcing most of their microservice tools with **Netflix Open Source Software Center** (**OSS**).

According to the Netflix blog, when Netflix was developing their platform, they used Apache Cassandra for data storage, which is an open source tool from Apache. They started contributing to Cassandra with fixes and optimization extensions. This led to Netflix seeing the benefits of releasing Netflix projects with the name Open Source Software Center.

Spring took the opportunity to integrate many Netflix OSS projects, such as Zuul, Ribbon, Hystrix, Eureka Server, and Turbine, into Spring Cloud. This is one of the reasons Spring Cloud provides a ready-made platform for developing production-ready microservices. Now, let's take a look at a few important Netflix tools and how they fit into microservice architecture:



Microservice architecture diagram

As you can see in the preceding diagram, for each of the microservice practices, we have Netflix tool associated with it. We can go through the following mapping to understand it. Detailed information is covered in the respective sections of this chapter except concerning Eureka, which is elaborated on in the last chapter.

- **Edge Server**: We use Netflix Zuul Server as an Edge Server.
- **Load balancing**: Netflix Ribbon is used for load balancing.
- **Circuit breaker**: Netflix Hystrix is used as a circuit breaker and helps to keep the system up.
- **Service discovery and registration**: Netflix Eureka Server is used for service discovery and registration.
- **Monitoring dashboard**: Hystrix Dashboard is used with Netflix Turbine for microservice monitoring. It provides a dashboard to check the health of running microservices.

Load balancing

Load balancing is required to service requests in a manner that maximizes speed, capacity utilization, and it makes sure that no server is overloaded with requests. The load balancer also redirects requests to the remaining host servers if a server goes down. In microservice architecture, a microservice can serve internal or external requests. Based on this, we can have two types of load balancing – client-side and server-side load balancing.

Client-side load balancing

Microservices need interprocess communication so that services can communicate with each other. Spring Cloud uses Netflix Ribbon, a client-side load balancer that plays this critical role and can handle both HTTP and TCP. Ribbon is cloud-enabled and provides built-in failure resiliency. Ribbon also allows you to use multiple and pluggable load balancing rules. It integrates clients with load balancers.

In the last chapter, we added Eureka Server. Ribbon is integrated with Eureka Server in Spring Cloud by default. This integration provides the following features:

- You don't need to hardcode remote server URLs for discovery when Eureka Server is used. This is a prominent advantage, although you can still use the configured server list (*listOfServers*) in `application.yml` if required.
- The server list gets populated from Eureka Server. Eureka Server overrides `ribbonServerList` with `DiscoveryEnabledNIWSServerList`.
- The request to find out whether the server is up is delegated to Eureka. The `DiscoveryEnabledNIWSServerList` interface is used in place of Ribbon's `IPing`.

There are different clients available in Spring Cloud that use Ribbon, such as **RestTemplate** or **FeignClient**. These clients allow microservices to communicate with each other. Clients use instance IDs in place of hostnames and ports for making an HTTP call to service instances when Eureka Server is used. The client passes the service ID to Ribbon, Ribbon then uses the load balancer to pick the instance from the Eureka Server.

If there are multiple instances of services available in Eureka, as shown in the following screenshot, Ribbon picks only one for the request, based on load balancing algorithms:



Multiple service registration – Restaurant service

We can use `DiscoveryClient` to find all the available service instances in Eureka Server, as shown in the following code. Method `getLocalServiceInstance()` of class `DiscoveryClientSample` returns the all local service instances available in Eureka Server.

DiscoveryClient sample:

```
@Component
class DiscoveryClientSample implements CommandLineRunner {

    @Autowired
    private DiscoveryClient;

    @Override
    public void run(String... strings) throws Exception {
   //print the Discovery Client Description
        System.out.println(discoveryClient.description());
   // Get restaurant-service instances and prints its info
        discoveryClient.getInstances("restaurant-
service").forEach((ServiceInstance serviceInstance) -> {
            System.out.println(new StringBuilder("Instance -->
").append(serviceInstance.getServiceId())
                    .append("\nServer:
").append(serviceInstance.getHost()).append(":").append(serviceInstance.getPort(
))
                    .append("\nURI:
").append(serviceInstance.getUri()).append("\n\n\n"));
        });
    }
}
```

When executed, this code prints the following information. It shows two instances of the Restaurant service:

```
Spring Cloud Eureka Discovery Client
Instance: RESTAURANT-SERVICE
Server: SOUSHARM-IN:3402
URI: http://SOUSHARM-IN:3402
Instance --> RESTAURANT-SERVICE
Server: SOUSHARM-IN:3368
URI: http://SOUSHARM-IN:3368
```

The following samples showcase how these clients can be used. You can see that in both clients, the service name `restaurant-service` is used in place of a service hostname and port. These clients call `/v1/restaurants` to get a list of restaurants containing the name given in the name query parameter:

Rest Template sample:

```
@Override
public void run(String... strings) throws Exception {
ResponseEntity<Collection<Restaurant>> exchange
= this.restTemplate.exchange(
"http://restaurant-service/v1/restaurants?name=o",
                  HttpMethod.GET,
                  null,
                  new ParameterizedTypeReference<Collection<Restaurant>>() {
                 },
```

```
                    ( "restaurants");
exchange.getBody().forEach((Restaurant restaurant) -> {
System.out.println(new StringBuilder("\n\n\n[
").append(restaurant.getId()).append("
").append(restaurant.getName()).append("]"));
});
}
```

FeignClient sample:

```
@Component
class FeignSample implements CommandLineRunner {

    @Autowired
    private RestaurantClient restaurantClient;

    @Override
    public void run(String... strings) throws Exception {
        this.restaurantClient.getRestaurants("o").forEach((Restaurant
restaurant) -> {
            System.out.println(restaurant);
        });
    }
}

@FeignClient("restaurant-service")
interface RestaurantClient {

    @RequestMapping(method = RequestMethod.GET, value = "/v1/restaurants")
    Collection<Restaurant> getRestaurants(@RequestParam("name") String name);
}
```

All preceding examples will print the following output:

```
[ 1 Big-O Restaurant]
[ 2 O Restaurant]
```

Server-side load balancing

After client-side load balancing, it is important for us to define server-side load balancing. In addition, from the microservice architecture's point of view, it is important to define the routing mechanism for our OTRS app. For example, `/` may be mapped to our UI application, `/restaurantapi` is mapped to restaurant service, and `/userapi` is mapped to user service.

We'll use the Netflix Zuul Server as our Edge Server. Zuul is a JVM-based router and server-side load balancer. Zuul supports any JVM language for writing rules and filters and having the in-built support for Java and Groovy.

The external world (the UI and other clients) calls the Edge server, which uses the routes defined in `application.yml` to call internal services and provide the response. Your guess is right if you think it acts as a proxy server, carries gateway responsibility for internal networks, and calls internal services for defined and configured routes.

Normally, it is recommended to have a single Edge Server for all requests. However, few companies use a single Edge Server per client to scale. For example, Netflix uses a dedicated Edge Server for each device type.

An Edge Server will also be used in the next chapter, when we configure and implement microservice security.

Configuring and using the Edge Server is pretty simple in Spring Cloud. You need to use the following steps:

1. Define the Zuul Server dependency in `pom.xml`:

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

2. Use the `@EnableZuulProxy` annotation in your application class. It also internally uses `@EnableDiscoveryClient`: therefore it is also registered to Eureka Server automatically. You can find the registered Zuul Server in the last figure: *Multiple service registration – Restaurant service*".
3. Update the Zuul configuration in `application.yml`, as the following shows:
    - `zuul:ignoredServices`: This skips the automatic addition of services. We can define service ID patterns here. `*` denotes that we are ignoring all services. In the following sample, all services are ignored except `restaurant-service`.
    - `Zuul.routes`: This contains the `path` attribute that defines the URI's pattern. Here, `/restaurantapi` is mapped to Restaurant Service using `serviceId`. `serviceId` represents the service in Eureka Server. You can use a URL in place of a service, if Eureka Server is not used. We have also used the `stripPrefix` attribute to strip the prefix (`/restaurantapi`), and the resultant `/restaurantapi/v1/restaurants/1` call converts

to `/v1/restaurants/1` while calling the service:

application.yml

```
info:
    component: Zuul Server
# Spring properties
spring:
  application:
     name: zuul-server  # Service registers under this name

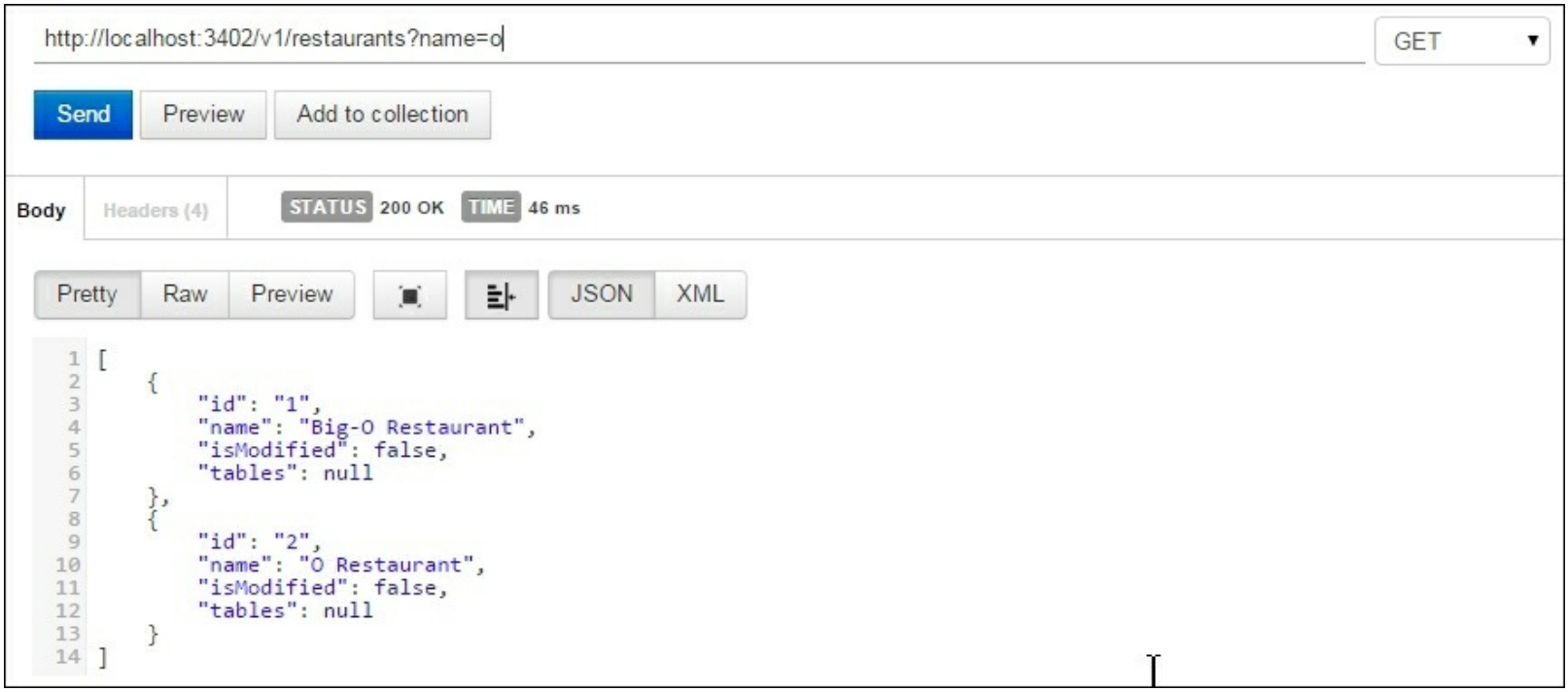
endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true
    health:
        sensitive: false

zuul:
    ignoredServices: "*"
    routes:
        restaurantapi:
            path: / restaurantapi/**
            serviceId: restaurant-service
            stripPrefix: true

server:
    port: 8765

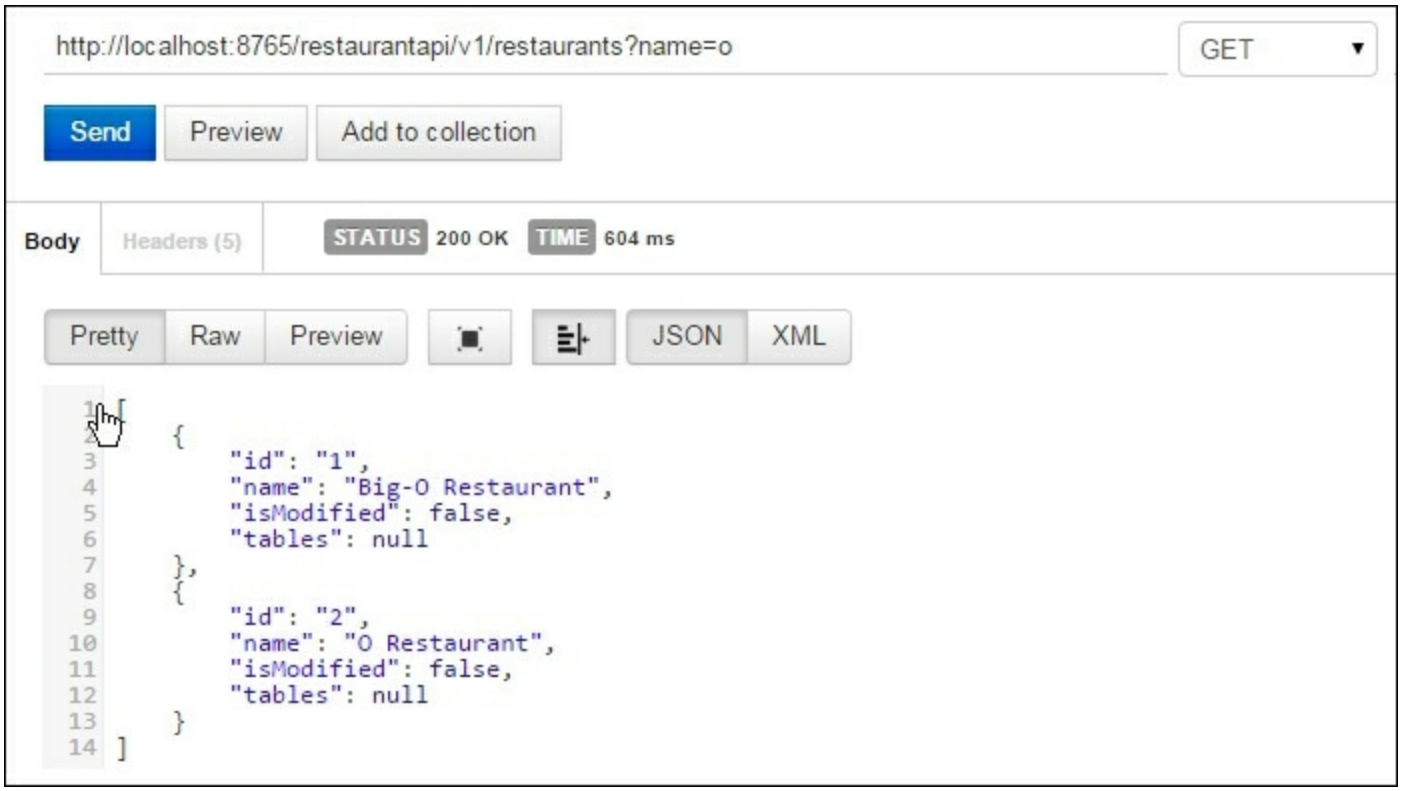
# Discovery Server Access
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 3
    metadataMap:
      instanceId:
${vcap.application.instance_id:${spring.application.name}:${spring.appli
cation.instance_id:${random.value}}}
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    fetchRegistry: false
```

Let's see a working Edge Server. First, we'll call the restaurant service deployed on port `3402`, shown as follows:



Direct Restaurant service call

Then, we'll call the same service using the Edge Server that is deployed on port `8765`. You can see that the `/restaurantapi` prefix is used for calling `/v1/restaurants?name=o`, and it gives the same result:



Restaurant Service call using Edge Server

Circuit breaker and monitoring

In general terms, a circuit breaker is:

An automatic device for stopping the flow of current in an electric circuit as a safety measure.

The same concept is used for microservice development, known as the Circuit Breaker design pattern. It tracks the availability of external services such as Eureka Server, API services such as `restaurant-service`, and so on, and prevents service consumers from performing any action on any service that is not available.

It is another important aspect of microservice architecture, a safety measure (failsafe mechanism) when the service does not respond to a call made by the service consumer – circuit breaker.

We'll use Netflix Hystrix as a circuit breaker. It calls the internal fallback method in the service consumer when failures occur (for example due to a communication error or timeout). It executes embedded within its consumer of service. In the next section, you will find the code that implements this feature.

Hystrix opens the circuit and fail-fast when the service fails to respond repeatedly, until the service is available again. You must be wondering, if Hystrix opens the circuit, then how does it know that the service is available? It exceptionally allows some requests to call the service.

Using Hystrix's fallback methods

There are three steps for implementing fallback methods:

1. **Enable the circuit breaker**: The main class of microservice that consumes other services should be annotated with `@EnableCircuitBreaker`. For example, if a user service would like to get the restaurant details, where a user has reserved the table:

```
@SpringBootApplication
@EnableCircuitBreaker
@ComponentScan({"com.packtpub.mmj.user.service", "com.packtpub.mmj.common"})
public class UsersApp {
```

2. **Configure the fallback method**: To configure the `fallbackMethod`, the `@HystrixCommand` annotation is used:

```
@HystrixCommand(fallbackMethod = "defaultRestaurant")
public ResponseEntity<Restaurant> getRestaurantById(int restaurantId) {

    LOG.debug("Get Restaurant By Id with Hystrix protection");

    URI uri = util.getServiceUrl("restaurant-service");

    String url = uri.toString() + "/v1/restaurants/" + restaurantId;
    LOG.debug("Get Restaurant By Id URL: {}", url);

    ResponseEntity<Restaurant> response = restTemplate.getForEntity(url,
Restaurant.class);
    LOG.debug("Get Restaurant By Id http-status: {}",
response.getStatusCode());
    LOG.debug("GET Restaurant body: {}", response.getBody());

    Restaurant restaurant = response.getBody();
    LOG.debug("Restaurant ID: {}", restaurant.getId());

    return serviceHelper.createOkResponse(restaurant);
}
```

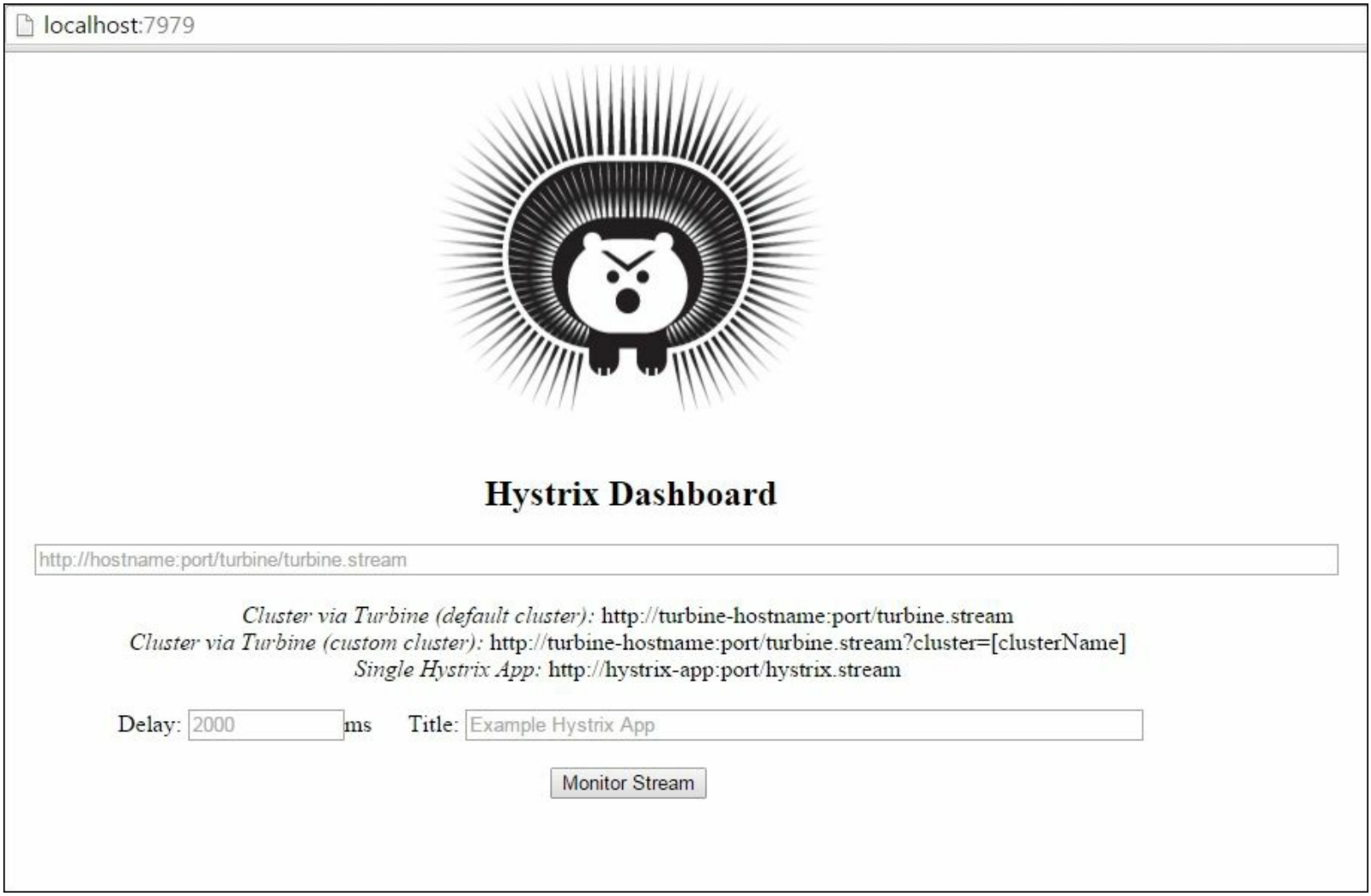
3. **Define fallback method**: A method that handles the failure and performs the steps for safety:

```
public ResponseEntity<Restaurant> defaultRestaurant(int restaurantId) {
    LOG.warn("Fallback method for restaurant-service is being used.");
    return serviceHelper.createResponse(null, HttpStatus.BAD_GATEWAY);
}
```

These steps should be enough to failsafe the service calls and return a more appropriate response to the service consumer.

Monitoring

Hystrix provides the dashboard with a web UI that provides nice graphics of circuit breakers:



Default Hystrix dashboard

Netflix Turbine is a web application that connects to the instances of your Hystrix applications in a cluster and aggregates information, which it does in real time (updated every 0.5 seconds). Turbine provides information using a stream that is known as a turbine stream.

If you combine Hystrix with Netflix Turbine, then you can get all the information from Eureka Server on the Hystrix dashboard. This gives you a landscape view of all the information about the circuit breakers.

To use Turbine with Hystrix, just type in the Turbine URL `http://localhost:8989/turbine.stream` (port `8989` is configured for the Turbine server in `application.yml`) in first textbox shown before, and click on **Monitory Stream**.

Netflix Hystrix and Turbine uses RabbitMQ, an open source message queuing software. RabbitMQ

works on **Advance Messaging Queue Protocol** (**AMQP**). It is a software in which queues can be defined, where applications can establish a connection and transfer a message through it. A message can include any kind of information. A message can be stored in the RabbitMQ queue until a receiver application connects and receives the message (taking the message off the queue).

Hystrix uses RabbitMQ to send a metrics data feed to Turbine.

Note

Before we configure Hystrix and Turbine, please install the RabbitMQ application on your platform. Hystrix and Turbine use RabbitMQ to communicate between themselves.

Setting up the Hystrix Dashboard

We'll add the new Maven dependency, `dashboard-server` for Hystrix Server. Configuring and using the Hystrix Dashboard is pretty simple in Spring Cloud like others. You just need to follow these steps:

1. Define the Hystrix Dashboard dependency in `pom.xml`:

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter- hystrix-dashboard</artifactId>
</dependency>
```

2. The `@EnableHystrixDashboard` annotation in the main Java class does everything for you to use it. We'll also use the `@Controller` to forward the request from the root to Hystrix, as shown here:

```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class DashboardApp extends SpringBootServletInitializer {

    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(DashboardApp.class).web(true);
    }

    public static void main(String[] args) {
        SpringApplication.run(DashboardApp.class, args);
    }
}
```

3. Update the Dashboard application configuration in `application.yml`, as shown here:

```
application.yml
# Hystrix Dashboard properties
spring:
    application:
        name: dashboard-server

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true

server:
```

```
    port: 7979

eureka:
    instance:
        leaseRenewalIntervalInSeconds: 3
        metadataMap:
            instanceId:
${vcap.application.instance_id:${spring.application.name}:${spring.applicati
on.instance_id:${random.value}}}

    client:
        # Default values comes from
org.springframework.cloud.netflix.eurek.EurekaClientConfigBean
        registryFetchIntervalSeconds: 5
        instanceInfoReplicationIntervalSeconds: 5
        initialInstanceInfoReplicationIntervalSeconds: 5
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/
        fetchRegistry: false

logging:
    level:
        ROOT: WARN
        org.springframework.web: WARN
```

Setting up Turbine

We'll create one more Maven dependency for Turbine. When you run the Hystrix Dashboard application, it will look like the *Default Hystrix Dashboard* screenshot shown earlier.

Now, we will configure the Turbine Server using the following steps:

1. Define the Turbine Server dependency in pom.xml:

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-turbine-amqp</artifactId>
</dependency>
```

2. Use the `@EnableTurbineAmqp` annotation in your application class as shown here. We are also defining a bean that will return the RabbitMQ Connection Factory:

```
@SpringBootApplication
@EnableTurbineAmqp
@EnableDiscoveryClient
public class TurbineApp {

    private static final Logger LOG =
LoggerFactory.getLogger(TurbineApp.class);

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMQHost;

    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Creating RabbitMQHost ConnectionFactory for host: {}",
rabbitMQHost);
        CachingConnectionFactory cachingConnectionFactory = new
CachingConnectionFactory(rabbitMQHost);
        return cachingConnectionFactory;
    }

    public static void main(String[] args) {
        SpringApplication.run(TurbineApp.class, args);
    }
}
```

3. Update the Turbine configuration in `application.yml`, as shown here:

`server:port`: The main port used by the the turbine HTTP

`management:port`: Port of turbine Actuator endpoints

```
application.yml
spring:
    application:
        name: turbine-server
```

```
server:
    port: 8989

management:
    port: 8990

PREFIX:

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true

eureka:
    instance:
        leaseRenewalIntervalInSeconds: 10
    client:
        registryFetchIntervalSeconds: 5
        instanceInfoReplicationIntervalSeconds: 5
        initialInstanceInfoReplicationIntervalSeconds: 5
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/

logging:
    level:
        root: WARN
        com.netflix.discovery: 'OFF'
```

Note

Please be aware the preceding steps always create the respective servers with default configurations. If required, you can override the default configuration with specific settings.

Microservice deployment using containers

You might have got the point about Docker after reading Chapter 1, *A Solution Approach*.

A Docker container provides a lightweight runtime environment, consisting of the core features of a virtual machine and the isolated services of operating systems, known as a docker image. Docker makes the packaging and execution of µServices easier and smoother. Each operating system can have multiple Dockers, and each Docker can run multiple applications.

Installation and configuration

Docker needs a virtualized server if you are not using a Linux OS. You can install VirtualBox or similar tools such as Docker Toolbox to make it work for you. The Docker installation page gives more details about it and lets you know how to do it. So, leave it to the Docker installation guide available on Docker's website.

You can install Docker, based on your platform, by following the instructions given at https://docs.docker.com/engine/installation/.

DockerToolbox-1.9.1f was the latest version available at the time of writing. This is the version we used.

Docker Machine with 4 GB

Default machines are created with 2 GB of memory. We'll recreate a Docker Machine with 4 GB of memory:

```
docker-machine rm default
docker-machine create -d virtualbox --virtualbox-memory 4096 default
```

Building Docker images with Maven

There are various Docker maven plugins that can be used:

- https://github.com/rhuss/docker-maven-plugin
- https://github.com/alexec/docker-maven-plugin
- https://github.com/spotify/docker-maven-plugin

You can use any of these, based on your choice. I found the Docker Maven plugin by `@rhuss` to be best suited for us. It is updated regularly and has many extra features when compared to the others.

We need to introduce the Docker Spring Profile in `application.yml` before we start discussing the configuration of `docker-maven-plugin`. It will make our job easier when building services for various platforms. We need to configure the following four properties:

- We'll use the Spring profile identified as Docker.
- There won't be any conflict of ports among embedded Tomcat, since services will be executed in their own respective containers. We can now use port `8080`.
- We will prefer to use an IP address to register our services in Eureka. Therefore, the Eureka instance property `preferIpAddress` will be set to `true`.
- Finally, we'll use the Eureka Server host name in `serviceUrl:defaultZone`.

To add a Spring profile in your project, add the following lines in `application.yml` after the existing content:

```
---
```

```
# For deployment in Docker containers
spring:
  profiles: docker

server:
  port: 8080

eureka:
  instance:
    preferIpAddress: true
  client:
    serviceUrl:
      defaultZone: http://eureka:8761/eureka/
```

We will also add the following code in `pom.xml` to activate the Spring profile Docker, while building a Docker container JAR. (This will create the JAR using the previously defined properties, for example `port:8080`.)

```
<profiles>
    <profile>
        <id>docker</id>
        <properties>
            <spring.profiles.active>docker</spring.profiles.active>
        </properties>
    </profile>
</profiles>
```

We just need to use Maven `docker` profile while building the service, shown as follows:

```
mvn -P docker clean package
```

The preceding command will generate the `service` JAR with Tomcat's 8080 port and will get registered on Eureka Server with the hostname `eureka`.

Now, let's configure `docker-maven-plugin` to build the image with our restaurant microservice. This plugin has to create a Dockerfile first. The Dockerfile is configured in two places – in `pom.xml` and `docker-assembly.xml`. We'll use the following plugin configuration in `pom.xml`:

```
<properties>
<!-- For Docker hub leave empty; use "localhost:5000/" for a local Docker
Registry -->
  <docker.registry.name>localhost:5000/</docker.registry.name>
  <docker.repository.name>${docker.registry.name}sourabhh /${project.artifactId}
</docker.repository.name>
</properties>
...
<plugin>
  <groupId>org.jolokia</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.13.7</version>
  <configuration>
    <images>
```

```
      <image>
<name>${docker.repository.name}:${project.version}</name>
        <alias>${project.artifactId}</alias>

        <build>
          <from>java:8-jre</from>
          <maintainer>sourabhh</maintainer>
          <assembly>
            <descriptor>docker-assembly.xml</descriptor>
          </assembly>
          <ports>
            <port>8080</port>
          </ports>
          <cmd>
            <shell>java -jar \
              /maven/${project.build.finalName}.jar server \
              /maven/docker-config.yml</shell>
          </cmd>
        </build>
        <run>
        <!-- To Do -->
        </run>
      </image>
    </images>
  </configuration>
</plugin>
```

Above the Docker Maven plugin configuration, create a Dockerfile that creates the JRE 8 (`java:8-jre`) -based image. This exposes ports 8080 and 8081.

Next, we'll configure `docker-assembly.xml`, which tells the plugin which files should be put into the container. It will be placed under `src/main/docker`:

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
  <id>${project.artifactId}</id>
  <files>
    <file>
      <source>{basedir}/target/${project.build.finalName}.jar</source>
      <outputDirectory>/</outputDirectory>
    </file>
    <file>
      <source>src/main/resources/docker-config.yml</source>
      <outputDirectory>/</outputDirectory>
    </file>
  </files>
</assembly>
```

Above assembly, add the `service` JAR and `docker-config.yml` in the generated Dockerfile. This Dockerfile is located under `target/docker/`. On opening this file, you will find the content to be similar to this:

```
FROM java:8-jre
MAINTAINER sourabhh
EXPOSE 8080
COPY maven /maven/
CMD java -jar \
  /maven/restaurant-service.jar server \
  /maven/docker-config.yml
```

The preceding file can be found at `restaurant-service\target\docker\sousharm\restaurant-service\PACKT-SNAPSHOT\build`. The `build` directory also contains the `maven` directory, which contains everything mentioned in `docker-assembly.xml`.

Lets' build the Docker Image:

```
mvn docker:build
```

Once this command completes, we can validate the image in the local repository using Docker Images, or by running the following command:

```
docker run -it -p 8080:8080 sourabhh/restaurant-service:PACKT-SNAPSHOT
```

Use `-it` to execute this command in the foreground, in place of `-d`.

Running Docker using Maven

To execute a Docker Image with Maven, we need to add the following configuration in the `pom.xml`. `<run>` block, to be put where we marked the *To Do* under the image block of `docker-maven-plugin` section in the `pom.xml` file:

```
<properties>
  <docker.host.address>localhost</docker.host.address>
  <docker.port>8080</docker.port>
</properties>
...
<run>
  <namingStrategy>alias</namingStrategy>
  <ports>
    <port>${docker.port}:8080</port>
  </ports>
  <volumes>
    <bind>
      <volume>${user.home}/logs:/logs</volume>
    </bind>
  </volumes>
  <wait>
    <url>http://${docker.host.address}:${docker.port}/v1/restaurants/1</url>
    <time>100000</time>
  </wait>
  <log>
    <prefix>${project.artifactId}</prefix>
    <color>cyan</color>
```

```
  </log>
</run>
```

Here, we have defined the parameters for running our Restaurant service container. We have mapped Docker container ports `8080` and `8081` to the host system's ports, which allows us to access the service. Similarly, we have also bound the containers' logs directory to the host systems' `<home>/logs` directory.

The Docker Maven plugin can detect if the container has finished starting up by polling the ping URL of the admin backend until it receives an answer.

Please note that Docker host is not *localhost* if you are using DockerToolbox or boot2docker on Windows or Mac OS X. You can check the Docker Image IP by executing `docker-machine ip default`. It is also shown while starting up.

The Docker container is ready to start. Use the following command to start it using Maven:

```
mvn docker:start .
```

Integration testing with Docker

Starting and stopping a Docker container can be done by binding the following executions to the `docker-maven-plugin` life cycle phase in `pom.xml`:

```
<execution>
  <id>start</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>build</goal>
    <goal>start</goal>
  </goals>
</execution>
<execution>
  <id>stop</id>
  <phase>post-integration-test</phase>
  <goals>
    <goal>stop</goal>
  </goals>
</execution>
```

We will now configure the failsafe plugin to perform integration testing with Docker. This allows us to execute the integration tests. We are passing the service URL in the `service.url` tag, so that our integration test can use it to perform integration testing.

We'll use the `DockerIntegrationTest` marker to mark our Docker integration tests. It is defined as follows:

```
package com.packtpub.mmj.restaurant.resources.docker;

public interface DockerIntegrationTest {
```

```
    // Marker for Docker integratino Tests
}
```

Look at the following integration plugin code. You can see that `DockerIntegrationTest` is configured for the inclusion of integration tests (failsafe plugin), whereas it is used for excluding in unit tests (Surefire plugin):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.18.1</version>
  <configuration>
    <phase>integration-test</phase>
    <includes>
      <include>**/*.java</include>
    </includes>

<groups>com.packtpub.mmj.restaurant.resources.docker.DockerIntegrationTest</grou
ps>
    <systemPropertyVariables>
      <service.url>http://${docker.host.address}:${docker.port}/</service.url>
    </systemPropertyVariables>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.18.1</version>
  <configuration>

<excludedGroups>com.packtpub.mmj.restaurant.resources.docker.DockerIntegrationTe
st</excludedGroups>
  </configuration>
</plugin>
```

A simple integration test looks like this:

```
@Category(DockerIntegrationTest.class)
public class RestaurantAppDockerIT {

    @Test
    public void testConnection() throws IOException {
        String baseUrl = System.getProperty("service.url");
        URL serviceUrl = new URL(baseUrl + "v1/restaurants/1");
        HttpURLConnection connection = (HttpURLConnection)
serviceUrl.openConnection();
        int responseCode = connection.getResponseCode();
```

```
        assertEquals(200, responseCode);
    }
}
```

You can use the following command to perform integration testing using Maven:

```
mvn integration-test
```

Pushing the image to a registry

Add the following tags under `docker-maven-plugin` to publish the Docker Image to Docker Hub:

```
<execution>
  <id>push-to-docker-registry</id>
  <phase>deploy</phase>
  <goals>
    <goal>push</goal>
  </goals>
</execution>
```

You can skip JAR publishing by using the following configuration for `maven-deploy-plugin`:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
```

Publishing a Docker image in Docker Hub also requires a username and password:

```
mvn -Ddocker.username=<username> -Ddocker.password=<password> deploy
```

You can also push a Docker image to your own Docker registry. To do this, add the `docker.registry.name` tag as shown in the following code. For example, if your Docker registry is available at `xyz.domain.com` on port `4994`, then define it by adding the following line of code:

```
<docker.registry.name>xyz.domain.com: 4994</docker.registry.name>
```

This does the job and we can not only deploy, but also test our Dockerized service.

Managing Docker containers

Each microservice will have its own Docker container. Therefore, we'll use the *Docker Compose* Docker container manager to manage our containers.

Docker Compose will help us to specify the number of containers and how these will be executed. We can specify the Docker Image, ports, and each container's links to other Docker containers.

We'll create a file called `docker-compose.yml` in our root project directory and add all the microservice containers to it. We'll first specify the Eureka Server as follows:

```
eureka:
  image: localhost:5000/sourabhh/eureka-server
  ports:
    - "8761:8761"
```

Here, image represents the published Docker image for Eureka Server and ports represents the mapping between the host being used for executing the Docker Image and the Docker host.

This will start Eureka Server and publish the specified ports for external access.

Now, our services can use these containers (dependent containers such as Eureka). Let's see how `restaurant-service` can be linked to dependent containers. It is simple; just use the `links` directive:

```
restaurant-service:
  image: localhost:5000/sourabhh/restaurant-service
  ports:
    - "8080:8080"
  links:
    - eureka
```

The preceding links declaration will update the `/etc/hosts` file in the `restaurant-service` container with one line per service that the `restaurant-service` depends on (let's assume the `security` container is also linked), for example:

```
192.168.0.22  security
192.168.0.31  eureka
```

Tip

If you don't have a `docker` local registry set up, then please do this first for issue-less or smoother execution.

Build the `docker` local registry by:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Then, perform `push` and `pull` commands for the local images:

```
docker push localhost:5000/sourabhh/restaurant-service:PACKT-SNAPSHOT
docker-compose pull
```

Finally, execute `docker-compose`:

```
docker-compose up -d
```

Once all the microservice containers (service and server) are configured, we can start all Docker

containers with a single command:

```
docker-compose up -d
```

This will start up all Docker containers configured in Docker Composer. The following command will list them:

```
docker-compose ps
Name                                              Command
                State           Ports
-------------------------------------------------------------
onlinetablereservation5_eureka_1         /bin/sh -c java -jar         ...
Up      0.0.0.0:8761->8761/tcp

onlinetablereservation5_restaurant-service_1  /bin/sh -c java -jar       ...
Up      0.0.0.0:8080->8080/tcp
```

You can also check docker image logs using the following command:

```
docker-compose logs
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.819  INFO 7 --- [pool-3-
thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_RESTAURANT-
SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce - Re-registering
apps/RESTAURANT-SERVICE
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.820  INFO 7 --- [pool-3-
thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_RESTAURANT-
SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce: registering service...
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.917  INFO 7 --- [pool-3-
thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_RESTAURANT-
SERVICE/172.17
```

References

The following links will give you more information:

- **Netflix Ribbon**: https://github.com/Netflix/ribbon
- **Netflix Zuul**: https://github.com/Netflix/zuul
- **RabbitMQ**: https://www.rabbitmq.com/download.html
- **Hystrix**: https://github.com/Netflix/Hystrix
- **Turbine**: https://github.com/Netflix/Turbine
- **Docker**: https://www.docker.com/

Summary

In this chapter, we have learned about various microservice management features: – load balancing, Edge Server (Gateway), circuit breakers, and monitoring. You should now know how to implement load balancing and routing after going through this chapter. We have also learned how Edge Server can be set up and configured. The failsafe mechanism is another important part that you have learned in this chapter. Deployment can be made simple by using Docker or any other container. Docker was demonstrated and integrated using Maven Build.

From a testing point of view, we performed the integration testing on the Docker image of the service. We also explored the way we can write clients such as RestTemplate and Netflix Feign.

In the next chapter, we will learn to secure the μServices with respect to authentication and authorization. We will also explore the other aspects of microservice securities.

Chapter 6. Securing Microservices

As you know, microservices are the components that we deploy in on-premises or cloud infrastructure. Microservices may offer APIs or web applications. Our sample application, OTRS, offers APIs. This chapter will focus on how to secure these APIs using Spring Security and Spring OAuth2. We'll also focus on OAuth 2.0 fundamentals. We'll use OAuth 2.0 to secure the OTRS APIs. For more understanding on securing REST APIs, you can refer to *RESTful Java Web Services Security*, *Packt Publishing* book. You can also refer to *Spring Security [Video]*, *Packt Publishing* video, for more information on Spring Security. We'll also learn about Cross Origin Request Site filters, and cross-site scripting blockers.

In this chapter, we will cover the following topics:

- Enabling Secure Socket Layer (SSL)
- Authentication and authorization
- OAuth 2.0

Enabling Secure Socket Layer

So far, we are using the **Hyper Text Transfer Protocol** (**HTTP**). HTTP transfers data in plain text, but data transfer over the Internet in plain text is not a good idea at all. It makes hackers' jobs easy and allows them to get your private information, such as your user ID, passwords, and credit card details easily using a packet sniffer.

We definitely don't want to compromise user data, so we will provide the most secure way to access our web application. Therefore, we need to encrypt the information that is exchanged between the end user and our application. We'll use **Secure Socket Layer** (**SSL**) or **Transport Security Layer** (**TSL**) to encrypt the data.

SSL is a protocol designed to provide security (encryption) for network communications. HTTP associates with SSL to provide the secure implementation of HTTP, known as **Hyper Text Transfer Protocol Secure**, or **Hyper Text Transfer Protocol over SSL** (**HTTPS**). HTTPS makes sure that the privacy and integrity of the exchanged data is protected. It also ensures the authenticity of websites visited. This security centers around the distribution of signed digital certificates between the server hosting the application, the end user's machine, and a third-party trust store server. Let's see how this process takes place:

1. The end user sends the request to the web application, for example http://twitter.com, using a web browser.
2. On receiving the request, the server redirects the browser to https://twitter.com using the HTTP code 302.
3. The end user's browser connects to https://twitter.com and, in response, the server provides the certificate containing the digital signature to the end user's browser.
4. The end user's browser receives this certificate and sends it to a trusted **Certificate Authority** (**CA**) for verification.
5. Once the certificate gets verified all the way to the root CA, an encrypted communication is established between the end user's browser and the application hosting server.



Secure HTTP communication

Tip

Although SSL ensures security in terms of encryption and web application authenticity, it does not safeguard against phishing and other attacks. Professional hackers can decrypt information sent using HTTPS.

Now, after going over the basics of SSL, let's implement it for our sample OTRS project. We don't need to implement SSL for all microservices. All microservices will be accessed using our proxy or edge server; Zuul-server by the external environment, except our new microservice, security-service, which we will introduce in this chapter for authentication and authorization.

First, we'll set up SSL in edge server. We need to have the keystore that is required for enabling SSL in embedded Tomcat. We'll use the self-signed certificate for demonstration. We'll use Java keytool to generate the keystore using the following command. You can use any other tool also:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext
```

```
san=dns:localhost -storepass password -validity 365 -keysize 2048
```

It asks for information such as name, address details, organization, and so on (see the following screenshot):

```
C:\dev\workspace\ms\online-table-reservation-6>keytool -genkey -keyalg RSA -alias selfsigned -keystor
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  org unit
What is the name of your organization?
  [Unknown]:  org
What is the name of your City or Locality?
  [Unknown]:  city
What is the name of your State or Province?
  [Unknown]:  state
What is the two-letter country code for this unit?
  [Unknown]:  CN
Is CN=localhost, OU=org unit, O=org, L=city, ST=state, C=CN correct?
  [no]:  yes

Enter key password for <selfsigned>
        (RETURN if same as keystore password):
Re-enter new password:

C:\dev\workspace\ms\online-table-reservation-6>
```

The keytool generates keys

Be aware of the following points to ensure the proper functioning of self-signed certificates:

- Use `-ext` to define **Subject Alternative Names** (**SAN**). You can also use IP (for example, `san=ip:190.19.0.11`). Earlier, use of the hostname of the machine, where application deployment takes place was being used as most **common name** (**CN**). It prevents the `java.security.cert.CertificateException` for `No name matching localhost found`.
- You can use a browser or OpenSSL to download the certificate. Add the newly generated certificate to the cacerts keystore located at `jre/lib/security/cacerts` inside active `JDK/JRE` home directory by using the `keytool -importcert` command. Note that `changeit` is the default password for the cacerts keystore. Run the following command:

  ```
  keytool -importcert -file path/to/.crt -alias <cert alias> -keystore
  <JRE/JAVA_HOME>/jre/lib/security/cacerts -storepass changeit
  ```

  ## Tip

  Self-signed certificates can be used only for development and testing purposes. The use of these certificates in a production environment does not provide the required security. Always use the certificates provided and signed by trusted signing authorities in production environments. Store your private keys safely.

Now, after putting the generated `keystore.jks` in the `src/main/resources` directory of the OTRS project, along with `application.yml`, we can update this information in EdgeServer

`application.yml` as follows:

```
server:
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    port: 8765
```

Rebuild the Zuul-server JAR to use the HTTPS.

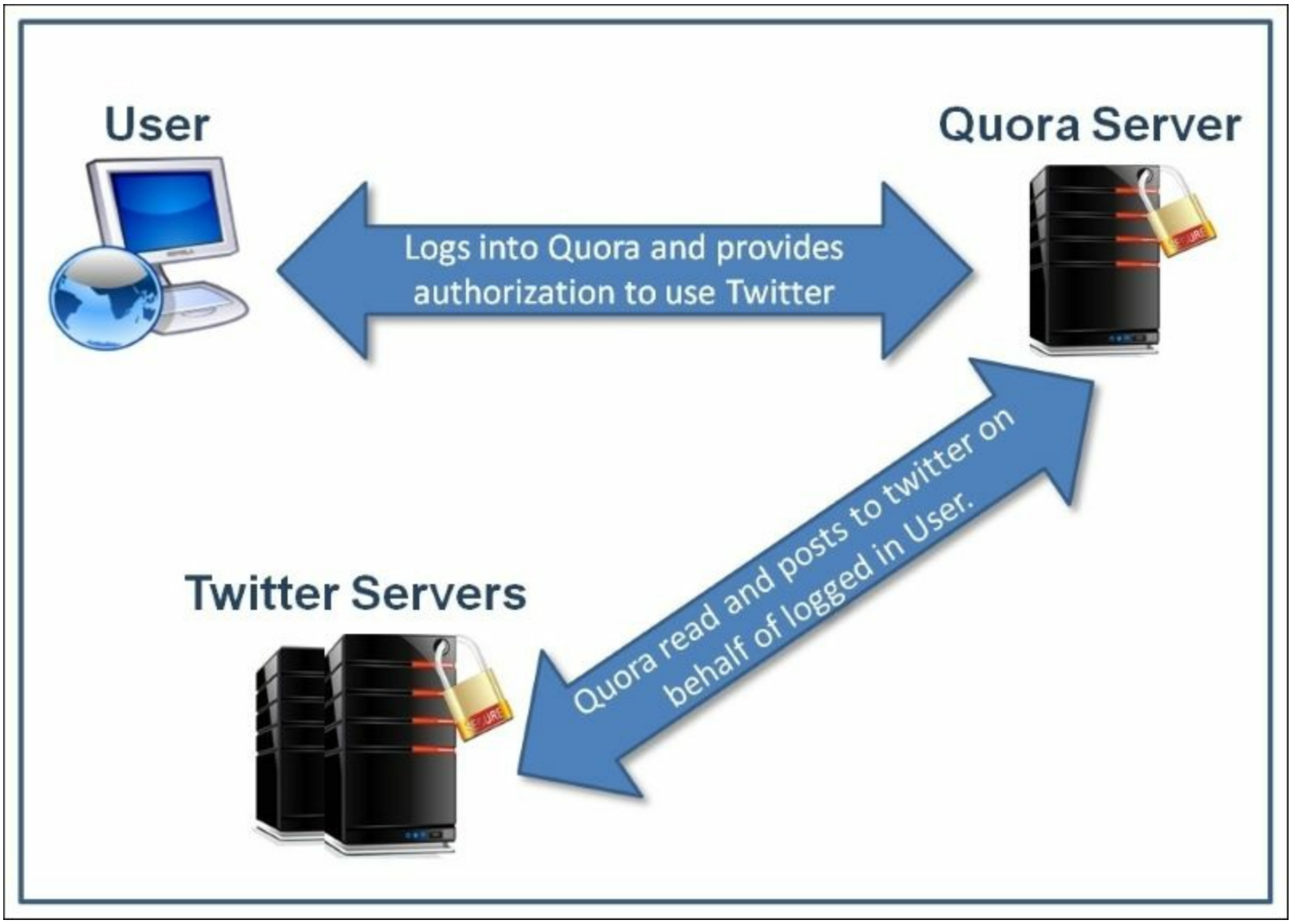
Tip

The key store file can be stored in the preceding class path in Tomcat version 7.0.66+ and 8.0.28+. For older versions, you can use the path of the key store file for the `server:ssl:key-store` value.

Similarly, you can configure SSL for other microservices.

Authentication and authorization

Providing authentication and authorization is de facto for web applications. We'll discuss authentication and authorization in this section. The new paradigm that has evolved over the past few years is OAuth. We'll learn and use OAuth 2.0 for implementation. OAuth is an open authorization mechanism, implemented in every major web application. Web applications can access each other's data by implementing the OAuth standard. It has become the most popular way to authenticate oneself for various web applications. Like on www.quora.com, you can register, and login using your Google or Twitter login IDs. It is also more user friendly, as client applications (for example, www.quora.com) don't need to store the user's passwords. The end user does not need to remember one more user ID and password.



OAuth 2.0 example usage

OAuth 2.0

The **Internet Engineering Task Force** (**IETF**) governs the standards and specifications of OAuth. OAuth 1.0a was the most recent version before OAuth 2.0 that was having a fix for session-fixation security flaw in the OAuth 1.0. OAuth 1.0 and 1.0a were very different from OAuth 2.0. OAuth 1.0 relies on security certificates and channel binding. OAuth 2.0 does not support security certification and channel binding. It works completely on **Transport Security Layer** (**TSL**). Therefore, OAuth 2.0 does not provide backward compatibility.

Usage of OAuth

- As discussed, it can be used for authentication. You might have seen it in various applications, displaying messages such as sign in using Facebook or sign in using Twitter.
- Applications can use it to read data from other applications, such as by integrating a Facebook widget into the application, or having a Twitter feed on your blog.
- Or, the opposite of the previous point can be true: you enable other applications to access the end user's data.

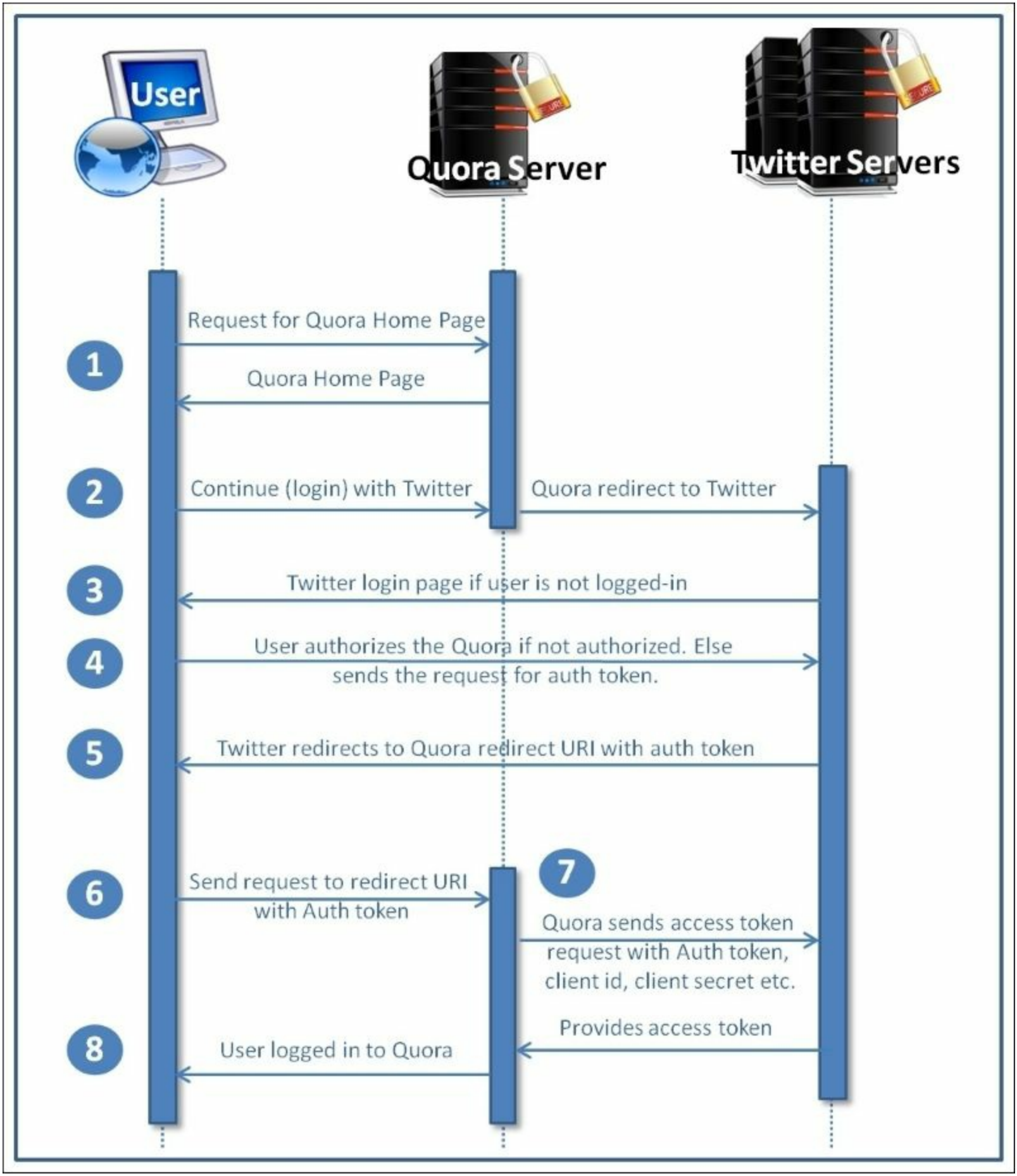
OAuth 2.0 specification – concise details

We'll try to discuss and understand the OAuth 2.0 specifications in a concise manner. Let's first see how signing in using Twitter works.

Please note that the process mentioned here was used at the time of writing. It may change in future. However, this process describes one of the OAuth 2.0 processes properly:

1. The user visits the Quora home page. It shows various login options. We'll explore the process of the **Continue with Twitter** link.
2. When the user clicks on the **Continue with Twitter** link, Quora opens a new window (in Chrome) that redirects the user to the www.twitter.com application. During this process few web applications redirect the user to the same opened tab/window.
3. In this new window/tab, the user signs in to www.twitter.com with their credentials.
4. If the user has not authorized the Quora application to use their data earlier, Twitter asks for the user's permission to authorize Quora to access the user's information. If the user has already authorized Quora, then this step is skipped.
5. After proper authentication, Twitter redirects the user to Quora's redirect URI with an authentication code.
6. Quora sends the client ID, client secret token, and authentication code (sent by Twitter in step 5) to Twitter when Quora redirect URI entered in the browser.
7. After validating these parameters, Twitter sends the access token to Quora.
8. The user is logged in to Quora on successful retrieval of the access token.
9. Quora may use this access token to retrieve user information from Quora.

You must be wondering how Twitter got Quora's redirect URI, client ID, and secret token. Quora works as a client application and Twitter as an authorization server. Quora, as a client, registered on Twitter by using Twitter's OAuth implementation to use resource owner (end user) information. Quora provides a redirect URI at the time of registration. Twitter provides the client ID and secret token to Quora. It works this way. In OAuth 2.0, user information is known as user resources. Twitter provides a resource server and an authorization server. We'll discuss more of these OAuth terms in the next sections.

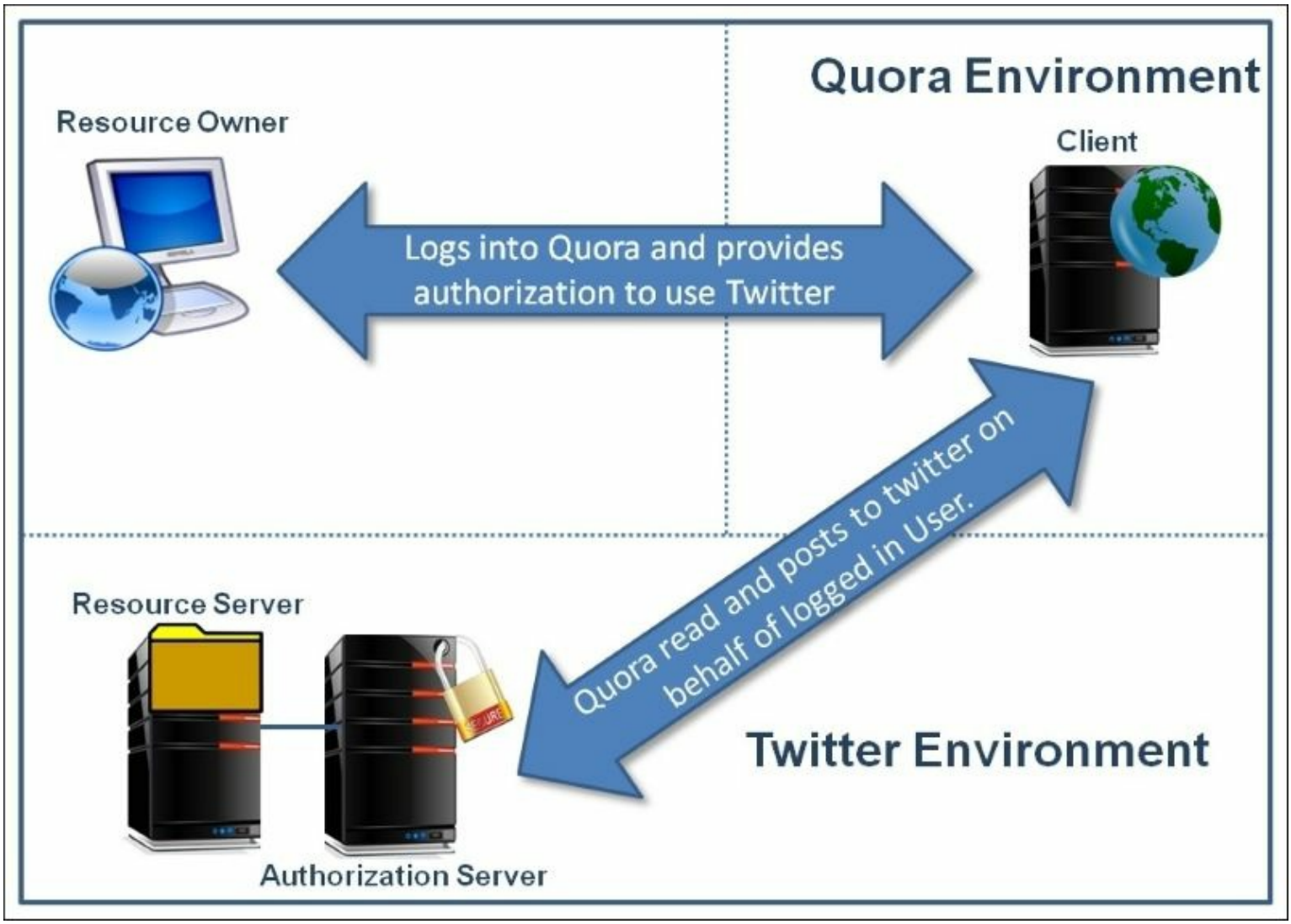


OAuth 2.0 example process for signing in with Twitter

OAuth 2.0 roles

There are four roles defined in the OAuth 2.0 specifications:

- Resource owner
- Resource server
- Client
- Authorization server



OAuth 2.0 roles

Resource owner

For the Quora sign in using Twitter example, the Twitter user was the resource owner. The resource owner is an entity that owns the protected resources (for example user handle, tweets and so on) that are to be shared. This entity can be an application or a person. We call this entity the resource owner because it can only grant access to its resources. Specification also defines, when resource owner is a person, it is referred to as an end user.

Resource server

The resource server hosts the protected resources. It should be capable of serving the access requests to these resources using access tokens. For the Quora sign in using Twitter example, Twitter is the resource server.

Client

For the Quora sign in using Twitter example, Quora is the client. The client is the application that makes access requests for protected resources to the resource server on behalf of the resource owner.

Authorization server

The authorization server provides different tokens to the client application, such as access tokens or refresh tokens, only after the resource owner authenticates themselves.

OAuth 2.0 does not provide any specifications for interactions between the resource server and the authorization server. Therefore, the authorization server and resource server can be on the same server, or can be on a separate one.

A single authorization server can also be used to issue access tokens for multiple resource servers.

OAuth 2.0 client registration

The client that communicates with the authorization server to obtain the access key for a resource should first be registered with the authorization server. The OAuth 2.0 specification does not specify the way a client registers with the authorization server. Registration does not require direct communication between the client and the authorization server. Registration can be done using self-issued or third-party-issued assertions. The authorization server obtains the required client properties using one of these assertions. Let's see what the client properties are:

- Client type (discussed in the next section).
- Client redirect URI, as we discussed in the Quora sign in using Twitter example. This is one of the endpoints used for OAuth 2.0. We will discuss other endpoints in the *Endpoints* section.
- Any other information required by the authorization server, for example client name, description, logo image, contact details, acceptance of legal terms and conditions, and so on.

Client types

There are two types of client described by the specification, based on their ability to maintain the confidentiality of client credentials: confidential and public. Client credentials are secret tokens issued by the authorization server to clients in order to communicate with them.

Confidential client type

This is a client application that keeps passwords and other credentials securely or maintains them confidentially. In the Quora sign in using Twitter example, the Quora app server is secure and has restricted access implementation. Therefore, it is of the confidential client type. Only the Quora app administrator has access to client credentials.

Public client type

These are client applications that do *not* keep passwords and other credentials securely or maintain them confidentially. Any native app on mobile or desktop, or an app that runs on browser, are perfect examples of the public client type, as these keep client credentials embedded inside them. Hackers can crack these apps and the client credentials can be revealed.

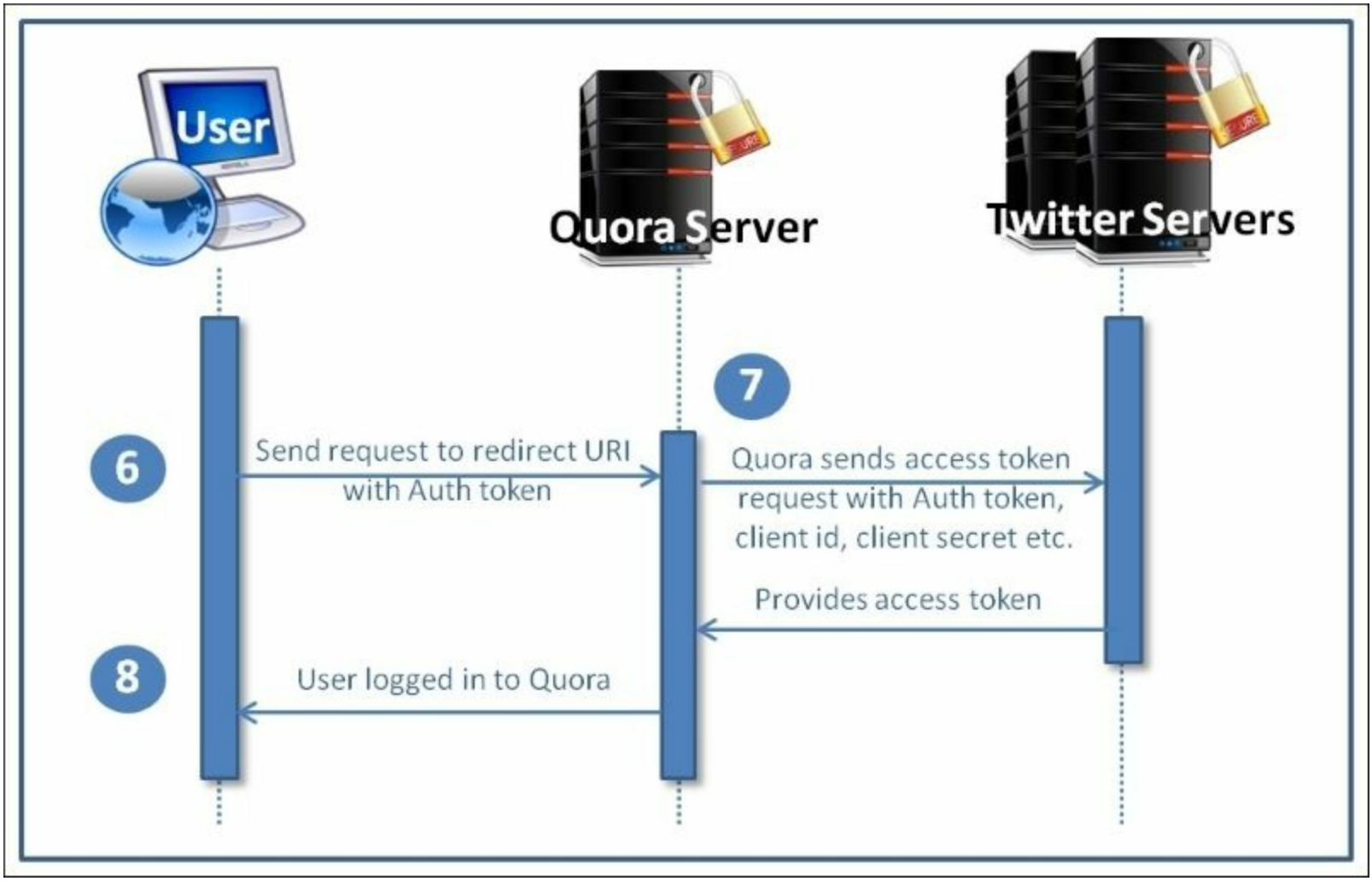
A client can be a distributed component-based application, for example, it could have both a web browser component and a server-side component. In this case, both components will have different client types and security contexts. Such a client should register each component as a separate client if the authorization server does not support such clients.

Based on the OAuth 2.0 client types, a client can have the following profiles:

- Web application
- User agent-based application
- Native application

Web application

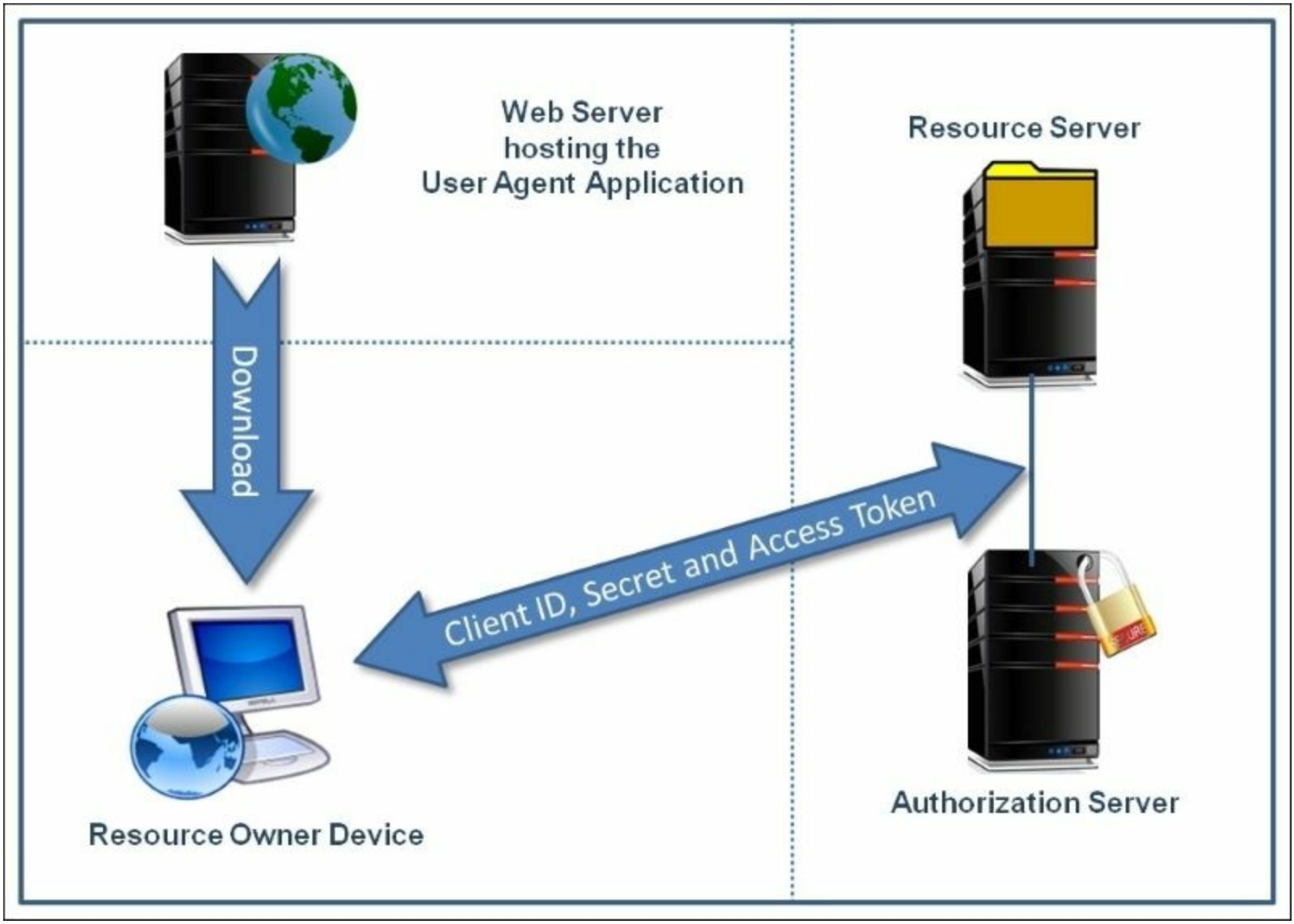
The Quora web application used in the Quora sign in using Twitter example is a perfect example of an OAuth 2.0 web application client profile. Quora is a confidential client running on a web server. The resource owner (end user) accesses the Quora application (OAuth 2.0 client) on the browser (user agent) using a HTML user interface on his device (desktop/tablet/cell phone). The resource owner cannot access the client (Quora OAuth 2.0 client) credentials and access tokens, as these are stored on the web server. You can see this behavior in the diagram of the OAuth 2.0 sample flow. See steps 6 to 8 in the following figure:



OAuth 2.0 client web application profile

User agent-based application

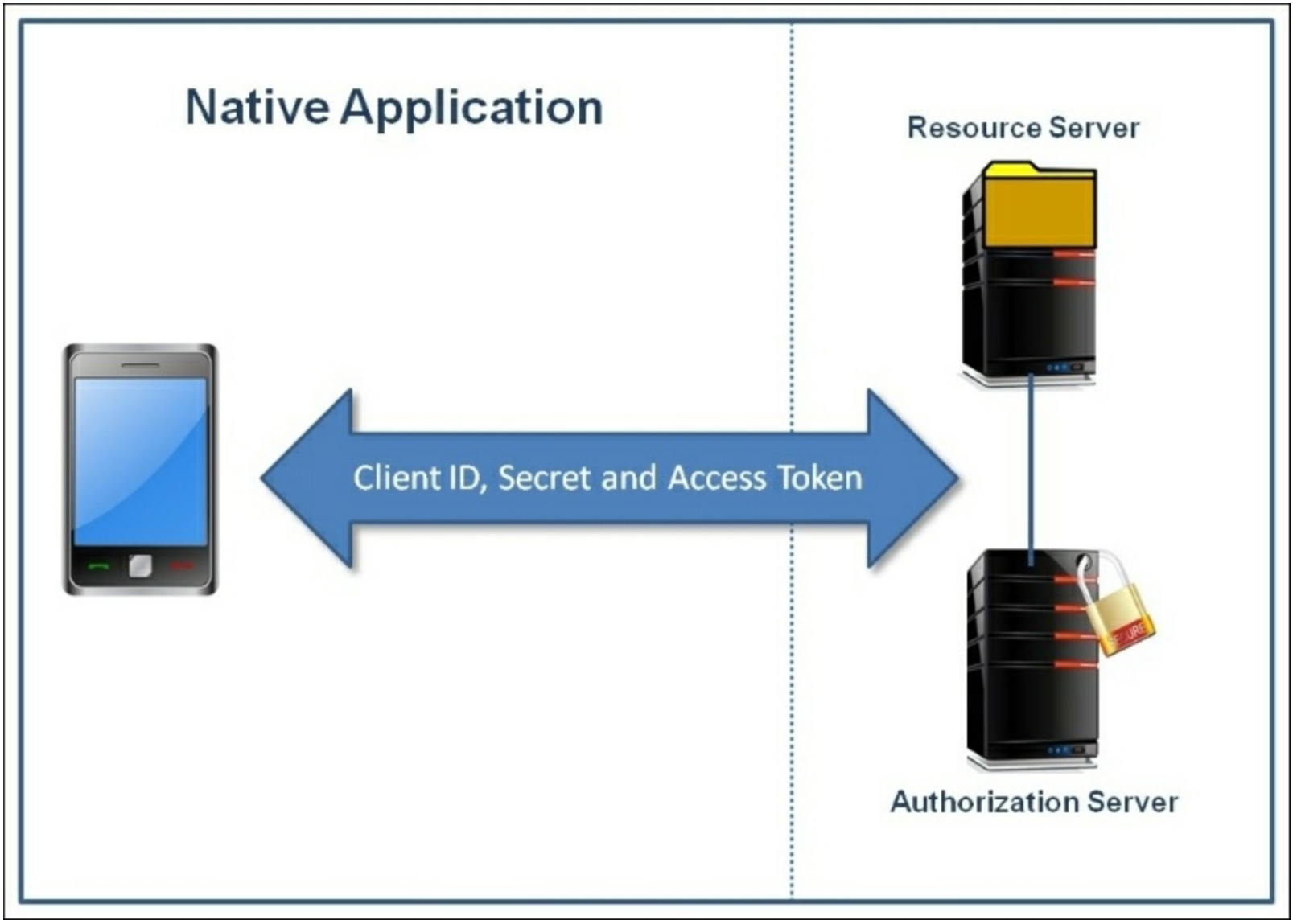
User agent-based applications are of the public client type. Here, though, the application resides in the web server, but the resource owner downloads it on the user agent (for example, a web browser) and then executes the application. Here, the downloaded application that resides in the user agent on the resource owner's device communicates with the authorization server. The resource owner can access the client credentials and access tokens. A gaming application is a good example of such an application profile.



OAuth 2.0 client user agent application profile

Native application

Native applications are similar to user agent-based applications, except these are installed on the resource owner's device and execute natively, instead of being downloaded from the web server, and then executes inside the user agent. Many native clients (mobile apps) you download on your mobile are of the native application type. Here, the platform makes sure that other application on the device do not access the credentials and access tokens of other applications. In addition, native applications should not share client credentials and OAuth tokens with servers that communicate with native applications.



OAuth 2.0 client native application profile

Client identifier

It is the authorization server's responsibility to provide a unique identifier to the registered client. This client identifier is a string representation of the information provided by the registered client. The authorization server needs to make sure that this identifier is unique. The authorization server should not use it on its own for authentication.

The OAuth 2.0 specification does not specify the size of the client identifier. The authorization server can set the size, and it should document the size of the client identifier it issues.

Client authentication

The authorization server should authenticate the client based on their client type. The authorization server should determine the authentication method that suits and meets security requirements. It should only use one authentication method in each request.

Typically, the authorization server uses a set of client credentials, such as the client password and some key tokens, to authenticate confidential clients.

The authorization server may establish a client authentication method with public clients. However, it must not rely on this authentication method to identify the client, for security reasons.

A client possessing a client password can use basic HTTP authentication. OAuth 2.0 does not recommend sending client credentials in the request body. It recommends using TLS and brute force attack protection on endpoints required for authentication.

OAuth 2.0 protocol endpoints

An endpoint is nothing but a URI we use for REST or web components such as Servlet or JSP. OAuth 2.0 defines three types of endpoint. Two are authorization server endpoints and one is a client endpoint:

- Authorization endpoint (authorization server endpoint)
- Token endpoint (authorization server endpoint)
- Redirection endpoint (client endpoint)

Authorization endpoint

This endpoint is responsible for verifying the identity of the resource owner and, once verified, obtaining the authorization grant. We'll discuss the authorization grant in the next section.

The authorization server require TLS for the authorization endpoint. The endpoint URI must not include the fragment component. The authorization endpoint must support the HTTP `GET` method.

The specification does not specify the following:

- The way the authorization server authenticates the client.
- How the client will receive the authorization endpoint URI. Normally, documentation contains the authorization endpoint URI, or the client obtains it at the time of registration.

Token endpoint

The client calls the token endpoint to receive the access token by sending the authorization grant or refresh token. The token endpoint is used by all authorization grants except an implicit grant.

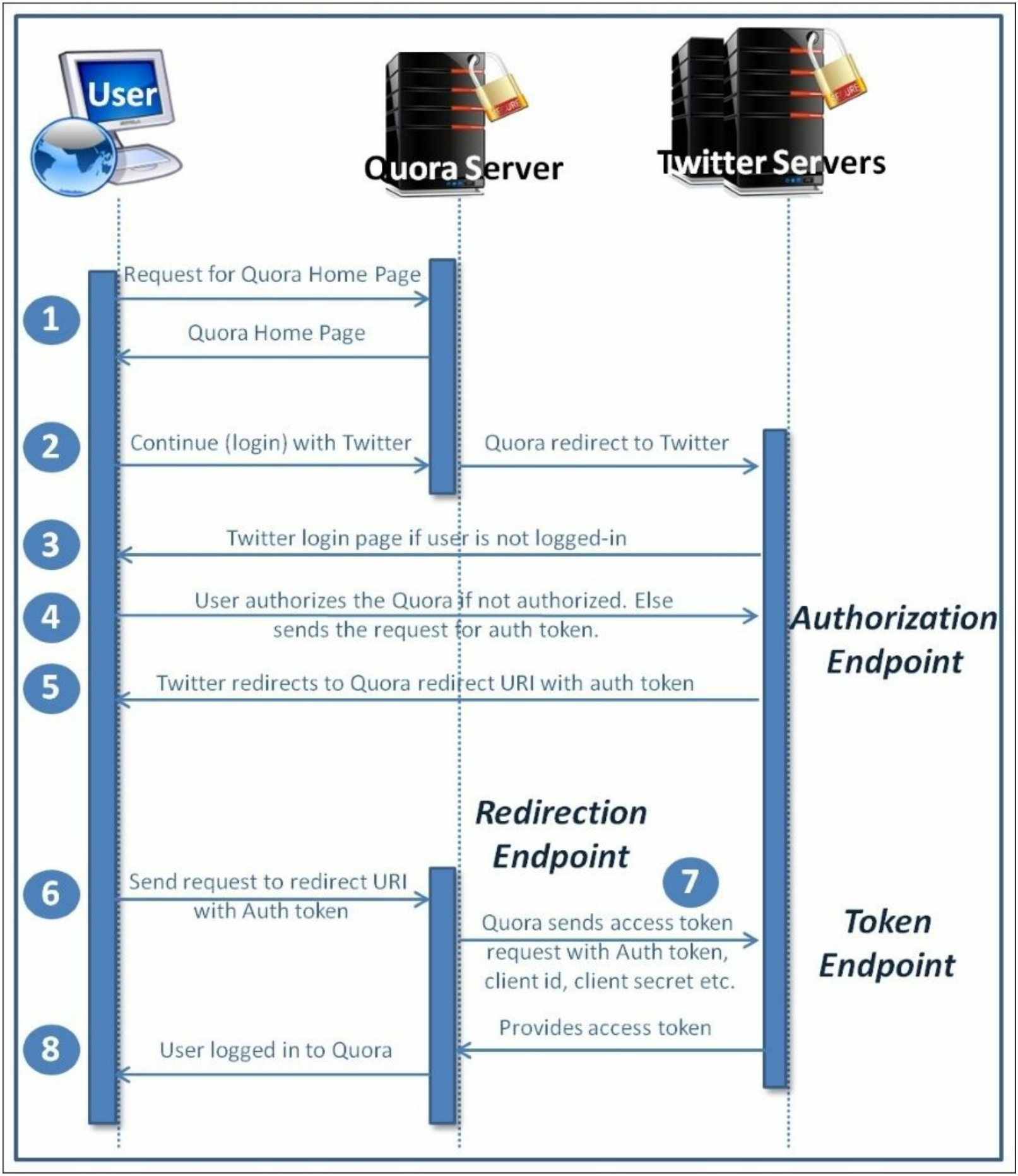
Like the authorization endpoint, the token endpoint also requires TLS. The client must use the HTTP `POST` method to make the request to the token endpoint.

Like the authorization endpoint, the specification does not specify how the client will receive the token endpoint URI.

Redirection endpoint

The authorization server redirects the resource owner's user agent (for example, a web browser) back

to the client using the redirection endpoint, once the authorization endpoint's interactions are completed between the resource owner and the authorization server. The client provides the redirection endpoint at the time of registration. The redirection endpoint must be an absolute URI and not contain a fragment component.



OAuth 2.0 endpoints

OAuth 2.0 grant types

The client requests an access token from the authorization server, based on the obtained authorization from the resource owner. The resource owner gives authorization in the form of an authorization grant. OAuth 2.0 defines four types of authorization grant:

- Authorization code grant
- Implicit grant
- Resource owner password credentials grant
- Client credentials grant

OAuth 2.0 also provides an extension mechanism to define additional grant types. You can explore this in the official OAuth 2.0 specifications.

Authorization code grant

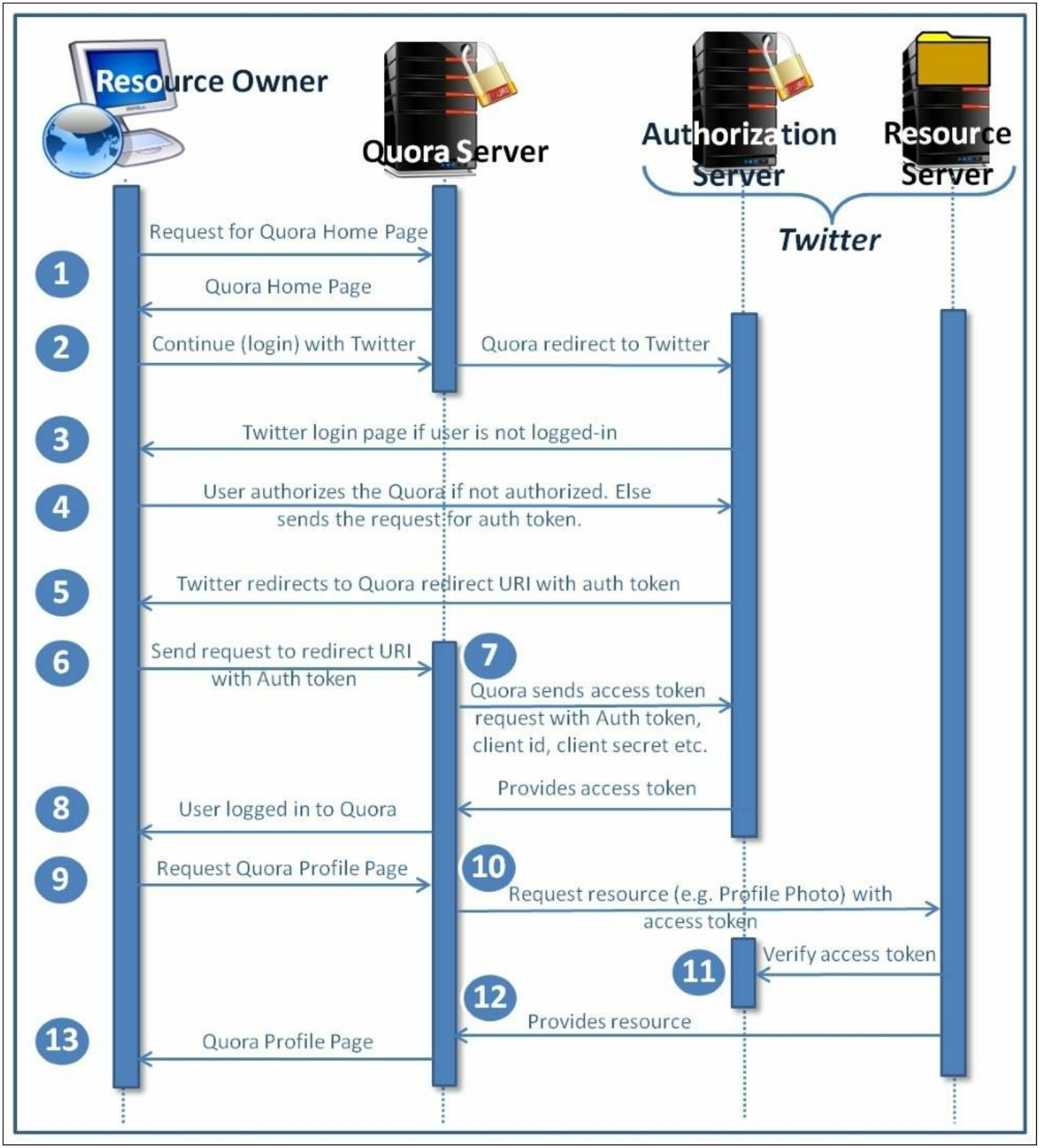
The first sample flow that we discussed in the OAuth 2.0 example flow for signing in with Twitter depicts an authorization code grant. We'll add a few more steps for the complete flow. As you know, after the eighth step, the end user logs in to the Quora application. Let's assume the user is logging in to Quora for the first time and requests their Quora profile page:

1. After logging in, the Quora user clicks on their Quora profile page.
2. The OAuth client Quora requests the Quora user's (resource owner) resources (for example, Twitter profile photo and so on) from the Twitter resource server and sends the access token received in the previous step.
3. The Twitter resource server verifies the access token using the Twitter authorization server.
4. After successful validation of the access token, the Twitter resource server provides the requested resources to Quora (OAuth client).
5. Quora uses these resources and displays the Quora profile page of the end user.

Authorization code requests and responses

If you looked at all the steps (a total of 13) of the authorization code flow, you can see that there are a total of two requests made by the client to the authorization server, and the authorization server in reply provides two responses: one request-response for the authentication token and one request-response for the access token.

Let's discuss the parameters used for each of these requests and responses.



OAuth 2.0 authorization code grant flow

The authorization request (step 4) to the authorization endpoint URI:

Parameter	Required / Optional	Description
response_type	Required	code (this value must be used).
client_id	Required	It represents the ID issued by the authorization server to the client at the time of registration.
redirect_uri	Optional	It represents the redirect URI given by the client at the time of registration.
scope	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.
state	Recommended	The client uses this parameter to maintain the client state between the requests and callback (from the authorization server). The specification recommends it to protect against cross site request forgery attacks.

Authorization response (step 5):

Parameter	Required / Optional	Description
code	Required	Code (authorization code) generated by the authorization server. Code should be expired after it is generated; the maximum recommended lifetime is 10 minutes. The client must not use the code more than once. If the client uses it more than once, then the request must be denied and all previous tokens issued based on the code should be revoked. Code is bound to the client ID and redirect URI.
state	Required	It represents the ID issued by the authorization server to the client at the time of registration.

Token request (step 7) to token endpoint URI:

Parameter	Required / Optional	Description
grant_type	Required	authorization_code (this value must be used).
code	Required	Code (authorization code) received from the authorization server.
redirect_uri	Required	Required if it was included in the authorization code request and the values should match.
client_id	Required	It represents the ID issued by the authorization server to the client at the time of registration.

Token response (step 8):

Parameter	Required / Optional	Description
`access_token`	Required	The access token issued by the authorization server.
`token_type`	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, bearer or mac.
`refresh_token`	Optional	This token can be used by the client to get a new access token using the same authorization grant.
`expires_in`	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.
`scope`	Optional/Required	Optional if identical to the scope requested by the client. Required if the access token scope is different from the one the client provided in their request to inform the client about the actual scope of the access token granted. If the client does not provide the scope while requesting the access token, then the authorization server should provide the default scope, or deny the request, indicating the invalid scope.

Error response:

Parameter	Required / Optional	Description
`error`	Required	One of the error codes defined in the specification, for example, `unauthorized_client`, `invalid_scope`.
`error_description`	Optional	Short description of the error.
`error_uri`	Optional	The URI of the error page describing the error.

An additional error parameter state is also sent in the error response if the state was passed in the client authorization request.

Implicit grant

The first sample flow that we discussed in the OAuth 2.0 example flow for signing in with Twitter depicts the authorization code grant. We'll add a few more steps for its complete flow. As you know after eighth steps, end user logs in to the Quora application. Let's assume user is logging in first time on Quora and requests for its Quora profile page:

1. *Step 9*: After login, the Quora user clicks on their Quora profile page.
2. *Step 10*: The OAuth client Quora requests the Quora user's (resource owner) resources (for example, Twitter profile photo and so on) from the Twitter resource server and sends the access token received in the previous step.
3. *Step 11*: The Twitter resource server verifies the access token using the Twitter authorization server.
4. *Step 12*: After successful validation of the access token, the Twitter resource server provides the requested resources to Quora (OAuth client).
5. *Step 13*: Quora uses these resources and displays the Quora profile page of the end user.

Implicit grant requests and responses

If you looked at all the steps (a total of 13) of the authorization code flow, you can see that there are total of two request made by the client to the authorization server, and the authorization server in reply provides two responses: one request-response for the authentication token and one request-response for the access token.

Let's discuss the parameters used for each of these requests and responses.

Authorization request to the authorization endpoint URI:

Parameter	Required / Optional	Description
`response_type`	Required	Token (this value must be used).
`client_id`	Required	It represents the ID issued by the authorization server to the client at the time of registration.
`redirect_uri`	Optional	It represents the redirect URI given by the client at the time of registration.
`scope`	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.
`state`	Recommended	The client uses this parameter to maintain the client state between the requests and the callback (from the authorization server). The specification recommends it to protect against cross site request forgery attacks.

Access token response:

Parameter	Required / Optional	Description
`access_token`	Required	The access token issued by the authorization server.

`token_type`	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, bearer or mac.
`refresh_token`	Optional	This token can be used by the client to get a new access token using the same authorization grant.
`expires_in`	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.
`scope`	Optional/Required	Optional if identical to the scope requested by the client. Required if the access token scope is different from the one the client provided in the request to inform the client about the actual scope of the access token granted. If the client does not provide the scope while requesting the access token, then the authorization server should provide the default scope, or deny the request, indicating the invalid scope.
`State`	Optional/Requried	Required if the state was passed in the client authorization request.

Error response:

Parameter	**Required / Optional**	**Description**
`error`	Required	One of the error codes defined in the specification, for example, `unauthorized_client`, `invalid_scope`.
`error_description`	Optional	Short description of the error.
`error_uri`	Optional	The URI of the error page describing the error.

An additional error parameter state is also sent in the error response if the state was passed in the client authorization request.

Resource owner password credentials grant

The first sample flow that we discussed in the OAuth 2.0 example flow for signing in with Twitter depicts the authorization code grant. We'll add a few more steps for its complete flow. As you know, after the eighth step, the end user logs in to the Quora application. Let's assume the user is logging in to Quora for the first time and requests their Quora profile page:

1. *Step 9*: After login, the Quora user clicks on their Quora profile page.
2. *Step 10*: The OAuth client Quora requests the Quora user's (resource owner) resources (for example, Twitter profile photo and so on) from the Twitter resource server and sends the access token received in the previous step.
3. *Step 11*: The Twitter resource server verifies the access token using the Twitter authorization

server.
4. *Step 12*: After successful validation of the access token, the Twitter resource server provides the requested resources to Quora (OAuth client).
5. *Step 13*: Quora uses these resources and displays the Quora profile page of the end user.

Resource owner password credentials grant requests and responses.

As seen in the previous section, in all the steps (a total of 13) of the authorization code flow, you can see that there are total of two requests made by the client to the authorization server, and the authorization server in reply provides two responses: one request-response for the authentication token and one request-response for the access token.

Let's discuss the parameters used for each of these requests and responses.

Access token request to the token endpoint URI:

Parameter	Required / Optional	Description
`grant_type`	Required	Password (this value must be used).
`username`	Required	Username of the resource owner.
`password`	Required	Password of the resource owner.
`scope`	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.

Access token response (step 8):

Parameter	Required / Optional	Description
`access_token`	Required	The access token issued by the authorization server.
`token_type`	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, bearer or mac.
`refresh_token`	Optional	This token can be used by the client to get a new access token using the same authorization grant.
`expires_in`	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.

Optional parameter	Optional	Additional parameter.

Client credentials grant

The first sample flow that we discussed in the OAuth 2.0 example flow for signing in with Twitter depicts the authorization code grant. We'll add a few more steps for its complete flow. As you know, after the eighth step, the end user logs in to the Quora application. Let's assume the user is logging in to Quora for the first time and requests their Quora profile page:

1. *Step 9*: After login, the Quora user clicks on their Quora profile page.
2. *Step 10*: The OAuth client Quora requests the Quora user's (resource owner) resources (for example, Twitter profile photo and so on) from the Twitter resource server and sends the access token received in the previous step.
3. *Step 11*: The Twitter resource server verifies the access token using the Twitter authorization server.
4. *Step 12*: After successful validation of the access token, the Twitter resource server provides the requested resources to Quora (OAuth client).
5. *Step 13*: Quora uses these resources and displays the Quora profile page of the end user.

Client credentials grant requests and responses.

If you looked at all the steps (a total of 13) of the authorization code flow, you can see that there are total of two requests made by the client to the authorization server, and the authorization server in reply provides two responses: one request-response for the authentication token and one request-response for the access token.

Let's discuss the parameters used for each of these requests and responses.

Access token request to the token endpoint URI:

Parameter	Required / Optional	Description
`grant_type`	Required	`client_credentials` (this value must be used).
`scope`	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.

Access token response:

Parameter	Required / Optional	Description

`access_token`	Required	The access token issued by the authorization server.
`token_type`	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, bearer or mac.
`expires_in`	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.

OAuth implementation using Spring Security

OAuth 2.0 is a way of securing APIs. Spring Security provides Spring Cloud Security and Spring Cloud OAuth2 components for implementing the rant flows we discussed above.

We'll create one more service, security-service, which will control authentication and authorization.

Create a new Maven project and follow these steps:

1. Add the Spring Security and Spring Security OAuth2 dependencies in `pom.xml`:

```
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

2. Use the `@EnableResourceServer` annotation in your application class. This will allow this application to work as a resource server. `@EnableAuthorizationServer` is another annotation we will use to enable the authorization server as per OAuth 2.0 specifications:

```
@SpringBootApplication
@RestController
@EnableResourceServer
public class SecurityApp {

    @RequestMapping("/user")
    public Principal user(Principal user) {
        return user;
    }

    public static void main(String[] args) {
        SpringApplication.run(SecurityApp.class, args);
    }

    @Configuration
    @EnableAuthorizationServer
    protected static class OAuth2Config extends
AuthorizationServerConfigurerAdapter {

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer
endpointsConfigurer) throws Exception {

endpointsConfigurer.authenticationManager(authenticationManager);
        }
```

```
        @Override
        public void configure(ClientDetailsServiceConfigurer
clientDetailsServiceConfigurer) throws Exception {
  // Using hardcoded inmemory mechanism because it is just an example
            clientDetailsServiceConfigurer.inMemory()
             .withClient("acme")
             .secret("acmesecret")
             .authorizedGrantTypes("authorization_code", "refresh_token",
"implicit", "password", "client_credentials")
             .scopes("webshop");
        }
    }
}
```

3. Update the security-service configuration in `application.yml`, as shown in the following code:
   - `server.contextPath`: It denotes the context path.
   - `security.user.password`: We'll use the hardcoded password for this demonstration. You can re-configure it for real use:

```
application.yml
info:
    component:
        Security Server

server:
    port: 9001
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    contextPath: /auth

security:
    user:
        password: password

logging:
    level:
        org.springframework.security: DEBUG
```

Now we have our security server in place, we'll expose our APIs using the new microservice `api-service`, which will be used for communicating with external applications and UIs.

Create a new Maven project and follow these steps:

1. Add the Spring Security and Spring Security OAuth2 dependencies in `pom.xml`:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
  <groupId>com.packtpub.mmj</groupId>
  <artifactId>online-table-reservation-common</artifactId>
  <version>PACKT-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <!-- Testing starter -->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

2. Use the `@EnableResourceServer` annotation in your application class. This will allow this application to work as a resource server:

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
@EnableResourceServer
```

```
@ComponentScan({"com.packtpub.mmj.api.service", "com.packtpub.mmj.common"})
public class ApiApp {

    private static final Logger LOG = LoggerFactory.getLogger(ApiApp.class);

    static {
        // for localhost testing only
        LOG.warn("Will now disable hostname check in SSL, only to be used
during development");
        HttpsURLConnection.setDefaultHostnameVerifier((hostname, sslSession)
-> true);
    }

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMqHost;

    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Create RabbitMqCF for host: {}", rabbitMqHost);
        CachingConnectionFactory connectionFactory = new
CachingConnectionFactory(rabbitMqHost);
        return connectionFactory;
    }

    public static void main(String[] args) {
        LOG.info("Register MDCHystrixConcurrencyStrategy");
        HystrixPlugins.getInstance().registerConcurrencyStrategy(new
MDCHystrixConcurrencyStrategy());
        SpringApplication.run(ApiApp.class, args);
    }
}
```

3. Update the `api-service` configuration in `application.yml`, as shown in the following code:
   - `security.oauth2.resource.userInfoUri`: It denotes the security service user URI.

```
application.yml
info:
  component: API Service

spring:
  application:
      name: api-service
  aop:
      proxyTargetClass: true

server:
  port: 7771

security:
  oauth2:
    resource:
     userInfoUri: https://localhost:9001/auth/user

management:
  security:
```

```
    enabled: false
## Other properties like Eureka, Logging and so on
```

Now we have our security server in place, we'll expose our APIs using the new microservice `api-service`, which will be used for communicating with external applications and UIs.

Now let's test and explore how it works for different OAuth 2.0 grant types.

Note

We'll make use of the `postman` extension to the Chrome browser to test the different flows.

Authorization code grant

We will enter the following URL in our browser. A request for authorization code is as follows:

```
https://localhost:9001/auth/oauth/authorize?
response_type=code&client_id=client&redirect_uri=http://localhost:7771/1&scope=a
```

Here, we provide the client ID (hardcoded client is by default we have registered in our security service), redirect URI, scope (hardcoded value `apiAccess` in security service) and state. You must be wondering about the `state` parameter. It contains the random number that we re-validate in response to prevent cross site request forgery.

If the resource owner (user) is not already authenticated, it will ask for the user name and password. Provide user as the `username` and password as the `password`; we have hardcoded these values in security service.

Once the login is successful, it will ask to provide your (resource owner) approval:

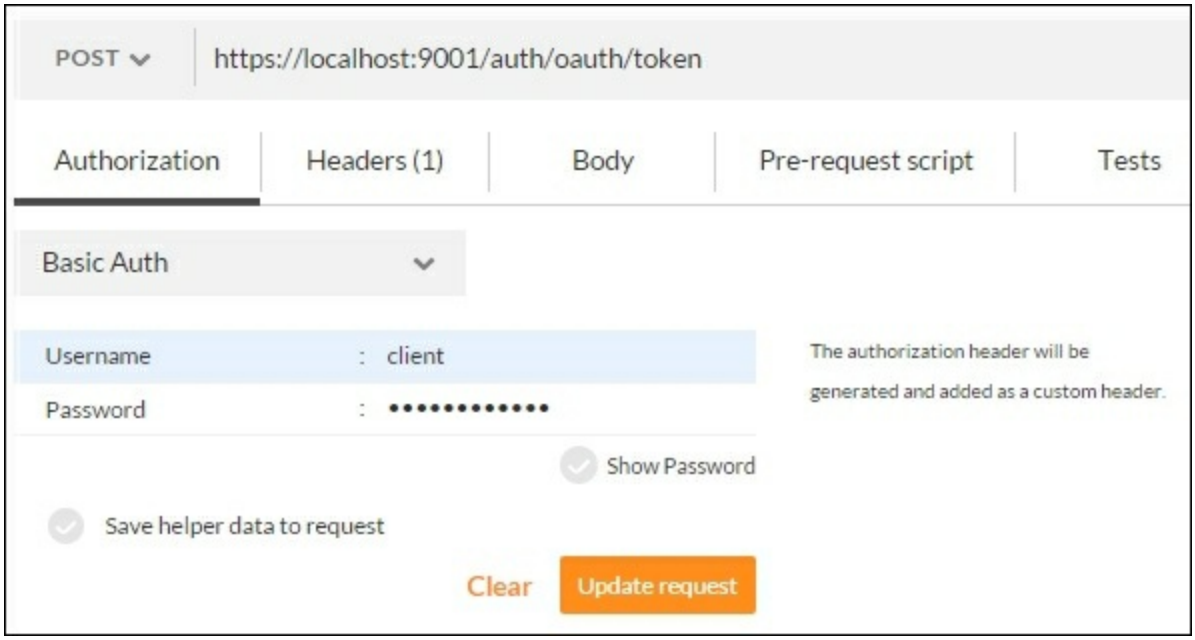


OAuth 2.0 authorization code grant – resource grant approval

Select **Approve** and click on **Authorize**. This action will redirect the application to `http://localhost:7771/1?code=o8t4fi&state=1234`.

As you can see, it has returned the authorization code and state.

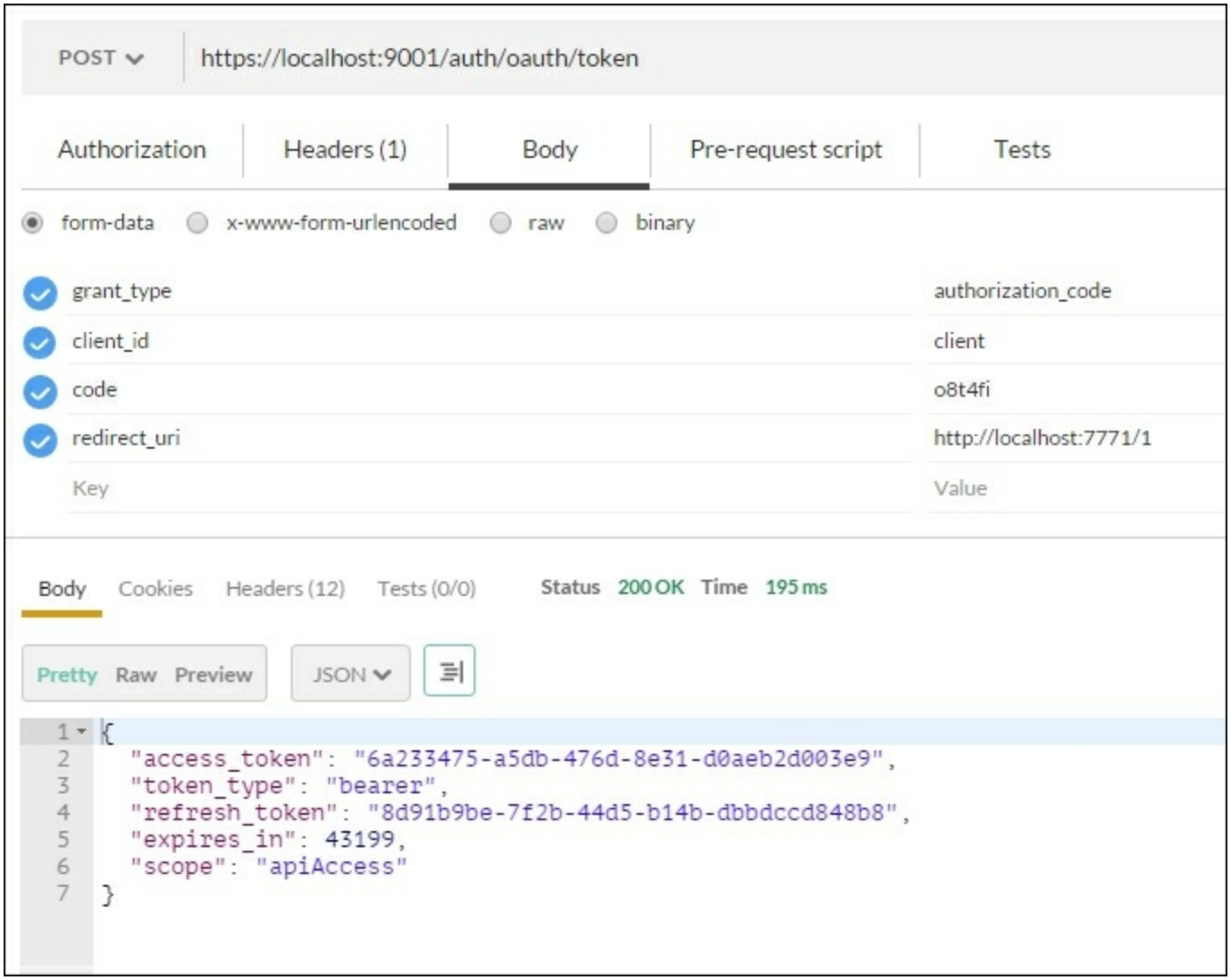
Now, we'll use this code to retrieve the access code. We'll use the postman Chrome extension. First we'll add the authorization header using **Username** as `client` and **Password** as `clientsecret`, as shown in the following screenshot:



OAuth 2.0 authorization code grant – access token request – adding the authentication

This will add the **Authorization** header to the request with the value `Basic Y2xpZW50OmNsaWVudHNlY3JldA==`.

Now, we'll add a few other parameters to the request, as shown in the following screenshot, and then submit the request:



OAuth 2.0 authorization code grant – access token request and response

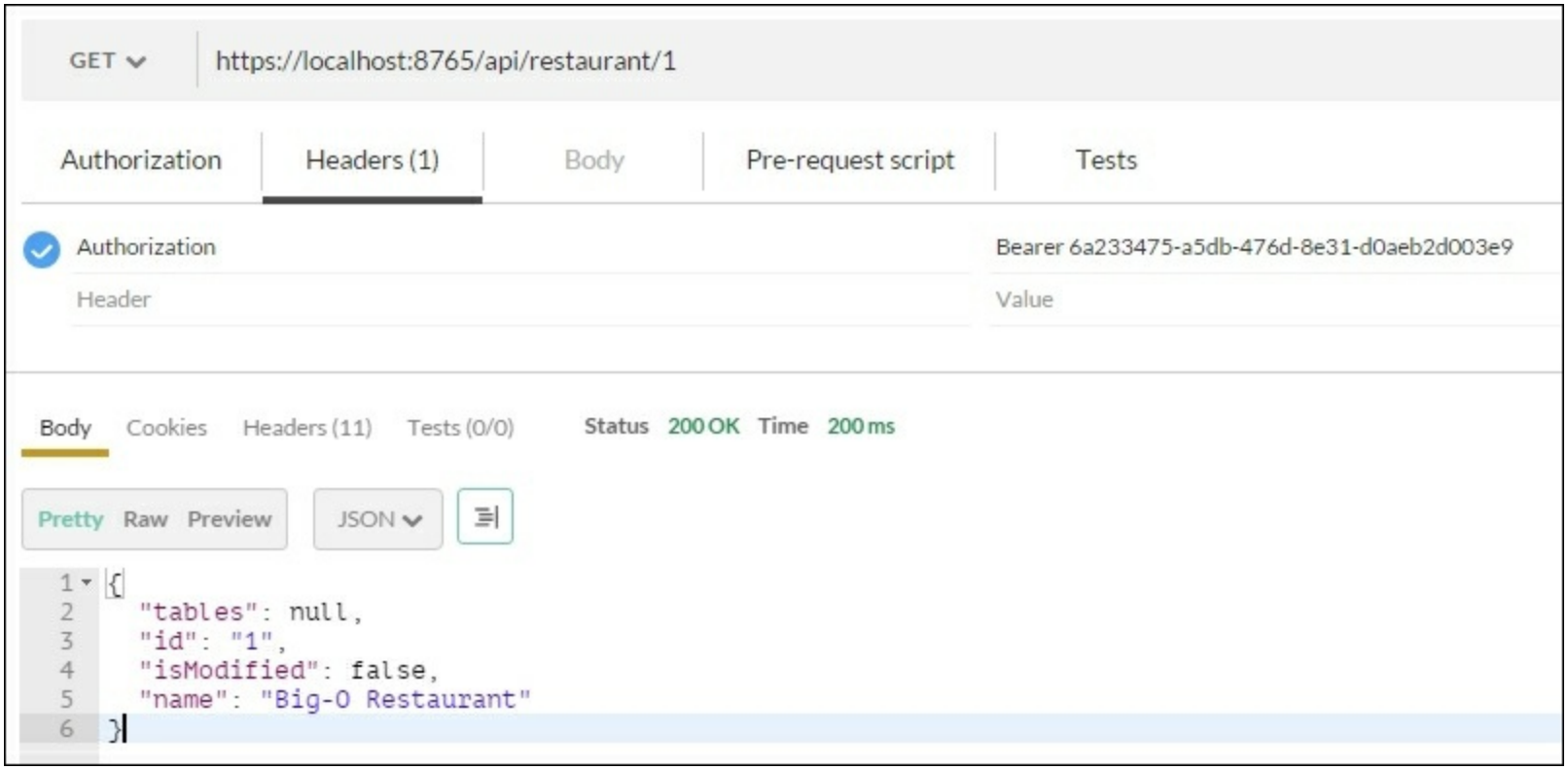
This returns the following response, as per the OAuth 2.0 specification:

```
{
  "access_token": "6a233475-a5db-476d-8e31-d0aeb2d003e9",
  "token_type": "bearer",
  "refresh_token": "8d91b9be-7f2b-44d5-b14b-dbbdccd848b8",
  "expires_in": 43199,
  "scope": "apiAccess"
}
```

Now we can use this information to access the resources owned by the resource owner. For example, if `https://localhost:8765/api/restaurant/1` represents the restaurant with the ID of `1`, then it should return the respective restaurant details.

Without the access token, if we enter the URL, it returns the error `Unauthorized`, with the message `Full authentication is required to access this resource.`

Now, let's access this URL with the access token, as shown in the following screenshot:



OAuth 2.0 authorization code grant – using the access token for API access

As you can see, we have added the **Authorization** header with the access token.

Now, we will explore implicit grant implementation.

Implicit grant

Implicit grants are very similar to authorization code grants, except for the code grant step. If you remove the first step—the code grant step (where the client application receives the authorization token from the authorization server)—from the authorization code grant, the rest of the steps are the same. Let's check it out.

Enter the following URL and parameters in the browser and press *Enter*. Also, make sure to add basic authentication, with client as the `username` and password as the `password` if asked:

```
https://localhost:9001/auth/oauth/authorize?
response_type=token&redirect_uri=https://localhost:8765&scope=apiAccess&state=55
```

Here, we are calling the authorization endpoint with the following request parameters: Response type, client ID, redirect URI, scope, and state.

When the request is successful, the browser will be redirected to the following URL with new request parameters and values:

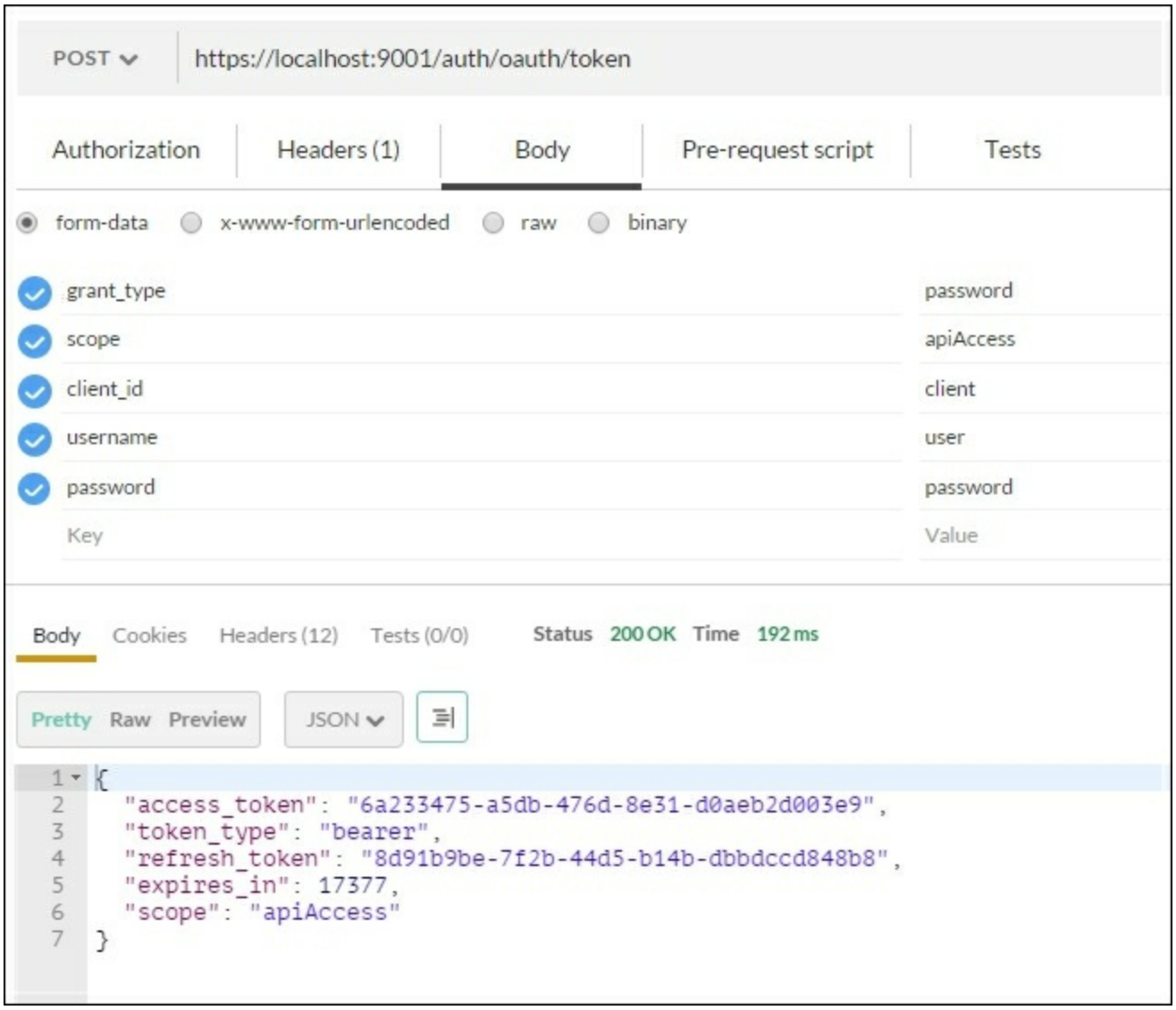
```
https://localhost:8765/#access_token=6a233475-a5db-476d-8e31-
d0aeb2d003e9&token_type=bearer&state=553344&expires_in=19592
```

Here, we receive the `access_token`, `token_type`, state, and expiry duration for the token. Now, we can make use of this access token to access the APIs, as used in the authorization code grant.

Resource owner password credential grant

In this grant, we provide the `username` and `password` as parameters when requesting the access token, along with the `grant_type`, `client`, and `scope` parameters. We also need to use the client ID and secret to authenticate the request. These grant flows use client applications in place of browsers, and are normally used in mobile and desktop apps.

In the following postman tool screenshot, the authorization header has already been added using basic authentication with `client_id` and `password`:



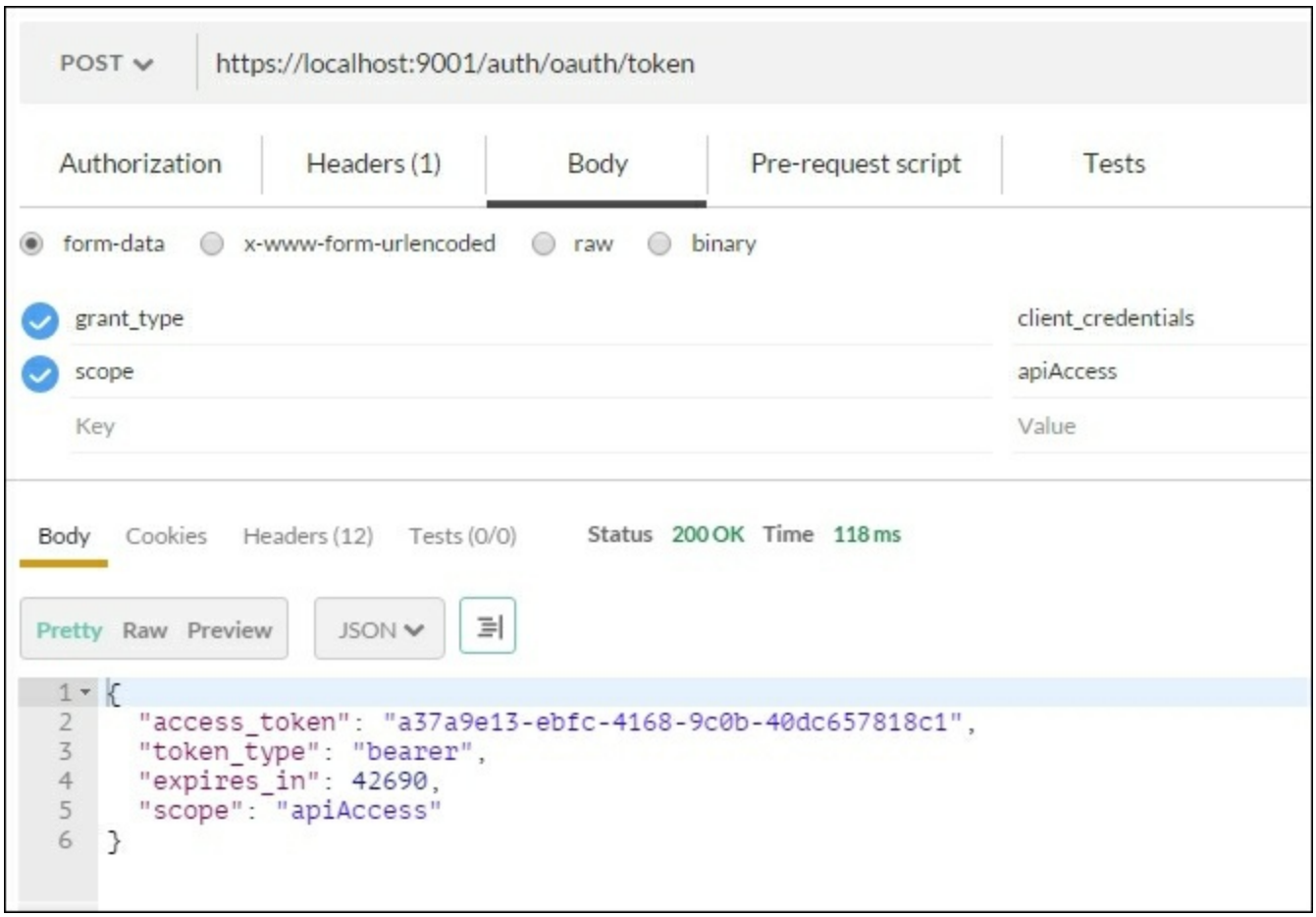
OAuth 2.0 resource owner password credentials grant – access token request and response

Once the access token is received by the client, it can be used in a similar way to how it is used in the authorization code grant.

Client credentials grant

In this flow, the client provides their own credentials and retrieves the access token. It does not use the resource owner's credentials and permissions.

As you can see in the following screenshot, we directly enter the token endpoint with only two parameters: `grant_type` and `scope`. The authorization header is added using `client_id` and `client secret`:



OAuth 2.0 client credentials grant – access token request and response

You can use the access token similarly as it is explained for the authorization code grant.

References

For more information, you refer to these links:

- *RESTful Java Web Services Security*, *Packt Publishing*, by René Enríquez, Andrés Salazar C: https://www.packtpub.com/application-development/restful-java-web-services-security
- *Spring Security [Video]*, *Packt Publishing*: https://www.packtpub.com/application-development/spring-security-video
- The OAuth 2.0 Authorization Framework: https://tools.ietf.org/html/rfc6749
- Spring Security: http://projects.spring.io/spring-security
- Spring OAuth2: http://projects.spring.io/spring-security-oauth/

Summary

In this chapter, we have learned how important it is to have the TLS layer or HTTPS in place for all web traffic. We have added a self-signed certificate to our sample application. I would like to reiterate that, for a production application, you must use the certificates offered by certificate signing authorities. We have also explored the fundamentals of OAuth 2.0 and various OAuth 2.0 grant flows. Different OAuth 2.0 grant flows are implemented using Spring Security and OAuth 2.0. In the next chapter, we'll implement the UI for the sample OTRS project and will explore how all the components work together.

Chapter 7. Consuming Services Using a Microservice Web App

Now, after developing the microservices, it would be interesting to see how the services offered by the **Online Table Reservation System** (**OTRS**) could be consumed by web or mobile applications. We will develop the web application (UI) using AngularJS/bootstrap to build the prototype of the web application. This sample application will display the data and flow of this sample project – a small utility project. This web application will also be a sample project and will run independently. Earlier, web applications were being developed in single web archives (files with `.war` extensions) that contain both UI and server-side code. The reason for doing so was pretty simple as UI was also developed using Java with JSPs, servlets, JSF, and so on. Nowadays, UIs are being developed independently using JavaScript. Therefore, these UI apps also deploy as a single microservice. In this chapter, we'll explore how these independent UI applications are being developed. We will develop and implement the OTRS sample app without login and authorization flow. We'll deploy a very limited functionality implementation and cover the high level AngularJS concepts. For more information on AngularJS, you can refer to *AngularJS by Example*, *Chandermani*, *Packt publishing*.

In this chapter, we will cover the following topics:

- AngularJS framework overview
- Development of OTRS features
- Setting up a web app (UI)

AngularJS framework overview

Now since we are ready with our HTML5 web app setup, we can go through the basics of AngularJS. This will help us to understand the AngularJS code. This section depicts the high level of understanding that you can utilize to understand the sample app and explore further using AngularJS documentation or by referring to other Packt publications.

AngularJS is a client side JavaScript framework. It is flexible enough to be used as a **MVC** (**Model View Controller**) or **MVVM** (**Model-View-ViewModel**). It also provides built-in services like `$http` or `$log` using a dependency injection pattern.

MVC

MVC is well-known design pattern. Struts and Spring MVC are popular examples. Let's see how they fit in the JavaScript world:

- **Model**: Models are JavaScript objects that contain the application data. They also represent the state of the application.
- **View**: View is a presentation layer that consists of HTML files. Here, you can show the data from models and provide the interactive interface to the user.
- **Controller**: You can define the controller in JavaScript and it contains the application logic.

MVVM

MVVM is an architecture design pattern that specifically targets the UI development. MVVM is designed to make two-way data binding easier. Two-way data binding provides the synchronization between the Model and View. When the Model (data) changes, it reflects immediately on the View. Similarly, when the user changes the data on the View, it reflects on the Model.

- **Model**: This is very similar to MVC and contains the business logic and data.
- **View**: Like MVC, it contains the presentation logic or user interface.
- **ViewModel**: ViewModel contains the data binding between the View and Model. Therefore, it is an interface between the View and Model.

Modules

A module is the first thing we define for any AngularJS application. A module is a container that contains the different parts of the app such as controllers, services, filters, and so on. An AngularJS app can be written in a single module or multiple modules. An AngularJS module can contain other modules also.

Many other JavaScript frameworks use the `main` method for instantiating and wiring the different parts of the app. AngularJS does not have the `main` method. It uses the module as an entry point due to following reasons:

- **Modularity**: You can divide and create your application feature-wise or with reusable components.
- **Simplicity**: You might have come across complex and large application code, which makes maintenance and enhancement a headache. No more, AngularJS makes code simple, readable, and easy to understand.
- **Testing**: It makes unit testing and end-to-end testing easier as you can override configuration and load only those modules which are required.

Each AngularJS app needs to have a single module for bootstrapping the AngularJS app. Bootstrapping our app requires the following three parts:

- **App module**: A JavaScript file (`app.js`) that contains the AngularJS module as shown:

```
var otrsApp = AngularJS.module('otrsApp', [ ])
// [] contains the reference to other modules
```

- **Loading Angular library and app module**: An `index.html` file containing the reference to the JavaScript file with other AngularJS libraries:

```
<script type="text/javascript" src="AngularJS/AngularJS.js"/>
<script type="text/javascript" src="scripts/app.js"/></script>
```

- **App DOM configuration**: This tells the AngularJS location of the DOM element where bootstrapping should take place. It can be done in either of two ways:
  - `Index.html` file that also contains an HTML element (typically `<html>`) with the `ng-app` (AngularJS directive) attribute having the value given in `app.js`. AngularJS directives are prefixed with `ng` (AngularJS): `<html lang="en" ng-app="otrsApp" class="no-js">`.
  - Or use this command if you are loading the JavaScript files asynchronously: `AngularJS.bootstrap(document.documentElement, ['otrsApp']);`.

An AngularJS module has two important parts, `config()` and `run()`, apart from other components like controllers, services, filters, and so on.

- `config()` is used for registering and configuring the modules and it only entertains the providers and constants using `$injector`. `$injector` is an AngularJS service. We'll cover providers and `$injector` in the next section. You cannot use instances here. It prevents the use of services

before it is fully configured.

- `run()` is used for executing the code after `$injector` is created using the preceding config method. This only entertains the instances and constants. You cannot use providers here to avoid configuration at run time.

Providers and services

Let's have a look at the following code:

```
.controller('otrsAppCtrl', function ($injector) {
var log = $injector.get('$log');
```

`$log` is an inbuilt AngularJS service that provides the logging API. Here, we are using another inbuilt service, `$injector`, that allows us to use the `$log` service. `$injector` is an argument in the controller. AngularJS uses function definitions and regex to provide the `$injector` service to a caller, also known as the controller. These are examples of how AngularJS effectively uses the dependency injection pattern.

AngularJS heavily uses the dependency injection pattern. AngularJS uses the injector service (`$injector`) to instantiate and wire most of the objects we use in our AngularJS app. This injector creates two types of objects – services and specialized objects.

For simplification, you can say that we (developers) define services. On the contrary, specialized objects are AngularJS stuff like controllers, filters, directives, and so on.

AngularJS provides five recipe types that tell the injector how to create service objects – **provider**, **value**, **factory**, **service**, and **constant**.

- The provider is the core and most complex recipe type. Other recipes are synthetic sugar on it. We generally avoid using the provider except when we need to create reusable code that requires global configuration.
- The value and constant recipe types works as their name suggests. Both cannot have dependencies. Moreover, the difference between them lies with their usage. You cannot use value service objects in the configuration phase.
- The factory and service are the most used services types. They are of a similar type. We use the factory recipe when we want to produce JavaScript primitives and functions. On the other hand, the service is used when we want to produce custom defined types.

As we have now some understanding of services, we can say that there are two common uses of services – organizing code and sharing code across apps. Services are singleton objects, which are lazily instantiated by the AngularJS service factory. By now, we have already seen a few of the in-built AngularJS services like `$injector`, `$log`, and so on. AngularJS services are prefixed with the `$` symbol.

Scopes

In AngularJS apps, two types of scopes are widely used: `$rootScope` and `$scope`:

- `$rootScope` is the top most object in the scope hierarchy and has the global scope associated with it. That means that any variable you attached to it will be available everywhere and therefore use of `$rootScope` should be a carefully considered decision.
- Controllers have `$scope` as an argument in the callback function. It is used for binding data from the controller to the view. Its scope is limited to the use of the controller it is associated with.

Controllers

The controller is defined by the JavaScript `constructor` function as having a `$scope` as an argument. The controller's main purpose is to tie the data to the view. The controller function is also used for writing business logic – setting up the initial state of the `$scope` object and adding the behavior to `$scope`. The controller signature looks like the following:

```
RestModule.controller('RestaurantsCtrl', function ($scope, restaurantService) {
```

Here, the controller is a part of `RestModule`. The name of the controller is `RestaurantCtrl`. `$scope` and `restaurantService` are passed as arguments.

Filters

The purpose of filters is to format the value of a given expression. In the following code we have defined the `datetime1` filter that takes date as an argument and changes the value in *dd MMM yyyy HH:mm* format like *04 Apr 2016 04:13 PM*.

```
.filter('datetime1', function ($filter) {
    return function (argDateTime) {
        if (argDateTime) {
            return $filter('date')(new Date(argDateTime), 'dd MMM yyyy HH:mm
a');
        }
        return "";
    };
});
```

Directives

As we have seen in the *Modules* section, AngularJS directives are HTML attributes with an `ng` prefix. Some of the popular directives are:

- `ng-app`: This directive defines the AngularJS application
- `ng-model`: This directive binds the HTML form input to data
- `ng-bind`: This directive binds the data to the HTML view
- `ng-submit`: This directive submits the HTML form
- `ng-repeat`: This directive iterates the collection

```
<div ng-app="">
    <p>Search: <input type="text" ng-model="searchValue"></p>
    <p ng-bind="searchedTerm"></p>
</div>
```

UI-Router

In **single page applications** (**SPAs**), the page only loads once and the user navigates through different links without page refresh. It is all possible because of routing. Routing is a way to make SPA navigation feel like a normal site. Therefore, routing is very important for SPA.

The AngularUI team built UI-Router, an AngularJS routing framework. UI-Router is not a part of core AngularJS. UI-Router not only changes the route URL, but it also changes the state of the application when the user clicks on any link in the SPA. Because UI-Router can also make state changes, you can change the view of the page without changing the URL. This is possible because of the application state management by UI-Router.

If we consider the SPA as a state machine then the state is a current state of the application. We will use the attribute `ui-sref` in a HTML link tag when we create the route link. The attribute `href` in the link will be generated from this and point to certain states of the application which are created in `app.js`.

We use the `ui-view` attribute in the HTML div to use UI-Router: for example, `<div ui-view> </div>`.

Development of OTRS features

As you know, we are developing the SPA. Therefore, once the application loads, you can perform all the operations without page refresh. All interactions with the server are performed using AJAX calls. Now, we'll make use of the AngularJS concepts that we have covered in the first section. We'll cover the following scenarios:

- A page that will display a list of restaurants. This will also be our home page.
- Search restaurants.
- Restaurant details with reservation options.
- Login (not from the server, but used for displaying the flow).
- Reservation confirmation.

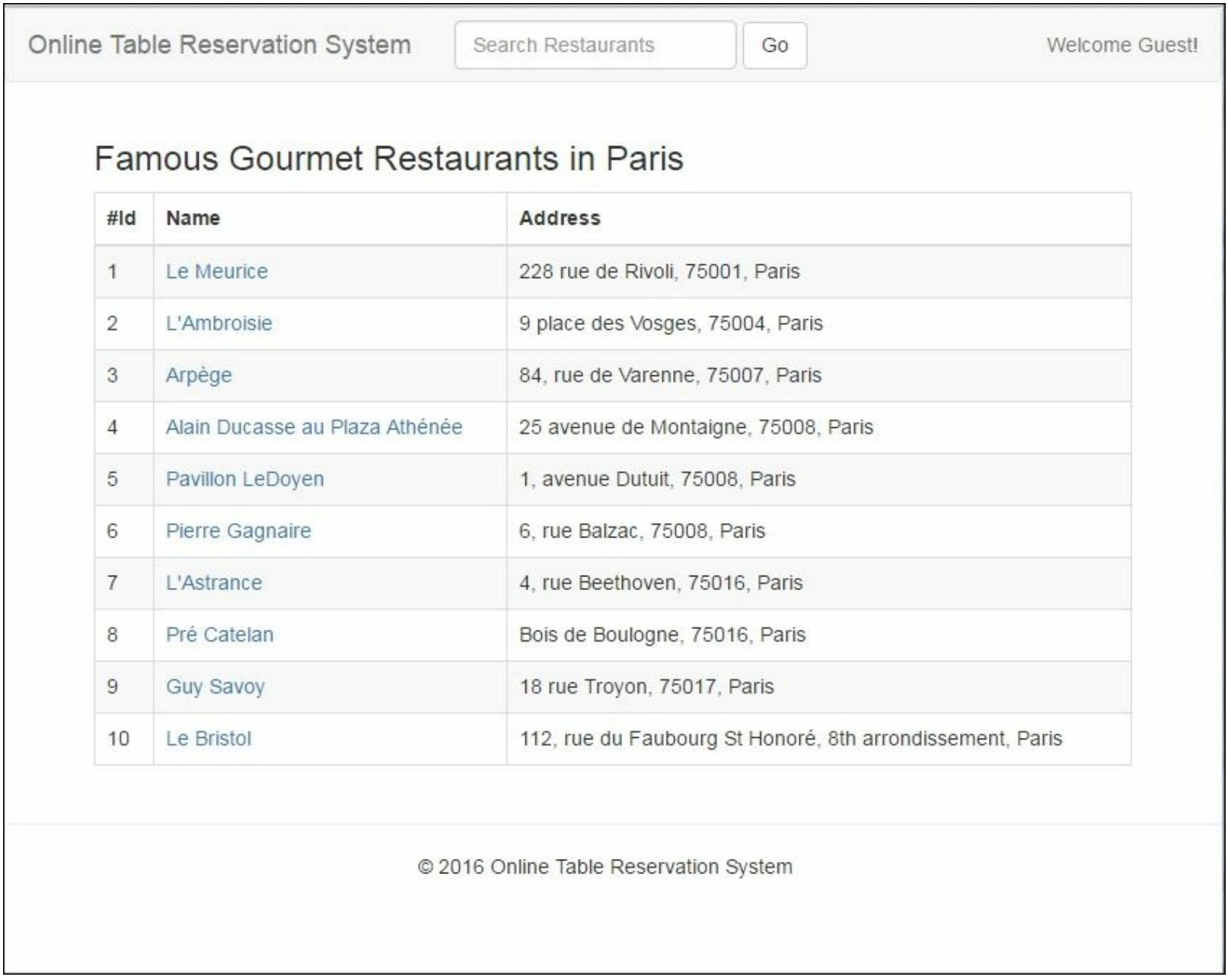
For the home page, we will create `index.html` and a template that will contain the restaurant listing in the middle section or the content area.

Home page/restaurant list page

The home page is the main page of any web application. To design the home page, we are going to use the Angular-UI bootstrap rather than the actual bootstrap. Angular-UI is an Angular version of the bootstrap. The home page will be divided into three sections:

- The header section will contain the app name, search restaurants form, and user name at top-right corner.
- The content or middle section will contain the restaurant listing which will have the restaurant name as the link. This link will point to the restaurant details and reservation page.
- The footer section will contain the app name with the copyright mark.

You must be interested in viewing the home page before designing or implementing it. Therefore, let us first see how it will look like once we have our content ready:



OTRS home page with restaurants listing

Now, to design our home page, we need to add following four files:

- `index.html`: Our main HTML file
- `app.js`: Our main AngularJS module
- `restaurants.js`: The restaurants module that also contains the restaurant Angular service
- `restaurants.html`: The HTML template that will display the list of restaurants

index.html

First, we'll add the `./app/index.html` in our project workspace. The contents of `index.html` will be as explained here onwards.

Note

I have added comments in between the code to make the code more readable and make it easier to understand.

`index.html` is divided into many parts. We'll discuss a few of the key parts here. First, we will see how to address old Internet Explorer versions. If you want to target the Internet Explorer browser versions greater than 8 or IE version 9 onwards, then we need to add following block that will prevent JavaScript rendering and give the no-js output to the end-user.

```
<!--[if lt IE 7]>      <html lang="en" ng-app="otrsApp" class="no-js lt-ie9 lt-
ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>         <html lang="en" ng-app="otrsApp" class="no-js lt-ie9 lt-
ie8"> <![endif]-->
<!--[if IE 8]>         <html lang="en" ng-app="otrsApp" class="no-js lt-ie9"> <!
[endif]-->
<!--[if gt IE 8]><!--> <html lang="en" ng-app="otrsApp" class="no-js"> <!--<!
[endif]-->
```

Then, after adding a few meta tags and the title of the application, we'll also define the important meta `tag` viewport. The viewport is used for responsive UI designs.

The `width` property defined in the content attribute controls the size of the viewport. It can be set to a specific number of pixels like `width = 600` or to the special value device-width value which is the width of the screen in CSS pixels at a scale of 100%.

The initial-scale property controls the zoom level when the page is first loaded. The maximum-scale, minimum-scale, and user-scalable properties control how users are allowed to zoom the page in or out.

```
        <meta name="viewport" content="width=device-width, initial-scale=1">
```

In the next few lines, we'll define the style sheets of our application. We are adding `normalize.css` and `main.css` from HTML5 boilerplate code. We are also adding our application's customer CSS `app.css`. Finally, we are adding the bootstrap 3 CSS. Apart from the customer `app.css`, other CSS

are referenced in it. There is no change in these CSS files.

```
<link rel="stylesheet" href="bower_components/html5-
boilerplate/dist/css/normalize.css">
        <link rel="stylesheet" href="bower_components/html5-
boilerplate/dist/css/main.css">
        <link rel="stylesheet" href="public/css/app.css">
        <link data-require="bootstrap-css@*" data-server="3.0.0"
rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css" />
```

Then we'll define the scripts using the `script` tag. We are adding the modernizer, Angular, Angular-route, and our own developed custom JavaScript file `app.js`. We have already discussed Angular and Angular-UI. `app.js` will be discussed in the next section.

Modernizer allows web developers to use new CSS3 and HTML5 features while maintaining a fine level of control over browsers that don't support them. Basically, modernizer performs the next generation feature detection (checking the availability of those features) while the page loads in the browser and reports the results. Based on these results you can detect what are the latest features available in the browser and based on that you can provide an interface to the end user. If the browser does not support a few of the features then an alternate flow or UI is provided to the end user.

We are also adding the bootstrap templates which are written in JavaScript using the `ui-bootstrap-tpls javascript` file.

```
        <script src="bower_components/html5-
boilerplate/dist/js/vendor/modernizr-2.8.3.min.js"></script>
        <script src="bower_components/angular/angular.min.js"></script>
        <script src="bower_components/angular-route/angular-route.min.js">
</script>
        <script src="app.js"></script>
      <script data-require="ui-bootstrap@0.5.0" data-semver="0.5.0"
src="http://angular-ui.github.io/bootstrap/ui-bootstrap-tpls-0.6.0.js"></script>
```

We can also add style to the `head` tag as shown in the following. This style allows drop-down menus to work.

```
        <style>
            div.navbar-collapse.collapse {
                display: block;
                overflow: hidden;
                max-height: 0px;
                -webkit-transition: max-height .3s ease;
                -moz-transition: max-height .3s ease;
                -o-transition: max-height .3s ease;
                transition: max-height .3s ease;
            }
            div.navbar-collapse.collapse.in {
                max-height: 2000px;
            }
        </style>
```

In the `body` tag we are defining the controller of the application using the `ng-controller` attribute. While the page loads, it tells the controller the name of the application to Angular.

```
    <body ng-controller="otrsAppCtrl">
```

Then, we define the `header` section of the home page. In the `header` section, we'll define the application title, `Online Table Reservation System`. Also, we'll define the search form that will search the restaurants.

```
<!-- BEGIN HEADER -->
        <nav class="navbar navbar-default" role="navigation">

            <div class="navbar-header">
                <a class="navbar-brand" href="#">
                    Online Table Reservation System
                </a>
            </div>
            <div class="collapse navbar-collapse" ng-class="!navCollapsed &&
'in'" ng-click="navCollapsed = true">
                <form class="navbar-form navbar-left" role="search" ng-
submit="search()">
                    <div class="form-group">
                        <input type="text" id="searchedValue" ng-
model="searchedValue" class="form-control" placeholder="Search Restaurants">
                    </div>
                    <button type="submit" class="btn btn-default" ng-
click="">Go</button>
                </form>
        <!-- END HEADER -->
```

Then, in the next section, the middle section, includes where we actually bind the different views, marked with actual content comments. The `ui-view` attribute in `div` gets its content dynamically from Angular such as restaurant details, restaurant list, and so on. We have also added a warning dialog and spinner to the middle section that will be visible as and when required.

```
        <div class="clearfix"></div>
        <!-- BEGIN CONTAINER -->
        <div class="page-container container">
            <!-- BEGIN CONTENT -->
            <div class="page-content-wrapper">
                <div class="page-content">
                    <!-- BEGIN ACTUAL CONTENT -->
                    <div ui-view class="fade-in-up"></div>
                    <!-- END ACTUAL CONTENT -->
                </div>
            </div>
            <!-- END CONTENT -->
        </div>
        <!-- loading spinner -->
        <div id="loadingSpinnerId" ng-show="isSpinnerShown()" style="top:0;
left:45%; position:absolute; z-index:999">
            <script type="text/ng-template" id="alert.html">
```

```
                <div class="alert alert-warning" role="alert">
                <div ng-transclude></div>
                </div>
            </script>
            <uib-alert type="warning" template-url="alert.html"><b>Loading...
</b></uib-alert>
        </div>
        <!-- END CONTAINER -->
```

The final section of the `index.html` is the footer. Here, we are just adding the static content and copyright text. You can add whatever content you want here.

```
        <!-- BEGIN FOOTER -->
        <div class="page-footer">
            <hr/><div style="padding: 0 39%">&copy; 2016 Online Table
Reservation System</div>
        </div>
        <!-- END FOOTER -->
    </body>
</html>
```

app.js

`app.js` is our main application file. Because we have defined it in `index.html`, it gets loaded as soon as our `index.html` is called.

Tip

We need to take care that we do not mix, route (URI) with REST endpoints. Routes represents the state/view of the SPA.

As we are using the Edge Server (Proxy Server), everything will be accessible from it including our REST endpoints. External applications including the UI will use the Edge Server host to access the application. You can configure it in some global constants file and then use it wherever it is required. This will allow you to configure the REST host at a single place and use it at other places.

```
'use strict';
/*
This call initializes our application and registers all the modules, which are
passed as an array in the second argument.
*/
var otrsApp = angular.module('otrsApp', [
    'ui.router',
    'templates',
    'ui.bootstrap',
    'ngStorage',
    'otrsApp.httperror',
    'otrsApp.login',
    'otrsApp.restaurants'
])
/*
  Then we have defined the default route /restaurants
```

```
*/
        .config([
            '$stateProvider', '$urlRouterProvider',
            function ($stateProvider, $urlRouterProvider) {
                $urlRouterProvider.otherwise('/restaurants');
            }])
/*
        This functions controls the flow of the application and handles the
events.
*/
        .controller('otrsAppCtrl', function ($scope, $injector,
restaurantService) {
            var controller = this;

            var AjaxHandler = $injector.get('AjaxHandler');
            var $rootScope = $injector.get('$rootScope');
            var log = $injector.get('$log');
            var sessionStorage = $injector.get('$sessionStorage');
            $scope.showSpinner = false;
/*
        This function gets called when the user searches any restaurant. It uses
the Angular restaurant service that we'll define in the next section to search
the given search string.
*/
            $scope.search = function () {
                $scope.restaurantService = restaurantService;
                restaurantService.async().then(function () {
                    $scope.restaurants =
restaurantService.search($scope.searchedValue);
                });
            }
/*
        When the state is changed, the new controller controls the flows based
on the view and configuration and the existing controller is destroyed. This
function gets a call on the destroy event.
*/
            $scope.$on('$destroy', function destroyed() {
                log.debug('otrsAppCtrl destroyed');
                controller = null;
                $scope = null;
            });

            $rootScope.fromState;
            $rootScope.fromStateParams;
            $rootScope.$on('$stateChangeSuccess', function (event, toState,
toParams, fromState, fromStateParams) {
                $rootScope.fromState = fromState;
                $rootScope.fromStateParams = fromStateParams;
            });

            // utility method
            $scope.isLoggedIn = function () {
                if (sessionStorage.session) {
                    return true;
```

```
                } else {
                    return false;
                }
            };

            /* spinner status */
            $scope.isSpinnerShown = function () {
                return AjaxHandler.getSpinnerStatus();
            };

        })
/*
        This function gets executed when this object loads. Here we are setting
the user object which is defined for the root scope.
*/
        .run(['$rootScope', '$injector', '$state', function ($rootScope,
$injector, $state) {
                $rootScope.restaurants = null;
                // self reference
                var controller = this;
                // inject external references
                var log = $injector.get('$log');
                var $sessionStorage = $injector.get('$sessionStorage');
                var AjaxHandler = $injector.get('AjaxHandler');

                if (sessionStorage.currentUser) {
                    $rootScope.currentUser = $sessionStorage.currentUser;
                } else {
                    $rootScope.currentUser = "Guest";
                    $sessionStorage.currentUser = ""
                }
            }])
```

restaurants.js

`restaurants.js` represents an Angular service for our app which we'll use for the restaurants. We know that there are two common uses of services – organizing code and sharing code across apps. Therefore, we have created a restaurants service which will be used among different modules like search, list, details, and so on.

Note

Services are singleton objects, which are lazily instantiated by the AngularJS service factory.

The following section initializes the restaurant service module and loads the required dependencies.

```
angular.module('otrsApp.restaurants', [
    'ui.router',
    'ui.bootstrap',
    'ngStorage',
    'ngResource'
])
```

In the configuration, we are defining the routes and state of the `otrsApp.restaurants` module using UI-Router.

First we define the `restaurants` state by passing the JSON object containing the URL that points the router URI, the template URL that points to the HTML template that display the `restaurants` state, and the controller that will handle the events on the `restaurants` view.

On top of the `restaurants` view (`route - /restaurants`), a nested state `restaurants.profile` is also defined that will represent the specific restaurant. For example, `/restaurant/1` would open and display the restaurant profile (details) page of a restaurant which is represented by `Id 1`. This state is called when a link is clicked in the `restaurants` template. In this `ui-sref="restaurants.profile({id: rest.id})"` rest represents the `restaurant` object retrieved from the `restaurants` view.

Notice that the state name is `'restaurants.profile'` which tells the AngularJS UI Router that the profile is a nested state of the `restaurants` state.

```
.config([
    '$stateProvider', '$urlRouterProvider',
    function ($stateProvider, $urlRouterProvider) {
        $stateProvider.state('restaurants', {
            url: '/restaurants',
            templateUrl: 'restaurants/restaurants.html',
            controller: 'RestaurantsCtrl'
        })
                // Restaurant show page
                .state('restaurants.profile', {
                    url: '/:id',
                    views: {
                        '@': {
                            templateUrl: 'restaurants/restaurant.html',
                            controller: 'RestaurantCtrl'
                        }
                    }
                });
    }])
```

In the next code section, we are defining the restaurant service using the Angular factory service type. This restaurant service on load fetches the list of restaurants from the server using a REST call. It provides a list and searches restaurant operations and restaurant data.

```
.factory('restaurantService', function ($injector, $q) {
    var log = $injector.get('$log');
    var ajaxHandler = $injector.get('AjaxHandler');
    var deffered = $q.defer();
    var restaurantService = {};
    restaurantService.restaurants = [];
    restaurantService.orignalRestaurants = [];
    restaurantService.async = function () {
        ajaxHandler.startSpinner();
```

```
            if (restaurantService.restaurants.length === 0) {
                ajaxHandler.get('/api/restaurant')
                        .success(function (data, status, headers, config) {
                            log.debug('Getting restaurants');
                            sessionStorage.apiActive = true;
                            log.debug("if Restaurants --> " +
restaurantService.restaurants.length);
                            restaurantService.restaurants = data;
                            ajaxHandler.stopSpinner();
                            deffered.resolve();
                        })
                        .error(function (error, status, headers, config) {
                            restaurantService.restaurants = mockdata;
                            ajaxHandler.stopSpinner();
                            deffered.resolve();
                        });
                return deffered.promise;
            } else {
                deffered.resolve();
                ajaxHandler.stopSpinner();
                return deffered.promise;
            }
        };
        restaurantService.list = function () {
            return restaurantService.restaurants;
        };
        restaurantService.add = function () {
            console.log("called add");
            restaurantService.restaurants.push(
                    {
                        id: 103,
                        name: 'Chi Cha\'s Noodles',
                        address: '13 W. St., Eastern Park, New County,
Paris',
                    });
        };
        restaurantService.search = function (searchedValue) {
            ajaxHandler.startSpinner();
            if (!searchedValue) {
                if (restaurantService.orignalRestaurants.length > 0) {
                    restaurantService.restaurants =
restaurantService.orignalRestaurants;
                }
                deffered.resolve();
                ajaxHandler.stopSpinner();
                return deffered.promise;
            } else {
                ajaxHandler.get('/api/restaurant?name=' + searchedValue)
                        .success(function (data, status, headers, config) {
                            log.debug('Getting restaurants');
                            sessionStorage.apiActive = true;
                            log.debug("if Restaurants --> " +
restaurantService.restaurants.length);
                            if (restaurantService.orignalRestaurants.length
```

```
< 1) {
                                    restaurantService.orignalRestaurants =
restaurantService.restaurants;
                                }
                                restaurantService.restaurants = data;
                                ajaxHandler.stopSpinner();
                                deffered.resolve();
                            })
                            .error(function (error, status, headers, config) {
                                if (restaurantService.orignalRestaurants.length
< 1) {
                                    restaurantService.orignalRestaurants =
restaurantService.restaurants;
                                }
                                restaurantService.restaurants = [];
                                restaurantService.restaurants.push(
                                        {
                                            id: 104,
                                            name: 'Gibsons - Chicago Rush St.',
                                            address: '1028 N. Rush St., Rush &
Division, Cook County, Paris'
                                        });
                                restaurantService.restaurants.push(
                                        {
                                            id: 105,
                                            name: 'Harry Caray\'s Italian
Steakhouse',
                                            address: '33 W. Kinzie St., River
North, Cook County, Paris',
                                        });
                                ajaxHandler.stopSpinner();
                                deffered.resolve();
                            });
                    return deffered.promise;
                }
            };
            return restaurantService;
        })
```

In the next section of the `restaurants.js` module, we'll add two controllers that we defined for the restaurants and `restaurants.profile` states in the routing configuration. These two controllers are `RestaurantsCtrl` and `RestaurantCtrl` that handle the `restaurants` state and the `restaurants.profiles` states respectively.

`RestaurantsCtrl` is pretty simple in that it loads the restaurants data using the restaurants service list method.

```
        .controller('RestaurantsCtrl', function ($scope, restaurantService) {
            $scope.restaurantService = restaurantService;
            restaurantService.async().then(function () {
                $scope.restaurants = restaurantService.list();
            });
        })
```

`RestaurantCtrl` is responsible for showing the restaurant details of a given ID. This is also responsible for performing the reservation operations on the displayed restaurant. This control will be used when we design the restaurant details page with reservation options.

```
        .controller('RestaurantCtrl', function ($scope, $state, $stateParams,
$injector, restaurantService) {
            var $sessionStorage = $injector.get('$sessionStorage');
            $scope.format = 'dd MMMM yyyy';
            $scope.today = $scope.dt = new Date();
            $scope.dateOptions = {
                formatYear: 'yy',
                maxDate: new Date().setDate($scope.today.getDate() + 180),
                minDate: $scope.today.getDate(),
                startingDay: 1
            };

            $scope.popup1 = {
                opened: false
            };
            $scope.altInputFormats = ['M!/d!/yyyy'];
            $scope.open1 = function () {
                $scope.popup1.opened = true;
            };
            $scope.hstep = 1;
            $scope.mstep = 30;

            if ($sessionStorage.reservationData) {
                $scope.restaurant = $sessionStorage.reservationData.restaurant;
                $scope.dt = new Date($sessionStorage.reservationData.tm);
                $scope.tm = $scope.dt;
            } else {
                $scope.dt.setDate($scope.today.getDate() + 1);
                $scope.tm = $scope.dt;
                $scope.tm.setHours(19);
                $scope.tm.setMinutes(30);
                restaurantService.async().then(function () {
                    angular.forEach(restaurantService.list(), function (value,
key) {
                        if (value.id === parseInt($stateParams.id)) {
                            $scope.restaurant = value;
                        }
                    });
                });
            }
            $scope.book = function () {
                var tempHour = $scope.tm.getHours();
                var tempMinute = $scope.tm.getMinutes();
                $scope.tm = $scope.dt;
                $scope.tm.setHours(tempHour);
                $scope.tm.setMinutes(tempMinute);
                if ($sessionStorage.currentUser) {
                    console.log("$scope.tm --> " + $scope.tm);
                    alert("Booking Confirmed!!!");
                    $sessionStorage.reservationData = null;
```

```
                    $state.go("restaurants");
                } else {
                    $sessionStorage.reservationData = {};
                    $sessionStorage.reservationData.restaurant =
$scope.restaurant;
                    $sessionStorage.reservationData.tm = $scope.tm;
                    $state.go("login");
                }
            }
        })
```

We have also added a few of the filters in the `restaurants.js` module to format the date and time. These filters perform the following formatting on the input data:

- `date1`: Returns the input date in 'dd MMM yyyy' format, for example 13-Apr-2016
- `time1`: Returns the input time in 'HH:mm:ss' format, for example 11:55:04
- `dateTime1`: Returns the input date and time in 'dd MMM yyyy HH:mm:ss' format, for example 13-Apr-2016 11:55:04

In the following code snippet we've applied these three filters:

```
        .filter('date1', function ($filter) {
            return function (argDate) {
                if (argDate) {
                    var d = $filter('date')(new Date(argDate), 'dd MMM yyyy');
                    return d.toString();
                }
                return "";
            };
        })
        .filter('time1', function ($filter) {
            return function (argTime) {
                if (argTime) {
                    return $filter('date')(new Date(argTime), 'HH:mm:ss');
                }
                return "";
            };
        })
        .filter('datetime1', function ($filter) {
            return function (argDateTime) {
                if (argDateTime) {
                    return $filter('date')(new Date(argDateTime), 'dd MMM yyyy
HH:mm a');
                }
                return "";
            };
        });
```

restaurants.html

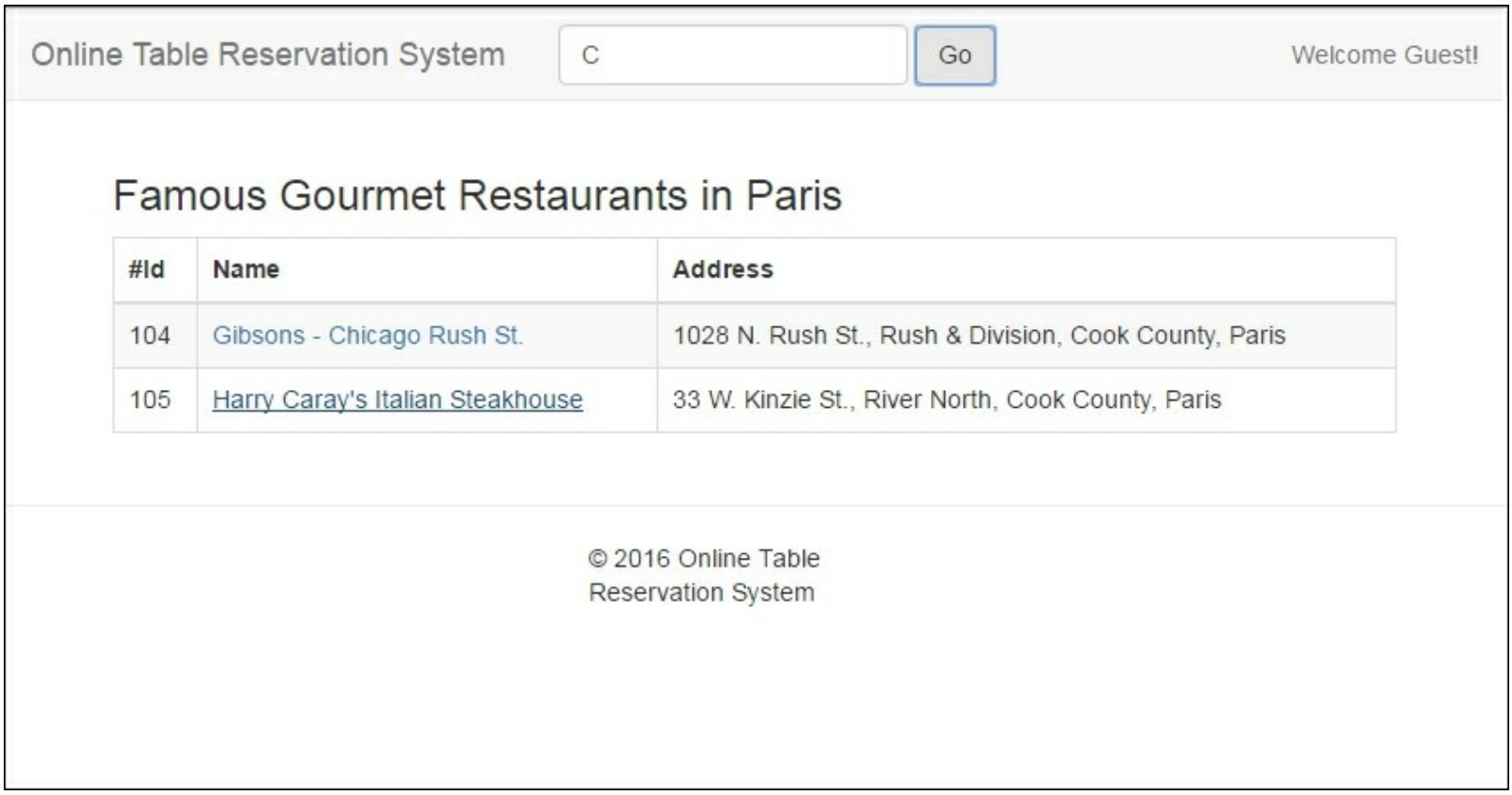
We need to add the templates that we have defined for the `restaurants.profile` state. As you can see in the template we are using the `ng-repeat` directive to iterate the list of objects returned by

`restaurantService.restaurants`. The `restaurantService` scope variable is defined in the controller. `'RestaurantsCtrl'` is associated with this template in the restaurants state.

```
<h3>Famous Gourmet Restaurants in Paris</h3>
<div class="row">
    <div class="col-md-12">
        <table class="table table-bordered table-striped">
            <thead>
                <tr>
                    <th>#Id</th>
                    <th>Name</th>
                    <th>Address</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="rest in restaurantService.restaurants">
                    <td>{{rest.id}}</td>
                    <td><a ui-sref="restaurants.profile({id: rest.id})">
{{rest.name}}</a></td>
                    <td>{{rest.address}}</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Search Restaurants

On the home page `index.html` we have added the search form in the `header` section that allows us to search restaurants. The Search Restaurants functionality will use the same files as described earlier. It makes use of the `app.js` (search form handler), `restaurants.js` (restaurant service), and `restaurants.html` to display the searched records.

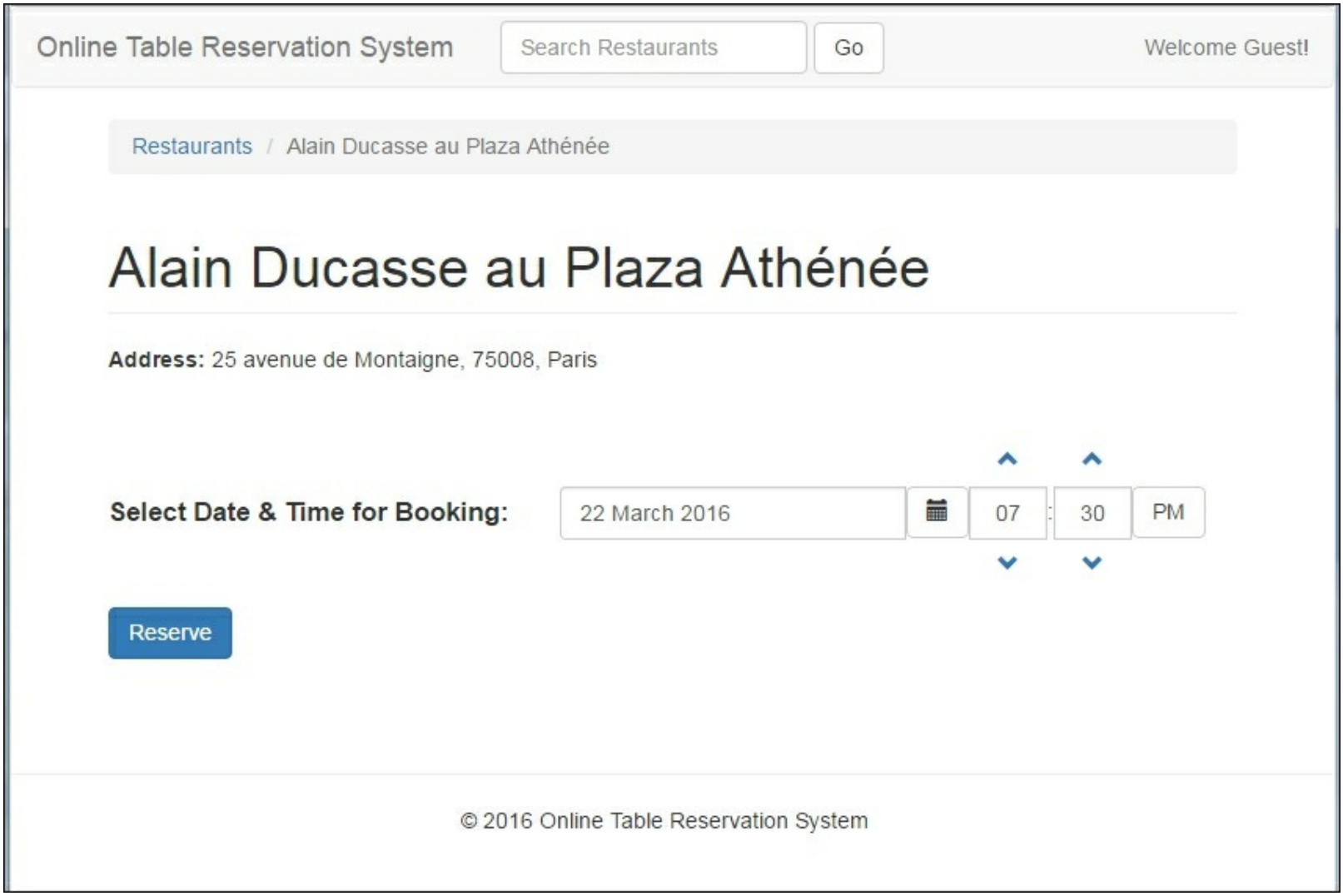


OTRS home page with restaurants listing

Restaurant details with reservation option

Restaurant details with reservation option will be the part of the content area (middle section of the page). This will contain a breadcrumb at the top with restaurants as a link to the restaurant listing page, followed by the name and address of the restaurant. The last section will contain the reservation section containing date time selection boxes and reserve button.

This page will look like the following screenshot:



Restaurants Detail Page with Reservation Option

Here, we will make use of the same restaurant service declared in `restaurants.js`. The only change will be the template as described for the state `restaurants.profile`. This template will be defined using the `restaurant.html`.

restaurant.html

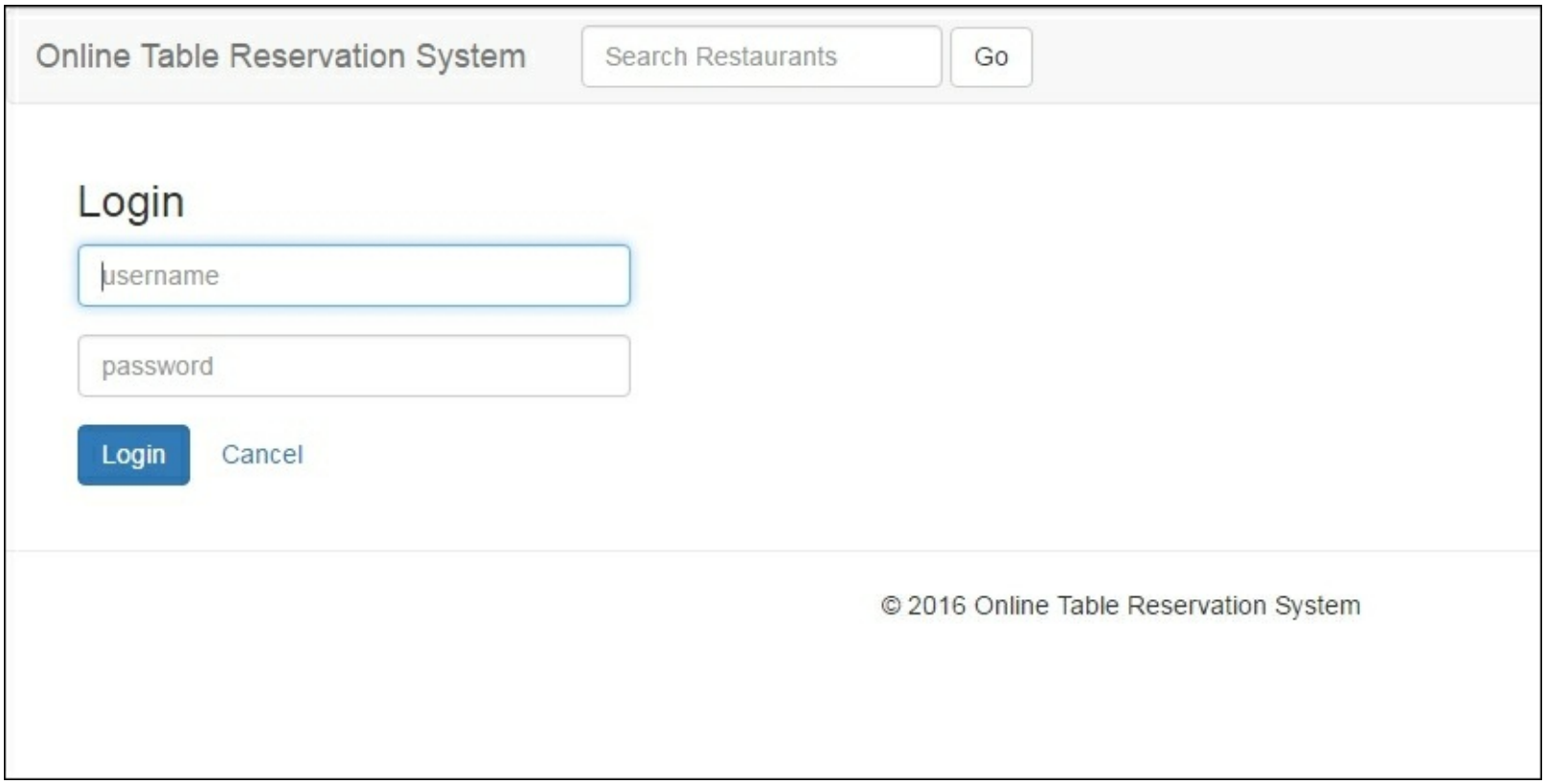
As you can see, the breadcrumb is using the restaurants route, which is defined using the `ui-sref` attribute. The reservation form designed in this template calls the `book()` function defined in the

controller `RestaurantCtrl` using the directive `ng-submit` on the form submit.

```
<div class="row">
<div class="row">
    <div class="col-md-12">
        <ol class="breadcrumb">
            <li><a ui-sref="restaurants">Restaurants</a></li>
            <li class="active">{{restaurant.name}}</li>
        </ol>
        <div class="bs-docs-section">
            <h1 class="page-header">{{restaurant.name}}</h1>
            <div>
                <strong>Address:</strong> {{restaurant.address}}
            </div>
            </br></br>
            <form ng-submit="book()">
                <div class="input-append date form_datetime">
                    <div class="row">
                        <div class="col-md-7">
                            <p class="input-group">
                                <span style="display: table-cell; vertical-
align: middle; font-weight: bolder; font-size: 1.2em">Select Date & Time for
Booking:</span>
                                <span style="display: table-cell; vertical-
align: middle">
                                    <input type="text" size=20 class="form-
control" uib-datepicker-popup="{{format}}" ng-model="dt" is-open="popup1.opened"
datepicker-options="dateOptions" ng-required="true" close-text="Close" alt-
input-formats="altInputFormats" />
                                </span>
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-
default" ng-click="open1()"><i class="glyphicon glyphicon-calendar"></i>
</button>
                                </span>
                            <uib-timepicker ng-model="tm" ng-change="changed()"
hour-step="hstep" minute-step="mstep"></uib-timepicker>
                            </p>
                        </div>
                    </div></div>
                <div class="form-group">
                    <button class="btn btn-primary"
type="submit">Reserve</button>
                </div>
            </form></br></br>
        </div>
    </div>
</div>
```

Login page

When a user clicks on the **Reserve** button on the **Restaurant Detail** page after selecting the date and time of the reservation, the **Restaurant Detail** page checks whether the user is already logged in or not. If the user is not logged in, then the **Login** page displays. It looks like the following screenshot:



Login page

Note

We are not authenticating the user from the server. Instead, we are just populating the user name in the session storage and rootscope for implementing the flow.

Once the user logs in, the user is redirected back to same booking page with the persisted state. Then the user can proceed with the reservation. The **Login** page uses basically two files: `login.html` and `login.js`.

login.html

The `login.html` template consists of only two input fields, username and password, with the **Login** button and **Cancel** link. The **Cancel** link resets the form and the **Login** button submits the login form.

Here, we are using the `LoginCtrl` with the `ng-controller` directive. The **Login** form is submitted using the `ng-submit` directive that calls the `submit` function of `LoginCtrl`. Input values are first collected using the `ng-model` directive and then submitted using their respective properties - `_email`

and `_password`.

```
<div ng-controller="LoginCtrl as loginC" style="max-width: 300px">
    <h3>Login</h3>
    <div class="form-container">
        <form ng-submit="loginC.submit(_email, _password)">
            <div class="form-group">
                <label for="username" class="sr-only">Username</label>
                <input type="text" id="username" class="form-control"
placeholder="username" ng-model="_email" required autofocus />
            </div>
            <div class="form-group">
                <label for="password" class="sr-only">Password</label>
                <input type="password" id="password" class="form-control"
placeholder="password" ng-model="_password" />
            </div>
            <div class="form-group">
                <button class="btn btn-primary" type="submit">Login</button>
                <button class="btn btn-link" ng-
click="loginC.cancel()">Cancel</button>
            </div>
        </form>
    </div>
</div>
```

login.js

The login module is defined in the `login.js` that contains and loads the dependencies using the module function. The state login is defined with the help of the config function that takes the JSON object containing the url, controller, and `templateUrl` properties.

Inside the controller, we define the cancel and submit operations, which are called from the `login.html` template.

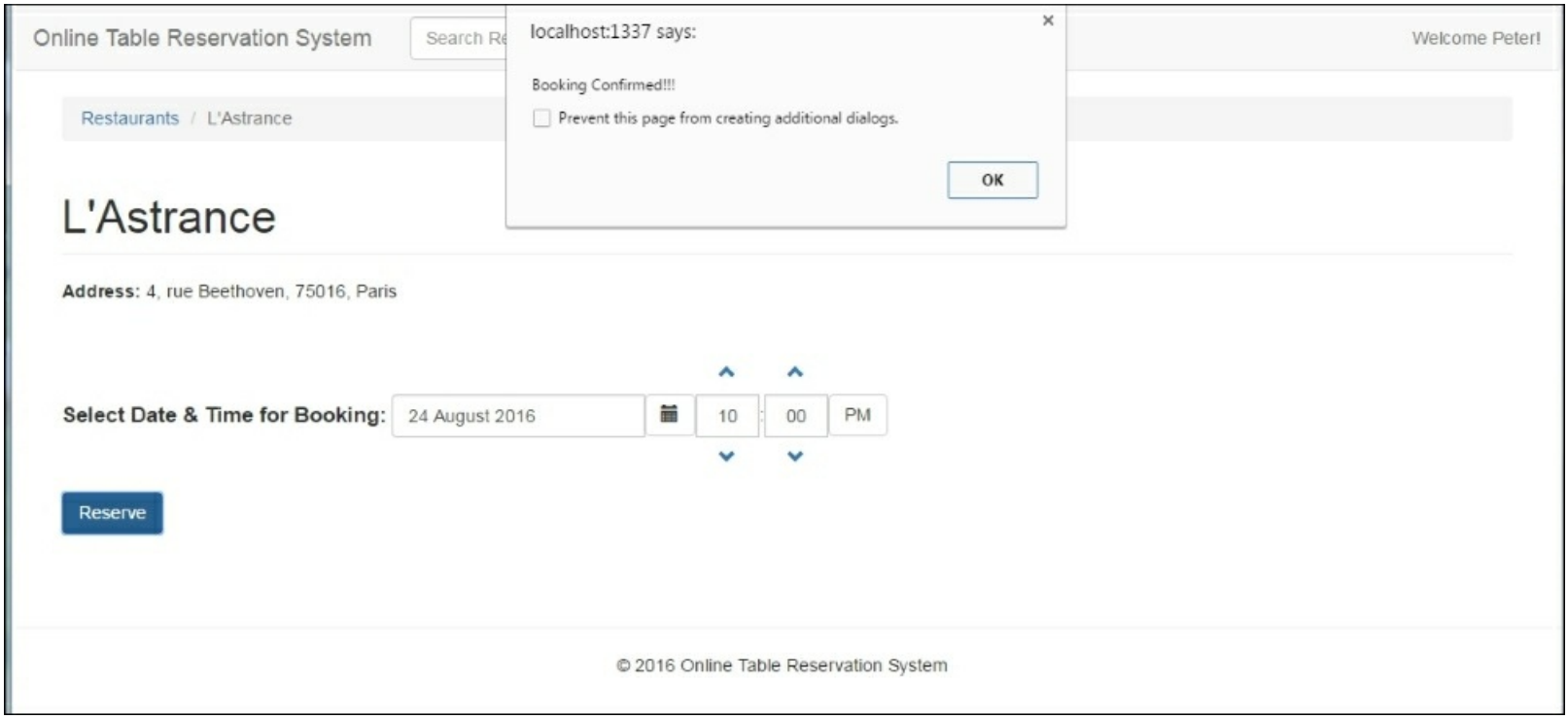
```
angular.module('otrsApp.login', [
    'ui.router',
    'ngStorage'
])
        .config(function config($stateProvider) {
            $stateProvider.state('login', {
                url: '/login',
                controller: 'LoginCtrl',
                templateUrl: 'login/login.html'
            });
        })
        .controller('LoginCtrl', function ($state, $scope, $rootScope,
$injector) {
            var $sessionStorage = $injector.get('$sessionStorage');
            if ($sessionStorage.currentUser) {
                $state.go($rootScope.fromState.name,
$rootScope.fromStateParams);
            }
            var controller = this;
```

```
            var log = $injector.get('$log');
            var http = $injector.get('$http');

            $scope.$on('$destroy', function destroyed() {
                log.debug('LoginCtrl destroyed');
                controller = null;
                $scope = null;
            });
            this.cancel = function () {
                $scope.$dismiss;
                $state.go('restaurants');
            }
            console.log("Current --> " + $state.current);
            this.submit = function (username, password) {
                $rootScope.currentUser = username;
                $sessionStorage.currentUser = username;
                if ($rootScope.fromState.name) {
                    $state.go($rootScope.fromState.name,
$rootScope.fromStateParams);
                } else {
                    $state.go("restaurants");
                }
            };
        });
```

Reservation confirmation

Once the user is logged in and has clicked on the **Reservation** button, the restaurant controller shows the alert box with confirmation as shown in the following screenshot.



Restaurants detail page with reservation confirmation

Setting up the web app

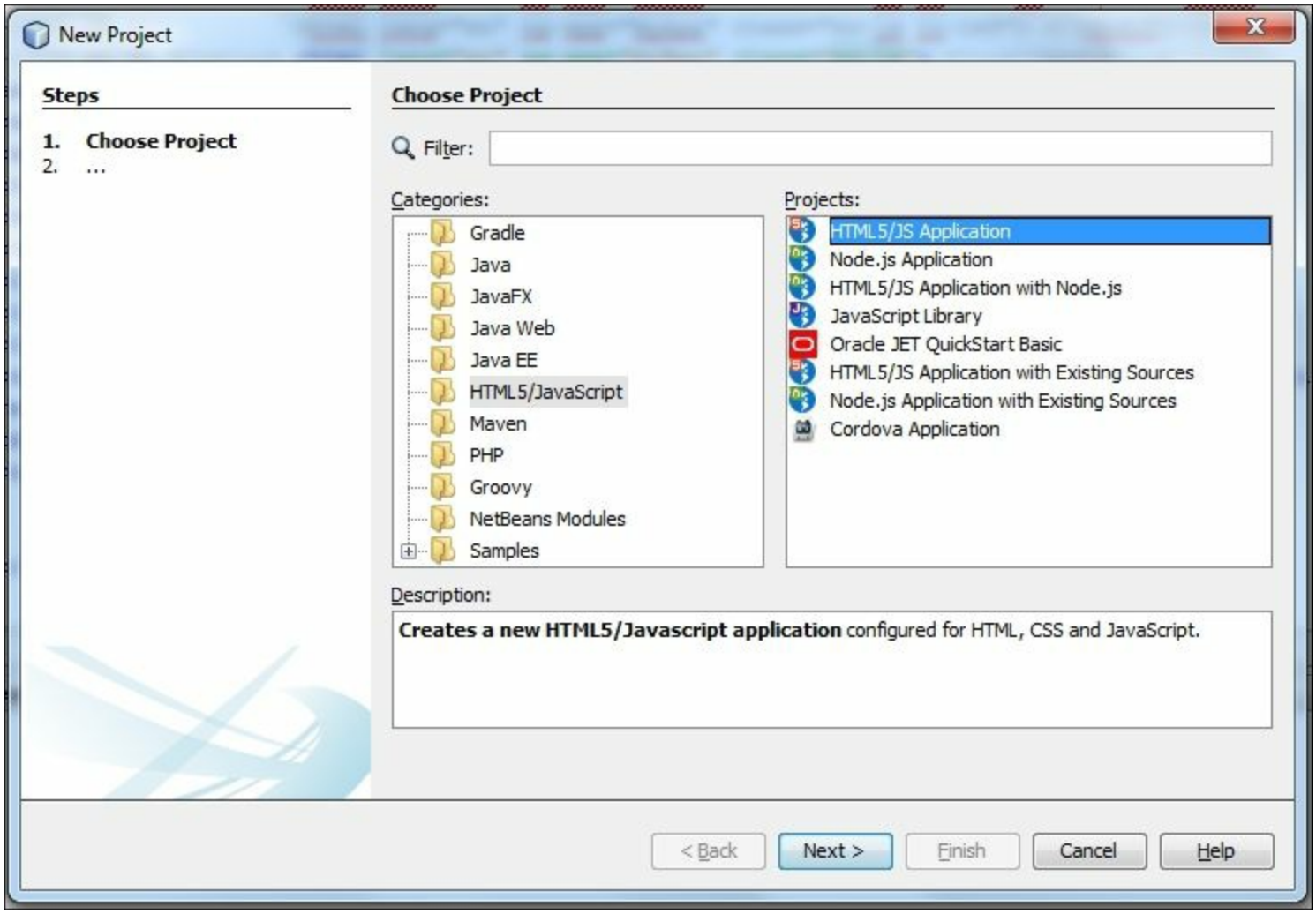
As we are planning to use the latest technology stack for our UI app development, we will use the Node.js and **npm** (**Node.js package manager**) that provides the open-source runtime environment for developing the server side JavaScript web application.

Tip

I would recommend to go through this section once. It will introduce you to JavaScript build tooling and stack. However, you can skip if you know the JavaScript build tools or do not want to explore them.

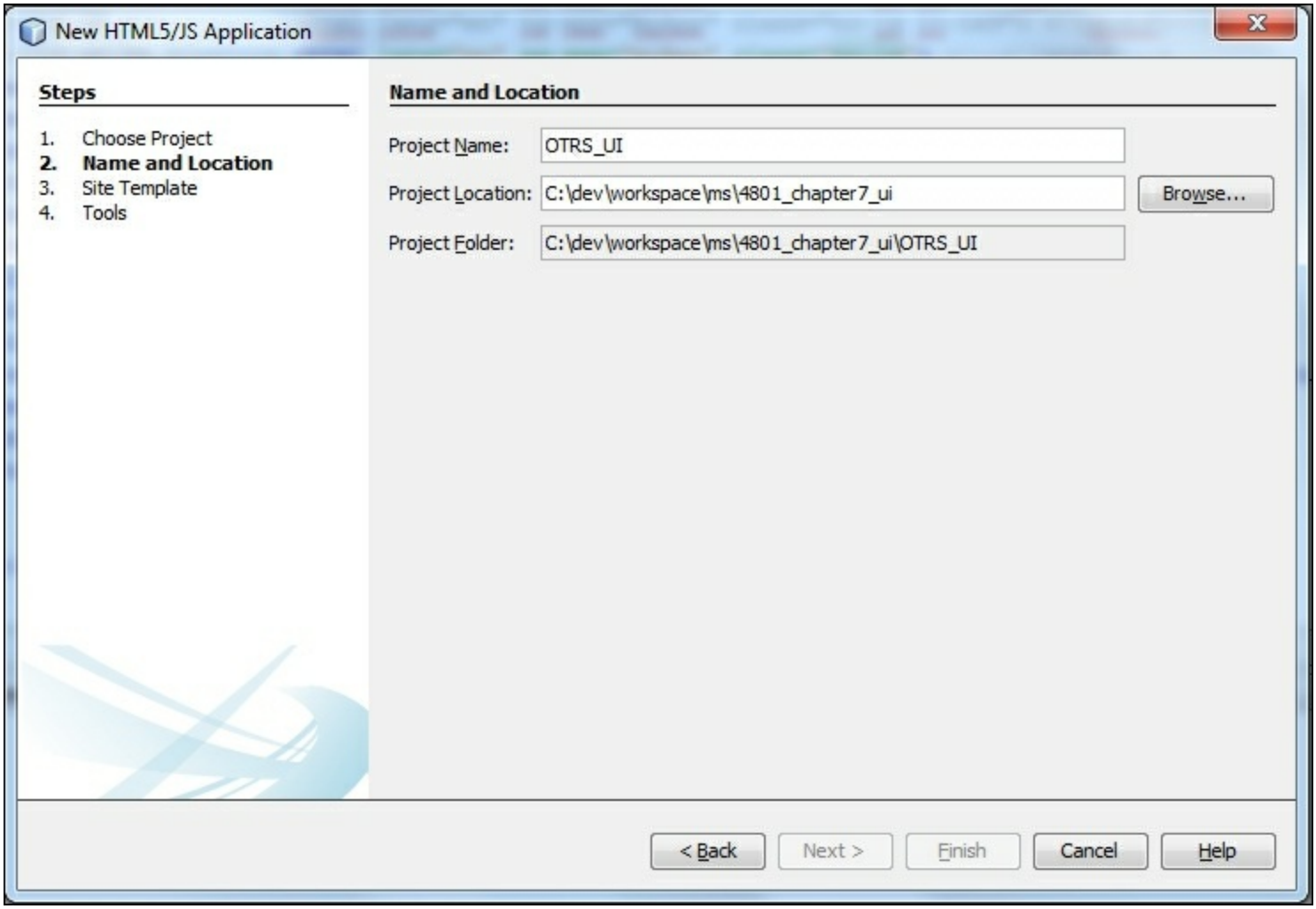
Node.js is built on Chrome's V8 JavaScript engine and uses an event-driven, non-blocking I/O, which makes it lightweight and efficient. The default package manager of Node.js, npm, is the largest ecosystem of open source libraries. It allows installing node programs and makes it easier to specify and link dependencies.

1. First we need to install npm if it's not already installed. It is a prerequisite. You can check the link at https://docs.npmjs.com/getting-started/installing-node to install npm.
2. To check if npm is set up correctly execute the npm `-v` command on CLI. It should return the installed npm version in the output. We can switch to NetBeans for creating a new AngularJS JS HTML 5 project in NetBeans. At the time of writing this chapter, I have used NetBeans 8.1.
3. Navigate to **File** | **New Project**. A new project dialog should appear. Select the **HTML5/JavaScript** under the **Categories** list and **HTML5/JS Application** under the **Projects** options as shown in the following screenshot:



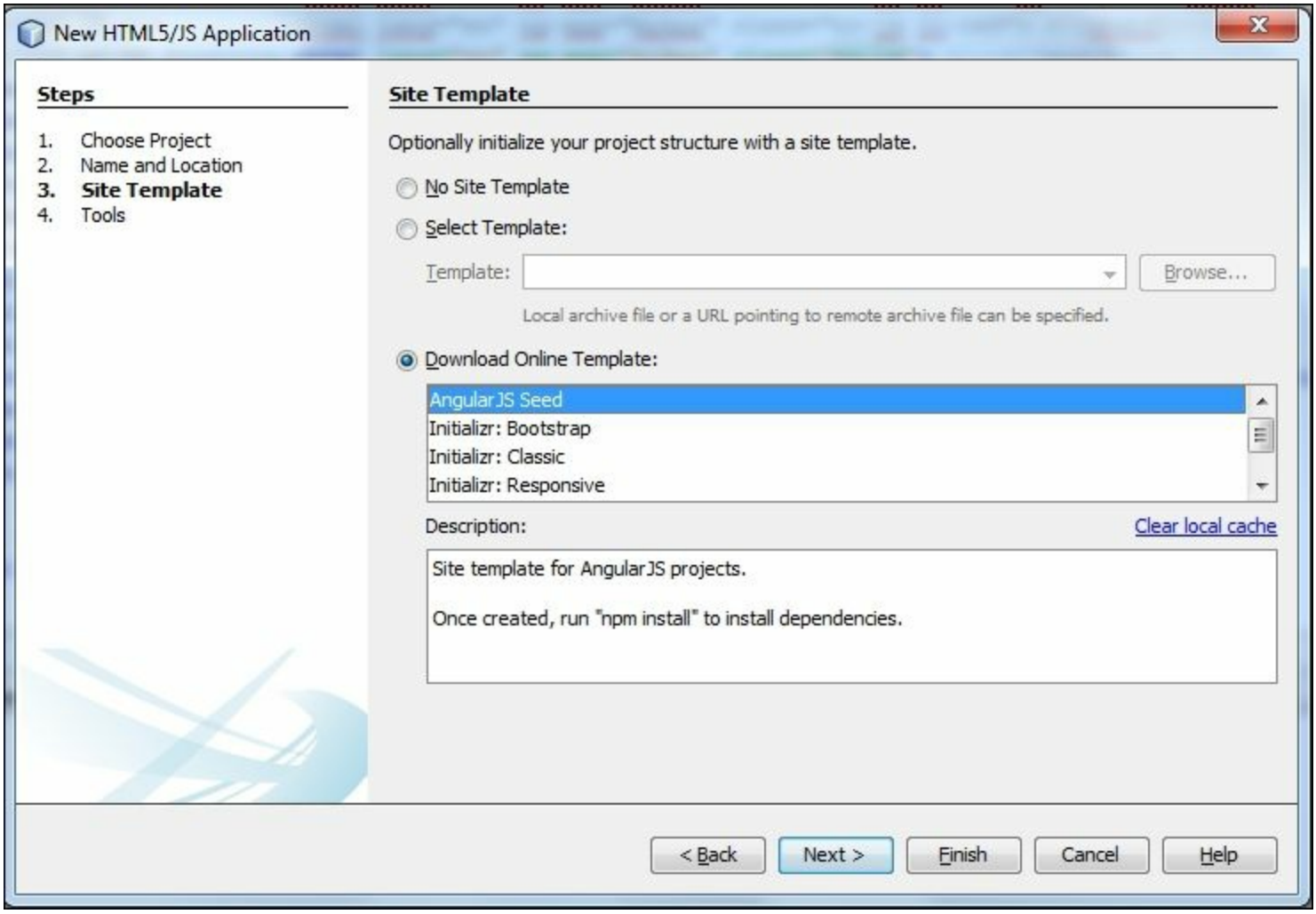
NetBeans – New HTML5/JavaScript project

4. Click on the **Next** button. Then feed the **Project Name**, **Project Location**, and **Project Folder** on the **Name and Location** dialog and click on the **Next** button.



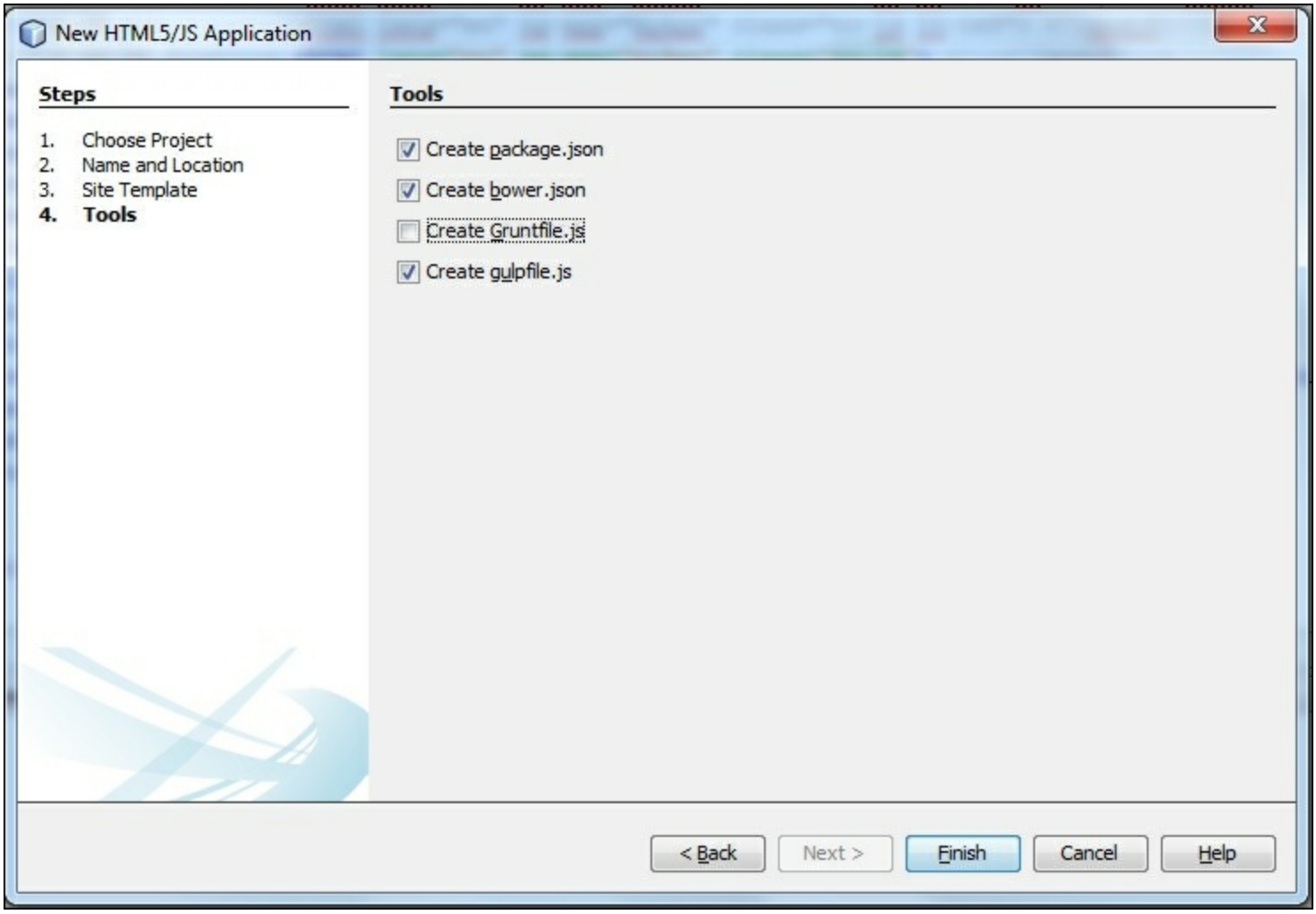
NetBeans New Project – Name and Location

5. On the **Site Template** dialog, select the **AngularJS Seed** item under the **Download Online Template:** option and click on the **Next** button. The AngularJS Seed project is available at https://github.com/angular/angular-seed:



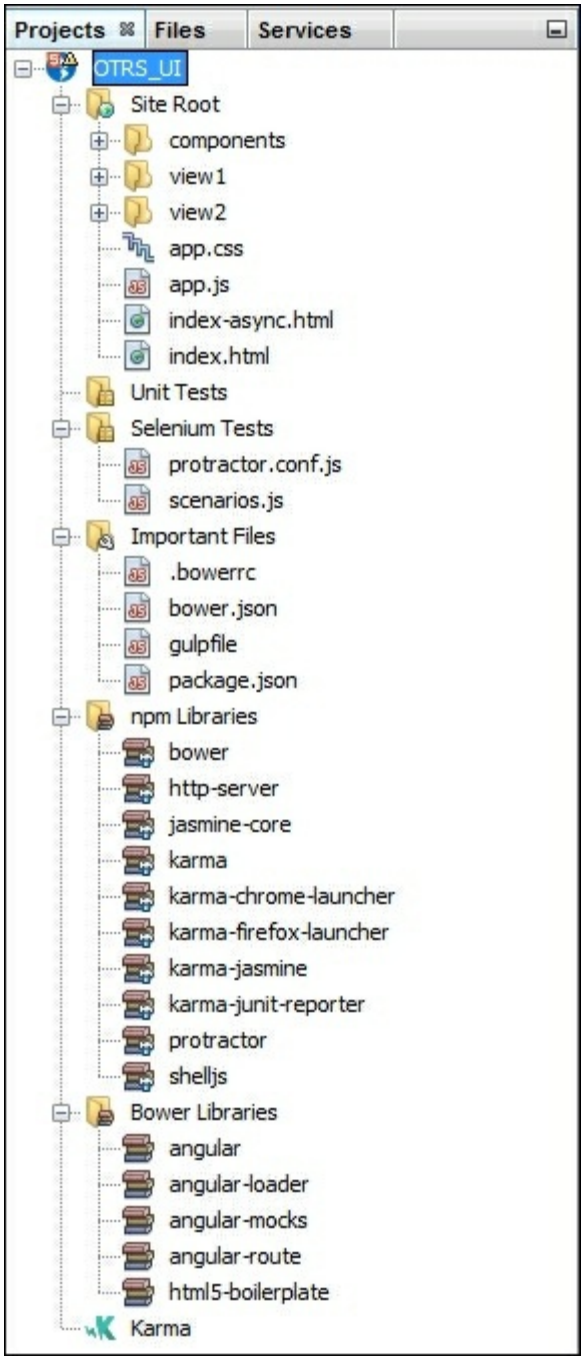
NetBeans New Project – Site Template

6. On the **Tools** dialog, select **Create package.json**, **Create bower.json**, and **Create gulpfile.js**. We'll use gulp as our build tool. Gulp and Grunt are two of the most popular build framework for JS. As a Java programmer, you can correlate these tools to ANT. Both are awesome in their own way. If you want, you can also use `Gruntfile.js` as a build tool.



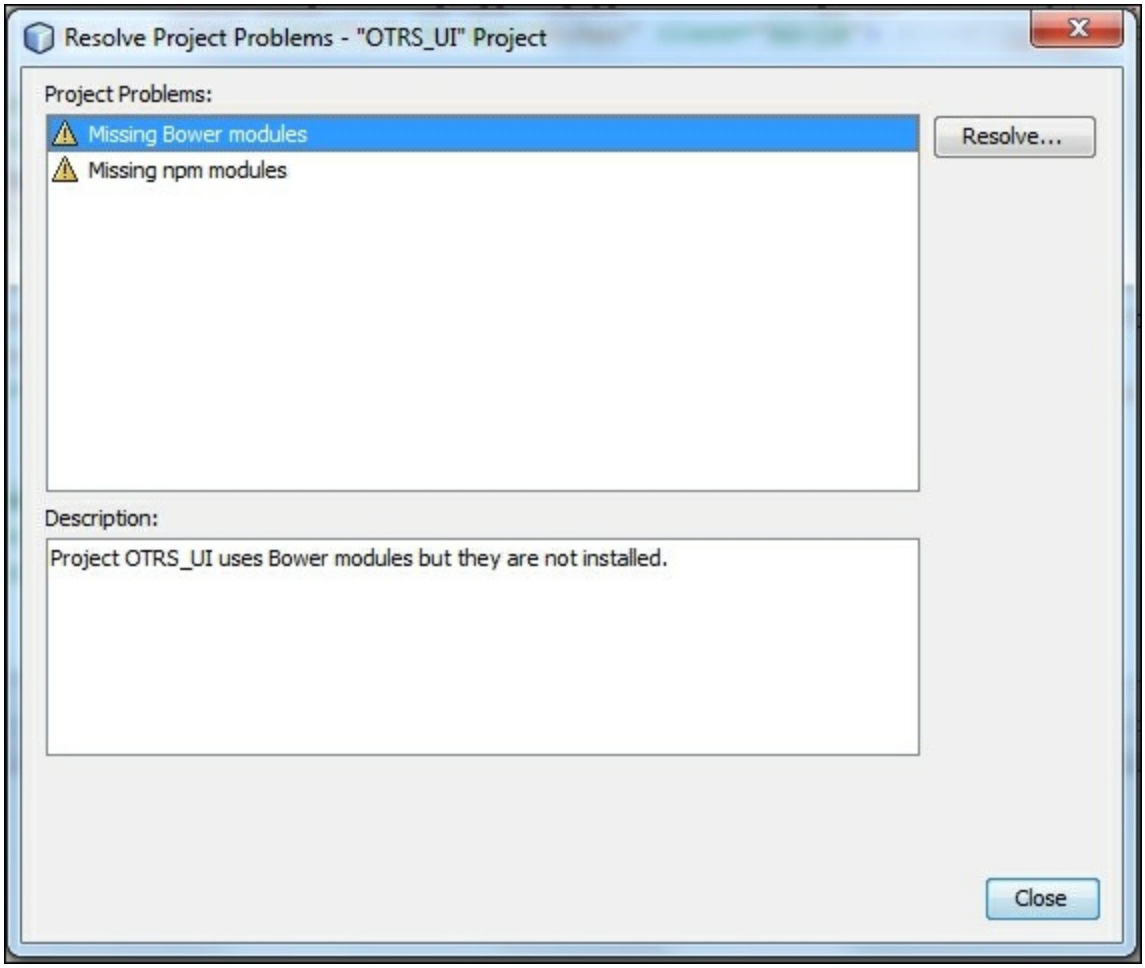
Netbeans New Project - Tools

7. Now, once you click on **Finish**, you can see the HTML5/JS Application directories and files. The directory structure will look like the following screenshot:



AngularJS seed directory structure

8. You will also see an exclamation mark in your project if all the required dependencies are not configured properly. You can resolve project problems by right clicking the project and then selecting the **Resolve Project Problems** option.



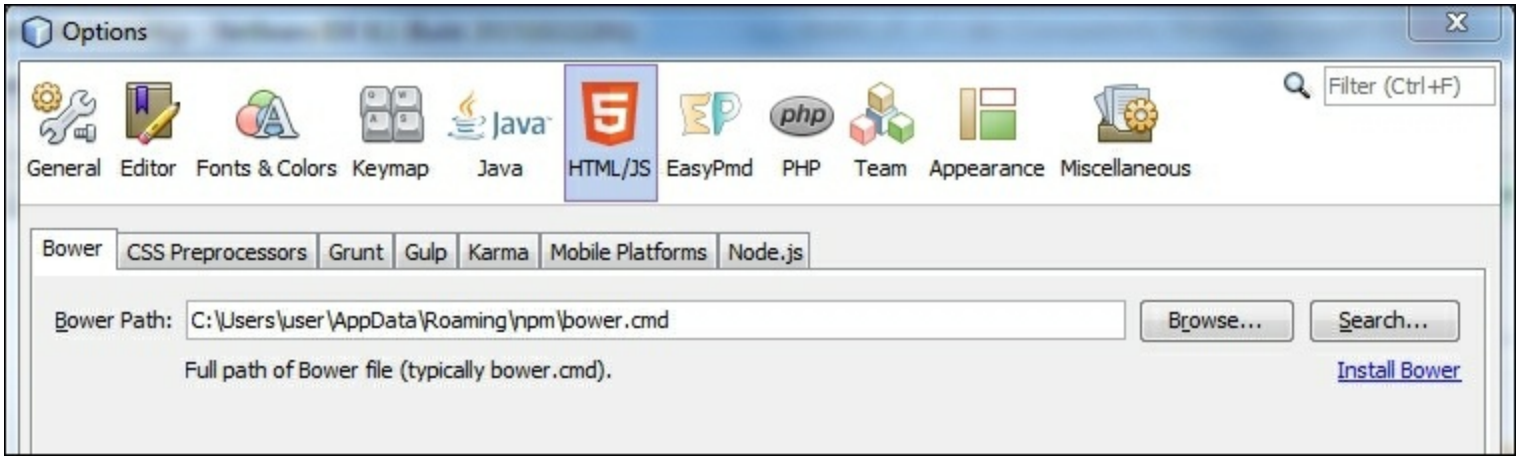
Resolve Project Problems dialog

9. Ideally, NetBeans resolves project problems if you click on the **Resolve…** button.
10. You can also resolve a few of the problems by giving the correct path for some of the JS modules like bower, gulp, and node:
    - **Bower**: Required to manage the JavaScript libraries for the OTRS app
    - **Gulp**: A task runner, required for building our projects like ANT
    - **Node**: For executing our server side OTRS app

    ## Tip

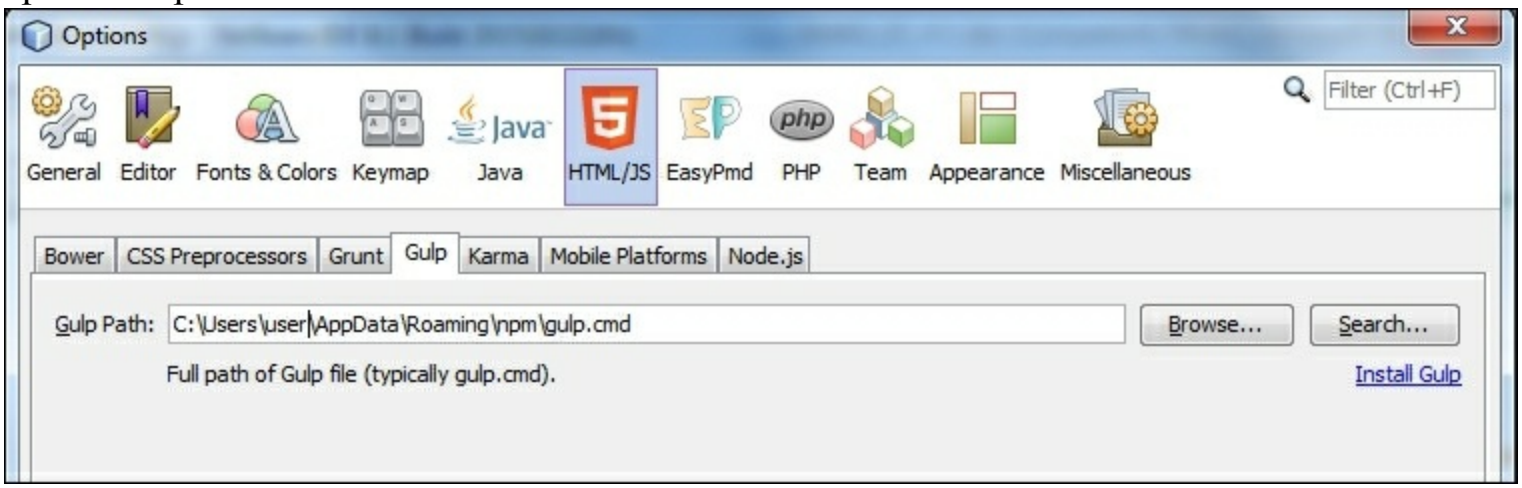
    Bower is a dependencies management tool that works like npm. npm is used for installing the Node.js modules, whereas bower is used for managing your web application's libraries/components.

11. Click on the **Tools** menu and select **Options**. Now, set the path of bower, gulp, and node.js as shown in the HTML/JS tools (top bar icon) in the following screenshot. For setting up the bower path click on the **Bower** tab as shown in the following screenshot and update the path:



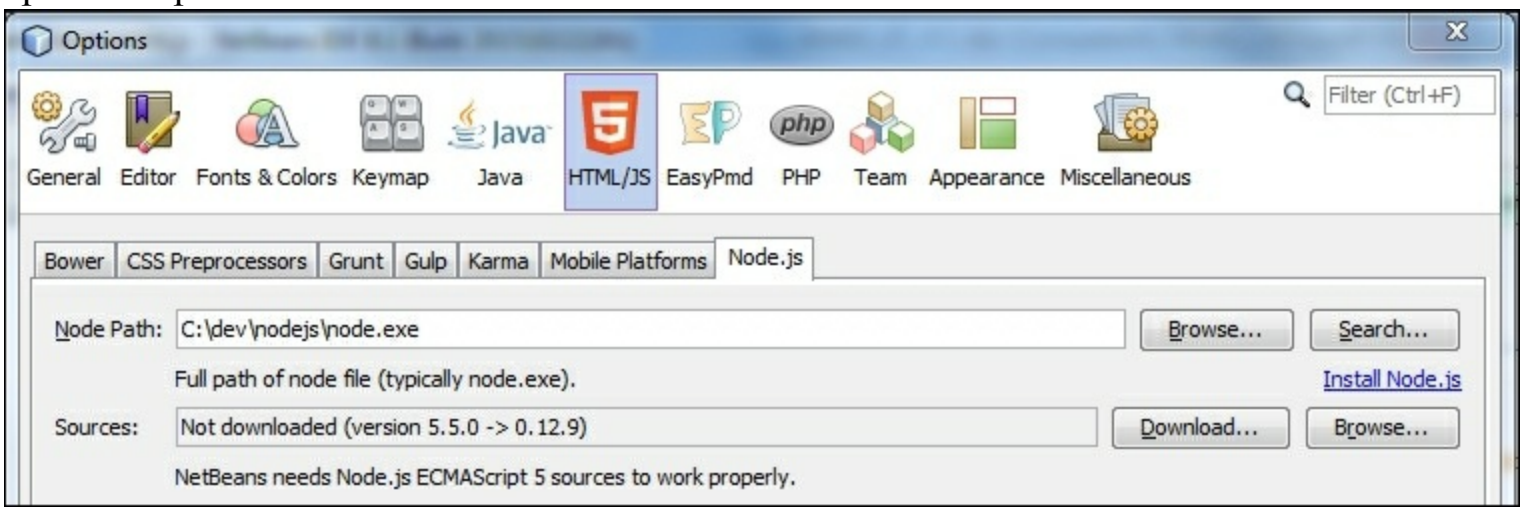
Setting Bower path

12. For setting up the **Gulp Path** click on the **Gulp** tab as shown in the following screenshot and update the path:



Setting Gulp path

13. For setting up the **Node Path** click on the **Node.js** tab as shown in the following screenshot and update the path:



14. Once this is done, **package.json** will look like the following. We have modified the values for a few of the entries like name, descriptions, dependencies, and so on:

```
{
    "name": "otrs-ui",
    "private": true,
    "version": "1.0.0",
    "description": "Online Table Reservation System",
    "main": "index.js",
    "license": "MIT",
    "dependencies": {
        "coffee-script": "^1.10.0",
        "gulp-AngularJS-templatecache": "^1.8.0",
        "del": "^1.1.1",
        "gulp-connect": "^3.1.0",
        "gulp-file-include": "^0.13.7",
        "gulp-sass": "^2.2.0",
        "gulp-util": "^3.0.7",
        "run-sequence": "^1.1.5"
    },
    "devDependencies": {
        "coffee-script": "*",
        "gulp-sass": "*",
        "bower": "^1.3.1",
        "http-server": "^0.6.1",
        "jasmine-core": "^2.3.4",
        "karma": "~0.12",
        "karma-chrome-launcher": "^0.1.12",
        "karma-firefox-launcher": "^0.1.6",
        "karma-jasmine": "^0.3.5",
        "karma-junit-reporter": "^0.2.2",
        "protractor": "^2.1.0",
        "shelljs": "^0.2.6"
    },
    "scripts": {
        "postinstall": "bower install",
        "prestart": "npm install",
        "start": "http-server -a localhost -p 8000 -c-1",
        "pretest": "npm install",
        "test": "karma start karma.conf.js",
        "test-single-run": "karma start karma.conf.js  --single-run",
        "preupdate-webdriver": "npm install",
        "update-webdriver": "webdriver-manager update",
        "preprotractor": "npm run update-webdriver",
        "protractor": "protractor e2e-tests/protractor.conf.js",
        "update-index-async": "node -e \"require('shelljs/global'); sed('-
i', /\\/\\/@@NG_LOADER_START@@[\\s\\S]*\\/\\/@@NG_LOADER_END@@/,
'//@@NG_LOADER_START@@\\n' + sed(/sourceMappingURL=AngularJS-
loader.min.js.map/,'sourceMappingURL=bower_components/AngularJS-
loader/AngularJS-loader.min.js.map','app/bower_components/AngularJS-
loader/AngularJS-loader.min.js') + '\\n//@@NG_LOADER_END@@', 'app/index-
```

```
async.html');\""
    }
}
```

15. Then, we'll update the `bower.json` as shown in the following:

```
{
    "name": "OTRS-UI",
    "description": "OTRS-UI",
    "version": "0.0.1",
    "license": "MIT",
    "private": true,
    "dependencies": {
        "AngularJS": "~1.5.0",
        "AngularJS-ui-router": "~0.2.18",
        "AngularJS-mocks": "~1.5.0",
        "AngularJS-bootstrap": "~1.2.1",
        "AngularJS-touch": "~1.5.0",
        "bootstrap-sass-official": "~3.3.6",
        "AngularJS-route": "~1.5.0",
        "AngularJS-loader": "~1.5.0",
        "ngstorage": "^0.3.10",
        "AngularJS-resource": "^1.5.0",
        "html5-boilerplate": "~5.2.0"
    }
}
```

16. Next, we'll modify the `.bowerrc` file as shown in the following to specify the directory where bower will store the components defined in `bower.json`. We'll store the bower component under the app directory.

```
{
  "directory": "app/bower_components"
}
```

17. Next, we'll set up the `gulpfile.js`. We'll use `CoffeeScript` to define the gulp tasks. Therefore, we will just define the `CoffeeScript` in `gulpfile.js` and the actual task will be defined in `gulpfile.coffee`. Let's see the content of `gulpfile.js`:

```
require('coffee-script/register');
require('./gulpfile.coffee');
```

18. In this step, we'll define the gulp configuration. We are using `CoffeeScript` to define the gulp file. The name of the `gulp` file written in `CoffeeScript` is `gulpfile.coffee`. The default task is defined as `default_sequence`:

```
default_sequence = ['connect', 'build', 'watch']
```

    As per the defined default sequence task, first it will connect to the server, then build the web app, and keep a watch on the changes. The watch will help to render changes we make in the code and will be displayed immediately on the UI.

    The most important parts in this script are `connect` and `watch`. Others are self-explanatory.

- `gulp-connect`: This is a gulp plugin to run the web server. It also allows live reload.
- `gulp-watch`: This is a file watcher that uses chokidar and emits vinyl objects (objects describe the file – its path and content). In simple words, we can say that `gulp-watch` watches files for changes and triggers tasks.

gulpfile.coffee:

```
gulp          = require('gulp')
gutil         = require('gulp-util')
del           = require('del');
clean         = require('gulp-clean')
connect       = require('gulp-connect')
fileinclude   = require('gulp-file-include')
runSequence   = require('run-sequence')
templateCache = require('gulp-AngularJS-templatecache')
sass          = require('gulp-sass')

paths =
  scripts:
    src: ['app/src/scripts/**/*.js']
    dest: 'public/scripts'
  scripts2:
    src: ['app/src/views/**/*.js']
    dest: 'public/scripts'
  styles:
    src: ['app/src/styles/**/*.scss']
    dest: 'public/styles'
  fonts:
    src: ['app/src/fonts/**/*']
    dest: 'public/fonts'
  images:
    src: ['app/src/images/**/*']
    dest: 'public/images'
  templates:
    src: ['app/src/views/**/*.html']
    dest: 'public/scripts'
  html:
    src: ['app/src/*.html']
    dest: 'public'
  bower:
    src: ['app/bower_components/**/*']
    dest: 'public/bower_components'

#copy bower modules to public directory
gulp.task 'bower', ->
  gulp.src(paths.bower.src)
  .pipe gulp.dest(paths.bower.dest)
  .pipe connect.reload()

#copy scripts to public directory
gulp.task 'scripts', ->
  gulp.src(paths.scripts.src)
  .pipe gulp.dest(paths.scripts.dest)
```

```
  .pipe connect.reload()

#copy scripts2 to public directory
gulp.task 'scripts2', ->
  gulp.src(paths.scripts2.src)
  .pipe gulp.dest(paths.scripts2.dest)
  .pipe connect.reload()

#copy styles to public directory
gulp.task 'styles', ->
  gulp.src(paths.styles.src)
  .pipe sass()
  .pipe gulp.dest(paths.styles.dest)
  .pipe connect.reload()

#copy images to public directory
gulp.task 'images', ->
  gulp.src(paths.images.src)
  .pipe gulp.dest(paths.images.dest)
  .pipe connect.reload()

#copy fonts to public directory
gulp.task 'fonts', ->
  gulp.src(paths.fonts.src)
  .pipe gulp.dest(paths.fonts.dest)
  .pipe connect.reload()

#copy html to public directory
gulp.task 'html', ->
  gulp.src(paths.html.src)
  .pipe gulp.dest(paths.html.dest)
  .pipe connect.reload()

#compile AngularJS template in a single js file
gulp.task 'templates', ->
  gulp.src(paths.templates.src)
  .pipe(templateCache({standalone: true}))
  .pipe(gulp.dest(paths.templates.dest))

#delete contents from public directory
gulp.task 'clean', (callback) ->
  del ['./public/**/*'], callback;

#Gulp Connect task, deploys the public directory
gulp.task 'connect', ->
  connect.server
    root: ['./public']
    port: 1337
    livereload: true

gulp.task 'watch', ->
  gulp.watch paths.scripts.src, ['scripts']
  gulp.watch paths.scripts2.src, ['scripts2']
  gulp.watch paths.styles.src, ['styles']
```

```
  gulp.watch paths.fonts.src, ['fonts']
  gulp.watch paths.html.src, ['html']
  gulp.watch paths.images.src, ['images']
  gulp.watch paths.templates.src, ['templates']

gulp.task 'build', ['bower', 'scripts', 'scripts2', 'styles', 'fonts',
'images', 'templates', 'html']

default_sequence = ['connect', 'build', 'watch']

gulp.task 'default', default_sequence

gutil.log 'Server started and waiting for changes'
```

19. Once we are ready with the preceding changes, we will install the gulp using the following command:

    `npm install --no-optional gulp`

20. Also, we'll install the other gulp libraries like `gulp-clean`, `gulp-connect`, and so on using the following command:

    `npm install --save --no-optional gulp-util gulp-clean gulp-connect gulp-file-include run-sequence gulp-AngularJS-templatecache gulp-sass`

21. Now, we can install the bower dependencies defined in the `bower.json` file using the following command:

    `bower install --save`



Sample output - bower install --save

22. This is the last step in the setup. Here, we will confirm that the directory structure should look like the following. We'll keep the src and published artifacts (in `./public` directory) as separate directories. Therefore, the following directory structure is different from the default AngularJS seed project:

```
+---app
|   +---bower_components
|   |   +---AngularJS
|   |   +---AngularJS-bootstrap
|   |   +---AngularJS-loader
|   |   +---AngularJS-mocks
|   |   +---AngularJS-resource
|   |   +---AngularJS-route
|   |   +---AngularJS-touch
|   |   +---AngularJS-ui-router
|   |   +---bootstrap-sass-official
|   |   +---html5-boilerplate
|   |   +---jquery
```

```
|   |   \---ngstorage
|   +---components
|   |   \---version
|   +---node_modules
|   +---public
|   |   \---css
|   \---src
|       +---scripts
|       +---styles
|       +---views
+---e2e-tests
+---nbproject
|   \---private
+---node_modules
+---public
|   +---bower_components
|   +---scripts
|   +---styles
\---test
```

References to some good reads:

- *AngularJS by Example*, *Packt Publishing* (https://www.packtpub.com/web-development/angularjs-example)
- *Angular Seed Project* (https://github.com/angular/angular-seed)
- *Angular UI* (https://angular-ui.github.io/bootstrap/)
- *Gulp* (http://gulpjs.com/)

Summary

In this chapter, we have learned the new dynamic web application development. It has changed completely over the years. The web application frontend is completely developed in pure HTML and JavaScript instead of using any server side technologies like JSP, servlets, ASP, and so on. UI app development with JavaScript now has its own development environment like npm, bower, and so on. We have explored the AngularJS framework to develop our web app. It made things easier by providing inbuilt features and support to bootstrap and the `$http` service that deals with the AJAX calls.

I hope you have grasped the UI development overview and the way modern applications are developed and integrated with server side microservices. In the next chapter, we will learn the best practices and common principals of microservice design. The chapter will provide details about microservices development using industry practices and examples. It will also contain examples of where microservices implementation goes wrong and how you can avoid such problems.

Chapter 8. Best Practices and Common Principles

After all the hard work put in by you towards gaining the experience of developing the microservice sample project, you must be wondering how to avoid common mistakes and improve the overall process of developing microservices-based products and services. We can follow these principles or guidelines to simplify the process of developing the microservices and avoid/reduce the potential limitations. We will focus on these key concepts in this chapter.

This chapter is spread across the following three sections:

- Overview and mindset
- Best practices and principals
- Microservices frameworks and tools

Overview and mindset

You can implement microservices-based design on both new and existing products and services. Contrary to the belief that it is easier to develop and design a new system from scratch rather than making changes to an existing one that is already live, each approach has its own respective challenges and advantages.

For example, since there is no existing system design for a new product or service, you have freedom and flexibility to design the system without giving any thought to its impact. However, you don't have the clarity on both functional and system requirements for a new system, as these mature and take shape over time. On the other hand, for mature products and services, you have detailed knowledge and information of the functional and system requirements. Nevertheless, you have a challenge to mitigate the risk of impact that design change brings to the table. Therefore, when it comes to updating a production system from monolithic to microservices, you will need to plan better than if you were building a system from scratch.

Experienced and successful software design experts and architects always evaluate the pros and cons and take a cautious approach to making any change to existing live systems. One should not make changes to existing live system design simply because it may be cool or trendy. Therefore, if you would like to update the design of your existing production system to microservices, you need to evaluate all the pros and cons before making this call.

I believe that monolithic systems provide a great platform to upgrade to a successful microservices-based design. Obviously, we are not discussing cost here. You have ample knowledge of the existing system and functionality, which enables you to divide the existing system and build microservices based on functionalities and how those would interact with each other. Also, if your monolithic product is already modularized in some way, then directly transforming microservices by exposing an API instead of **Application Binary Interface** (**ABI**) is possibly the easiest way of achieving a microservice architecture. A successful microservices-based system is more dependent on microservices and their interaction protocol rather than anything else.

Having said that, it does not mean that you cannot have a successful microservices-based system if you are starting from scratch. However, it is recommended to start a new project based on monolithic design that gives you perspective and understanding of the system and functionality. It allows you to find bottlenecks quickly and guides you to identify any potential feature that can be developed using microservices. Here, we have not discussed the size of the project, which is another important factor. We'll discuss this in the next section.

In today's cloud age and agile development world, it takes an hour between making any change and the change going live. In today's competitive environment, every organization would like to have an edge for quickly delivering features to the user. Continuous development, integration, and deployment are part of the production delivery process, a completely automatic process.

It makes more sense if you are offering cloud-based products or services. Then, a microservices-based system enables the team to respond with agility to fix any issue or provide a new feature to the user.

Therefore, you need to evaluate all pros and cons before you make a call for starting a new microservices-based project from scratch or planning to upgrade the design of an existing monolithic system to a microservices-based system. You have to listen to and understand the different ideas and perspectives shared across your team, and you need to take a cautious approach.

Finally, I would like to share the importance of having better processes and an efficient system in place for a successful production system. Having a microservices-based system does not guarantee a successful production system, and monolithic application does not mean you cannot have a successful production system in today's age. Netflix, a microservices-based cloud video rental service, and Etsy, a monolithic e-commerce platform, are both examples of successful live production systems (see an interesting Twitter discussion link in the *Reference* section later in the chapter). Therefore, processes and agility are also key to a successful production system.

Best practices and principals

As we have learned from the first chapter, microservices are a lightweight style of implementing **Service Oriented Architecture** (**SOA**). On top of that, microservices are not strictly defined, which gives you the flexibility of developing microservices the way you want and according to need. At the same time, you need to make sure that you follow a few of the standard practices and principals to make your job easier and implement microservices-based architecture successfully.

Nanoservice (not recommended), size, and monolithic

Each microservice in your project should be small in size and perform one functionality or feature (for example, user management), independently enough to perform the function on its own.

The following two quotes from Mike Gancarz (a member that designed the X windows system), which defines one of the paramount precepts of UNIX philosophy, suits the microservice paradigm as well:

> *"Small is beautiful."*
>
> *"Make each program do one thing well."*

Now, how to define the size, in today's age, when you have a framework (for example Finangle) that reduces the **lines of code** (**LOC**)? In addition, many modern languages, such as Python and Erlang, are less verbose. This makes it difficult to decide whether you want to make this code microservice or not.

Apparently, you may implement a microservice for a small number of LOC, that is actually not a microservice but a nanoservice.

Arnon Rotem-Gal-Oz defined nanoservice as follows:

> *"Nanoservice is an antipattern where a service is too fine-grained. A nanoservice is a service whose overhead (communications, maintenance, and so on) outweighs its utility."*

Therefore, it always makes sense to design microservices based on functionality. Domain driven design makes it easier to define functionality at a domain level.

As discussed previously, the size of your project is a key factor when deciding whether to implement microservices or determining the number of microservices you want to have for your project. In a simple and small project, it makes sense to use monolithic architecture. For example, based on the domain design that we learned in Chapter 3, *Domain-Driven Design* you would get a clear understanding of your functional requirements and it makes facts available to draw the boundaries between various functionalities or features. For example, in the sample project (OTRS) we have implemented, it is very easy to develop the same project using monolithic design; provided you don't want to expose the APIs to the customer, or you don't want to use it as SaaS, or there are plenty of similar parameters that you want to evaluate before making a call.

You can migrate the monolithic project to microservices design later, when the need arises. Therefore, it is important that you should develop the monolithic project in modular fashion and have the loose coupling at every level and layer, and ensure there are predefined contact points and boundaries between different functionalities and features. In addition, your data source, such as DB, should be designed accordingly. Even if you are not planning to migrate to a microservices-based

system, it would make bug fixes and enhancement easier to implement.

Paying attention to the previous points will mitigate any possible difficulties you may encounter when you migrate to microservices.

Generally, large or complex projects should be developed using microservices-based architecture, due to the many advantages it provides, as discussed in previous chapters.

Even I recommended developing your initial project as monolithic; once you gain a better understanding of project functionalities and project complexity, then you can migrate it to microservices. Ideally, a developed initial prototype should give you the functional boundaries that will enable you to make the right choice.

Continuous integration and deployment

You must have a continuous integration and deployment process in place. It gives you an edge to deliver changes faster and detect bugs early. Therefore, each service should have its own integration and deployment process. In addition, it must be automated. There are many tools available, such as Teamcity, Jenkins, and so on, that are used widely. It helps you to automate the build process—which catches build failure early, especially when you integrate your changes with mainline.

You can also integrate your tests with each automated integration and deployment process. **Integration Testing** tests the interactions of different parts of the system, like between two interfaces (API provider and consumer), or among different components or modules in a system, such as between DAO and database, and so on. Integration testing is important as it tests the interfaces between the modules. Individual modules are first tested in isolation. Then, integration testing is performed to check the combined behavior and validate that requirements are implemented correctly. Therefore, in microservices, integration testing is a key tool to validate the APIs. We will cover more about it in the next section.

Finally, you can see the update mainline changes on your DIT machine where this process deploys the build.

The process does not end here; you can make a container, like docker and hand it over to your WebOps team, or have a separate process that delivers to a configured location or deploy to a WebOps stage environment. From here it could be deployed directly to your production system once approved by the designated authority.

System/end-to-end test automation

Testing is a very important part of any product and service delivery. You do not want to deliver buggy applications to customers. Earlier, at the time when the waterfall model was popular, an organization used to take one to six months or more for the testing stage before delivering to the customer. In recent years, after agile process became popular, more emphasis is given to automation. Similar to prior point testing, automation is also mandatory.

Whether you follow **Test Driven Development** (**TDD**) or not, we must have system or end-to-end test automation in place. It's very important to test your business scenarios and that is also the case with end-to-end testing that may start from your REST call to database checks, or from UI app to database checks.

Also, it is important to test your APIs if you have public APIs.

Doing this makes sure that any change does not break any of the functionality and ensures seamless, bug-free production delivery. As discussed in the last section, each module is tested in isolation using unit testing to check everything is working as expected, then integration testing is performed among different modules to check the expected combined behavior and validate the requirements, whether implemented correctly or not. After integration tests, functional tests are executed that validate the functional and feature requirements.

So, if unit testing makes sure individual modules are working fine in isolation, integration testing makes sure that interaction among different modules works as expected. If unit tests are working fine, it implies that the chances of integration test failure is greatly reduced. Similarly, integration testing ensures that functional testing is likely to be successful.

Note

It is presumed that one always keeps all types of tests updated, whether these are unit-level tests or end-to-end test scenarios.

Self-monitoring and logging

Microservices should provide service information about itself and the state of the various resources it depends on. Service information represents the statistics such as the average, minimum, and maximum time to process a request, the number of successful and failed requests, being able to track a request, memory usage, and so on.

Adrian Cockcroft highlighted a few practices, which are very important for monitoring the microservices, in **Glue Conference** (**Glue Con**) 2015. Most of them are valid for any monitoring system:

- Spend more time working on code that analyzes the meaning of metrics than code that collects, moves, stores, and displays metrics.

  This helps to not only increase the productivity, but also provides important parameters to fine-tune the microservices and increase the system efficiency. The idea is to develop more analysis tools rather than developing more monitoring tools.
- The metric to display latency needs to be less than the human attention span. That means less than 10 seconds, according to Adrian.
- Validate that your measurement system has enough accuracy and precision. Collect histograms of response time.
- Accurate data makes decision making faster and allows you to fine-tune till precision level. He also suggests that the best graph to show the response time is a histogram.
- Monitoring systems need to be more available and scalable than the systems being monitored.
- The statement says it all: you cannot rely on a system which itself is not stable or available 24/7.
- Optimize for distributed, ephemeral, cloud native, containerized microservices.
- Fit metrics to models to understand relationships.

Monitoring is a key component of microservice architecture. You may have a dozen to thousands of microservices (true for a big enterprise's large project) based on project size. Even for scaling and high availability, organizations create a clustered or load-balanced pool/pod for each microservice, even separate pools for each microservice based on versions. Ultimately, it increases the number of resources you need to monitor, including each microservice instance. In addition, it is important that you should have a process in place so that whenever something goes wrong, you know it immediately, or better, receive a warning notification in advance before something goes wrong. Therefore, effective and efficient monitoring is crucial for building and using the microservice architecture. Netflix uses security monitoring using tools like Netflix Atlas (real-time operational monitoring which processes 1.2 billion metrics), Security Monkey (for monitoring security on AWS-based environments), Scumblr (intelligence gathering tool) and FIDO (for analyzing events and automated incident reporting).

Logging is another important aspect for microservices that should not be ignored. Having effective logging makes all the difference. As there could be 10 or more microservices, managing logging is a huge task.

For our sample project, we have used MDC logging, which is sufficient, in a way, for individual microservice logging. However, we also need logging for an entire system, or central logging. We also need aggregated statistics of logs. There are tools that do the job, like Loggly or Logspout.

Note

A request and generated correlated events gives you an overall view of the request. For tracing of any event and request, it is important to associate the event and request with service ID and request ID respectively. You can also associate the content of the event, such as message, severity, class name, and so on, to service ID.

A separate data store for each microservice

If you remember, the most important characteristics of microservices you can find out about is the way microservices run in isolation from other microservices, most commonly as standalone applications.

Abiding by this rule, it is recommended that you not use the same database, or any other data store across multiple microservices. In large projects, you may have different teams working on the same project, and you want the flexibility to choose the database for each microservice that best suits the microservice.

Now, this also brings some challenges.

For instance, the following is relevant to teams who may be working on different microservices within the same project, if that project shares the same database structure. There is a possibility that a change in one microservice may impact the other microservices model. In such cases, change in one may affect the dependent microservice, so you also need to change the dependent model structure.

To resolve this issue, microservices should be developed based on an API-driven platform. Each microservice would expose its APIs, which could be consumed by the other microservices. Therefore, you also need to develop the APIs, which is required for the integration of different microservices.

Similarly, due to different data stores, actual project data is also spread across multiple data stores and it makes data management more complicated, because the separate storage systems can more easily get out of sync or become inconsistent, and foreign keys can change unexpectedly. To resolve such an issue, you need to use **Master Data Management** (**MDM**) tools. MDM tools operate in the background and fix inconsistencies if they find any. For the OTRS sample example, it might check every database that stores booking request IDs, to verify that the same IDs exist in all of them (in other words, that there aren't any missing or extra IDs in any one database). MDM tools available in the market include Informatica, IBM MDM Advance Edition, Oracle Siebel UCM, Postgres (master streaming replication), mariadb (master/master configuration), and so on.

If none of the existing products suit your requirements, or you are not interested in any proprietary product, then you can write your own. Presently, API-driven development and platform reduce such complexities; therefore, it is important that microservices should be developed along with an API platform.

Transaction boundaries

We have gone through domain driven design concepts in Chapter 3, *Domain-Driven Design*. Please review this if you have not grasped it thoroughly, as it gives you an understanding of the state vertically. Since we are focusing on microservices-based design, the result is that we have a system of systems, where each microservice represents a system. In this environment, finding the state of a whole system at any given point in time is very challenging. If you are familiar with distributed applications, then you may be comfortable in such an environment, with respect to state.

It is very important to have transaction boundaries in place that describe which microservice owns a message at any given time. You need a way or process that can participate in transactions, transacted routes and error handlers, idempotent consumers, and compensating actions. It is not an easy task to ensure transactional behavior across heterogeneous systems, but there are tools available that do the job for you.

For example, Camel has great transactional capabilities that help developers easily create services with transactional behavior.

Microservices frameworks and tools

It is always better not to reinvent the wheel. Therefore, we would like to explore what tools are already available and provide the platform, framework, and features that make microservices development and deployment easier.

Throughout the book, we have used the Spring Cloud extensively, due to the same reason; it provides all the tools and platform required to make microservice development very easy. Spring Cloud uses the Netflix **Open Source Software** (**OSS**). Let us explore Netflix OSS—a complete package.

I have also added a brief overview about how each tool will help to build good microservice architecture.

Netflix Open Source Software (OSS)

Netflix OSS center is the most popular and widely-used open source software for Java-based microservice open source projects. The world's most successful video renting service is dependent on it. Netflix has more than 40 million users and is used across the globe. Netflix is a pure cloud-based solution, developed on microservice-based architecture. You can say that whenever anybody talks about microservices, Netflix is the first name that comes to mind. Let us discuss the wide variety of tools it provides. We have already discussed many of them while developing the sample OTRS app. However, there are a few which we have not explored. Here, we'll cover only the overview of each tool, instead of going into detail. It will give you an overall idea of the practical characteristics of microservices architecture and its use in Cloud.

Build – Nebula

Netflix Nebula is a collection of Gradle plugins that makes your microservice builds easier using Gradle (a Maven-like build tool). For our sample project, we have made use of Maven, therefore we haven't had the opportunity to explore Nebula in this book. However, exploring it would be fun. The most significant Nebula feature for developers is eliminating the boilerplate code in Gradle build files, which allows developers to focus on coding.

Note

Having a good build environment, especially CI/CD (continuous integration and continuous deployment) is a must for microservice development and keeping aligned with agile development. Netflix Nebula makes your build easier and more efficient.

Deployment and delivery – Spinnaker with Aminator

Once your build is ready, you want to move that build to **Amazon Web Services** (**AWS**) EC2. Aminator creates and packages images of builds in the form of **Amazon Machine Image** (**AMI**). Spinnaker then deploys these AMIs to AWS.

Spinnaker is a continuous delivery platform for releasing code changes with high velocity and efficiency. Spinnaker also supports other cloud services, such as Google Computer Engine and Cloud Foundry.

Note

You would like to deploy your latest microservice builds to cloud environments like EC2. Spinnaker and Aminator helps you to do that in an autonomous way.

Service registration and discovery – Eureka

Eureka, as we have explored in this book provides a service that is responsible for microservice registration and discovery. On top of that, Eureka is also used for load-balancing the middle-tier (processes hosting different microservices). Netflix also uses Eureka, along with other tools, like

Cassandra or memcached, to enhance its overall usability.

Note

Service registration and discovery is a must for microservice architecture. Eureka serves this purpose. Please refer to Chapter 4, *Implementing Microservices* for more information about Eureka.

Service communication – Ribbon

Microservice architecture is of no use if there is no inter-process or service communication. The Ribbon application provides this feature. Ribbon works with Eureka for load balancing and with Hystrix for fault tolerance or circuit breaker operations.

Ribbon also supports TCP and UDP protocols, apart from HTTP. It provides these protocol supports in both asynchronous and reactive models. It also provides the caching and batching capabilities.

Note

Since you will have many microservices in your project, you need a way to process information using inter-process or service communication. Netflix provides the Ribbon tool for this purpose.

Circuit breaker – Hystrix

Hystrix tool is for circuit breaker operations, that is, latency and fault tolerance. Therefore, Hystrix stops cascading failures. Hystrix performs the real-time operations for monitoring the services and property changes, and supports concurrency.

Note

Circuit breaker, or fault tolerance, is an important concept for any project, including microservices. Failure of one microservice should not halt your entire system; to prevent this, and provide meaningful information to the customer on failure, is the job of Netflix Hystrix.

Edge (proxy) server – Zuul

Zuul is an edge server or proxy server, and serves the requests of external applications such as UI client, Android/iOS app, or any third-party consumer of APIs offered by the product or service. Conceptually, it is a door to external applications.

Zuul allows dynamic routing and monitoring of requests. It also performs security operations like authentication. It can identify authentication requirements for each resource and reject any request that does not satisfy them.

Note

You need an edge server or API gateway for your microservices. Netflix Zuul provides this feature. Please refer to Chapter 5, *Deployment and Testing* for more information.

Operational monitoring – Atlas

Atlas is an operational monitoring tool that provides near real-time information on dimensional time-series data. It captures operational intelligence that provides a picture of what is currently happening within a system. It features in-memory data storage, allowing it to gather and report very large numbers of metrics very quickly. At present, it processes 1.3 billion metrics for Netflix.

Atlas is a scalable tool. This is why it can now process 1.3 billion metrics, from 1 million metrics a few years back. Atlas not only provides scalability in terms of reading the data, but also aggregating it as a part of graph request.

Atlas uses the Netflix Spectator library for recording dimensional time-series data.

Note

Once you deploy microservices in Cloud environment, you need to have a monitoring system in place to track and monitor all microservices. Netflix Atlas does this job for you

Reliability monitoring service – Simian Army

In Cloud, no single component can guarantee 100% uptime. Therefore, it is a requirement for successful microservice architecture to make the entire system available in case a single cloud component fails. Netflix has developed a tool named Simian Army to avoid system failure. Simian Army keeps a cloud environment safe, secure, and highly available. To achieve high availability and security, it uses various services (Monkeys) in the cloud for generating various kinds of failures, detecting abnormal conditions, and testing the cloud's ability to survive these challenges. It uses the following services (Monkeys), which are taken from the Netflix blog:

- **Chaos Monkey**: Chaos Monkey is a service which identifies groups of systems and randomly terminates one of the systems in a group. The service operates at a controlled time and interval. Chaos Monkey only runs in business hours with the intent that engineers will be alert and able to respond.
- **Janitor Monkey**: Janitor Monkey is a service which runs in the AWS cloud looking for unused resources to clean up. It can be extended to work with other cloud providers and cloud resources. The schedule of service is configurable. Janitor Monkey determines whether a resource should be a cleanup candidate, by applying a set of rules on it. If any of the rules determines that the resource is a cleanup candidate, Janitor Monkey marks the resource and schedules a time to clean it up. For exceptional cases, when you want to keep an unused resource longer, before Janitor Monkey deletes a resource, the owner of the resource will receive a notification a configurable number of days ahead of the cleanup time.
- **Conformity Monkey**: Conformity Monkey is a service which runs in the AWS cloud looking for instances that are not conforming to predefined rules for the best practices. It can be extended to work with other cloud providers and cloud resources. The schedule of service is configurable.

  If any of the rules determines that the instance is not conforming, the monkey sends an e-mail

notification to the owner of the instance. There could be exceptional cases where you want to ignore warnings of a specific conformity rule for some applications.
- **Security Monkey**: Security Monkey monitors policy changes and alerts on insecure configurations in an AWS account. The main purpose of Security Monkey is security, though it also proves a useful tool for tracking down potential problems, as it is essentially a change-tracking system.
- Successful microservice architecture makes sure that your system is always up, and failure of a single cloud component should not fail the entire system. Simian Army uses many services to achieve high availability.

AWS resource monitoring – Edda

In a cloud environment, nothing is static. For example, virtual host instance changes frequently, an IP address could be reused by various applications, or a firewall or related changes may take place.

Edda is a service that keeps track of these dynamic AWS resources. Netflix named it Edda (meaning *a tale of Norse mythology*), as it records the tales of cloud management and deployments. Edda uses the AWS APIs to poll AWS resources and records the results. These records allow you to search and see how the cloud has changed over time. For instance, if any host of the API server is causing any issue, then you need to find out what that host is and which team is responsible for it.

These are the features it offers:

- **Dynamic querying**: Edda provides the REST APIs, and it supports the matrix arguments and provides fields selectors that let you retrieve only the desired data.
- **History/Changes**: Edda maintains the history of all AWS resources. This information helps you when you analyze the causes and impact of outage. Edda can also provide the different view of current and historical information about resources. It stores the information in MongoDB at the time of writing.
- **Configuration**: Edda supports many configuration options. In general, you can poll information from multiple accounts and multiple regions and can use the combination of account and regions that account points. Similarly, it provides different configurations for AWS, Crawler, Elector, and MongoDB.
- If you are using the AWS for hosting your microservice based product, then Edda serves the purpose of monitoring the AWS resources.

On-host performance monitoring – Vector

Vector is a static web application and runs inside a web browser. It allows it to monitor the performance of those hosts where **Performance Co-Pilot** (**PCP**) is installed. Vector supports PCP version 3.10+. PCP collects metrics and makes them available to Vector.

It provides high-resolution right metrics available on demand. This helps engineers to understand how a system behaves and correctly troubleshoot performance issues.

Note

A monitoring tool that helps you to monitor the performance of a remote host.

Distributed configuration management – Archaius

Archaius is a distributed configuration management tool that allows you to do the following:

- Use dynamic and typed properties
- Perform thread-safe configuration operations
- Check for property changes using a polling framework
- Use a callback mechanism in an ordered hierarchy of configurations
- Inspect and perform operations on properties using JConsole, as Archaius provides the JMX MBean
- A good configuration management tool is required when you have a microservices-based product. Archaius helps to configure different types of properties in a distributed environment.

Scheduler for Apache Mesos – Fenzo

Fenzo is a scheduler library for Apache Mesos frameworks written in Java. Apache Mesos frameworks match and assign resources to pending tasks. The following are its key features:

- It supports long-running service style tasks and for batch
- It can auto-scale the execution host cluster, based on resource demands
- It supports plugins that you can create based on requirements
- You can monitor resource-allocation failures, which allows you to debug the root cause

Cost and cloud utilization – Ice

Ice provides a bird's eye view of cloud resources from a cost and usage perspective. It provides the latest information of provisioned cloud resources allocation to different teams that add value for optimal utilization of the cloud resources.

Ice is a grail project. Users interacts with the Ice UI component that displays the information sent via the Ice reader component. The reader fetches information from the data generated by the Ice processor component. The Ice processor component reads data information from a detailed cloud billing file and converts it into data that is readable by the Ice reader component.

Other security tools – Scumblr and FIDO

Along with Security Monkey, Netflix OSS also makes use of Scumblr and **Fully Integrated Defense Operation** (**FIDO**) tools.

Note

To keep track and protect your microservices from regular threats and attacks, you need an automated way to secure and monitor your microservices. Netflix Scumblr and FIDO do this job for you.

Scumblr

Scumblr is a Ruby on Rails-based web application that allows you to perform periodic searches and

store/take action on the identified results. Basically, it gathers intelligence that leverages Internet-wide targeted searches to surface specific security issues for investigation.

Scumblr makes use of Workflowable gem to allow setting up flexible workflows for different types of results. Scumblr searches utilize plugins called **Search Providers**. It checks the anomaly like following. Since it is extensible, you can add as many as you want:

- Compromised credentials
- Vulnerability/hacking discussion
- Attack discussion
- Security-relevant social media discussion

Fully Integrated Defence Operation (FIDO)

FIDO is a security orchestration framework for analyzing events and automating incident responses. It automates the incident response process by evaluating, assessing and responding to malware. FIDO's primary purpose is to handle the heavy manual effort needed to evaluate threats coming from today's security stack and the large number of alerts generated by them.

As an orchestration platform, FIDO can make using your existing security tools more efficient and accurate by heavily reducing the manual effort needed to detect, notify, and respond to attacks against a network. For more information, you can refer these following links:

https://github.com/Netflix/Fido https://github.com/Netflix

References

- Monolithic (Etsy) versus Microservices (Netflix) Twitter discussion https://twitter.com/adrianco/status/441169921863860225
- *Monitoring Microservice and Containers Presentation* by Adrian Cockcroft: http://www.slideshare.net/adriancockcroft/gluecon-monitoring-microservices-and-containers-a-challenge
- Nanoservice Antipattern: http://arnon.me/2014/03/services-microservices-nanoservices/Apache Camel for Microservice Architectures: https://www.javacodegeeks.com/2014/09/apache-camel-for-micro%C2%ADservice-architectures.html
- Teamcity: https://www.jetbrains.com/teamcity/
- Jenkins: https://jenkins-ci.org/
- Loggly: https://www.loggly.com/

Summary

In this chapter, we have explored various practices and principles, which are best-suited for microservices-based products and services. Microservices architecture is a result of cloud environments, which are being used widely in comparison to on-premise-based monolithic systems. We have identified a few of the principals related to size, agility, and testing, that have to be in place for successful implementation.

We have also got an overview of different tools used by Netflix OSS for the various key features required for successful implementation of microservices architecture-based products and services. Netflix offers a video rental service, using the same tools successfully.

In the next chapter, readers may encounter issues and they may get stuck at those problems. The chapter explains the common problems encountered during the development of microservices, and their solutions.

Chapter 9. Troubleshooting Guide

We have come so far and I am sure you are enjoying each and every moment of this challenging and joyful learning journey. I will not say that this book ends after this chapter, but rather you are completing the first milestone. This milestone opens the doors for learning and implementing a new paradigm in the cloud with microservice-based design. I would like to reaffirm that integration testing is an important way to test interaction among microservices and APIs. While working on your sample app **Online Table Reservation System** (**OTRS**), I am sure you faced many challenges, especially while debugging the app. Here, we will cover a few of the practices and tools that will help you to troubleshoot the deployed application, Docker containers, and host machines.

This chapter covers the following three topics:

- Logging and ELK stack
- Use of correlation ID for service calls
- Dependencies and versions

Logging and ELK stack

Can you imagine debugging any issue without seeing a log on the production system? Simply, no, as it would be difficult to go back in time. Therefore, we need logging. Logs also give us warning signals about the system if they are designed and coded that way. Logging and log analysis is an important step for troubleshooting any issue, and also for throughput, capacity, and monitoring the health of the system. Therefore, having a very good logging platform and strategy will enable effective debugging. Logging is one of the most important key components of software development in the initial days.

Microservices are generally deployed using image containers like Docker that provide the log with commands that help you to read logs of services deployed inside the containers. Docker and Docker Compose provide commands to stream the log output of running services within the container and in all containers respectively. Please refer to the following `logs` command of Docker and Docker Compose:

Note

Docker logs command:

Usage: `docker logs [OPTIONS] <CONTAINER NAME>`

```
Fetch the logs of a container:
  -f, --follow    Follow log output
  --help      Print usage
  --since=""                  Show logs since timestamp
  -t, --timestamps            Show timestamps
  --tail="all"                Number of lines to show from the end of the logs
```

Docker Compose logs Command:

```
Usage: docker-compose logs [options] [SERVICE...]
```

Options:

```
--no-color  Produce monochrome output
-f, --follow  Follow log output
-t, --timestamps  Show timestamps
--tail    Number of lines to show from the end of the logs for each  container
[SERVICES…]  Service representing the container - you can give multiple
```

These commands help you to explore the logs of microservices and other processes running inside the containers. As you can see, using the above commands would be a challenging task when you have a higher number of services. For example, if you have 10s or 100s of microservices, it would be very difficult to track each microservice log. Similarly, you can imagine, even without containers, how difficult it would be to monitor logs individually. Therefore, you can assume the difficulty of exploring and correlating the logs of 10s to 100s of containers. It is time-consuming and adds very little value.

Therefore, a log aggregator and visualizing tools like the ELK stack come to our rescue. It will be used for centralizing logging. We'll explore this in the next section.

A brief overview

The **Elasticsearch, Logstash, Kibana** (**ELK**) stack is a chain of tools that performs log aggregation, analysis, visualization, and monitoring. The ELK stack provides a complete logging platform that allows you to analyze, visualize, and monitor all your logs, including all types of product logs and system logs. If you already know about the ELK stack, please skip to the next section. Here, we'll provide a brief introduction to each tool in the ELK Stack.

Elasticsearch

Elasticsearch is one of the most popular enterprise full text search engines. It is open sourced software. It is distributable and supports multitenancy. A single Elasticsearch server stores multiple indexes (each index represents a database), and a single query can search data of multiple indexes. It is a distributed search engine and supports clustering.

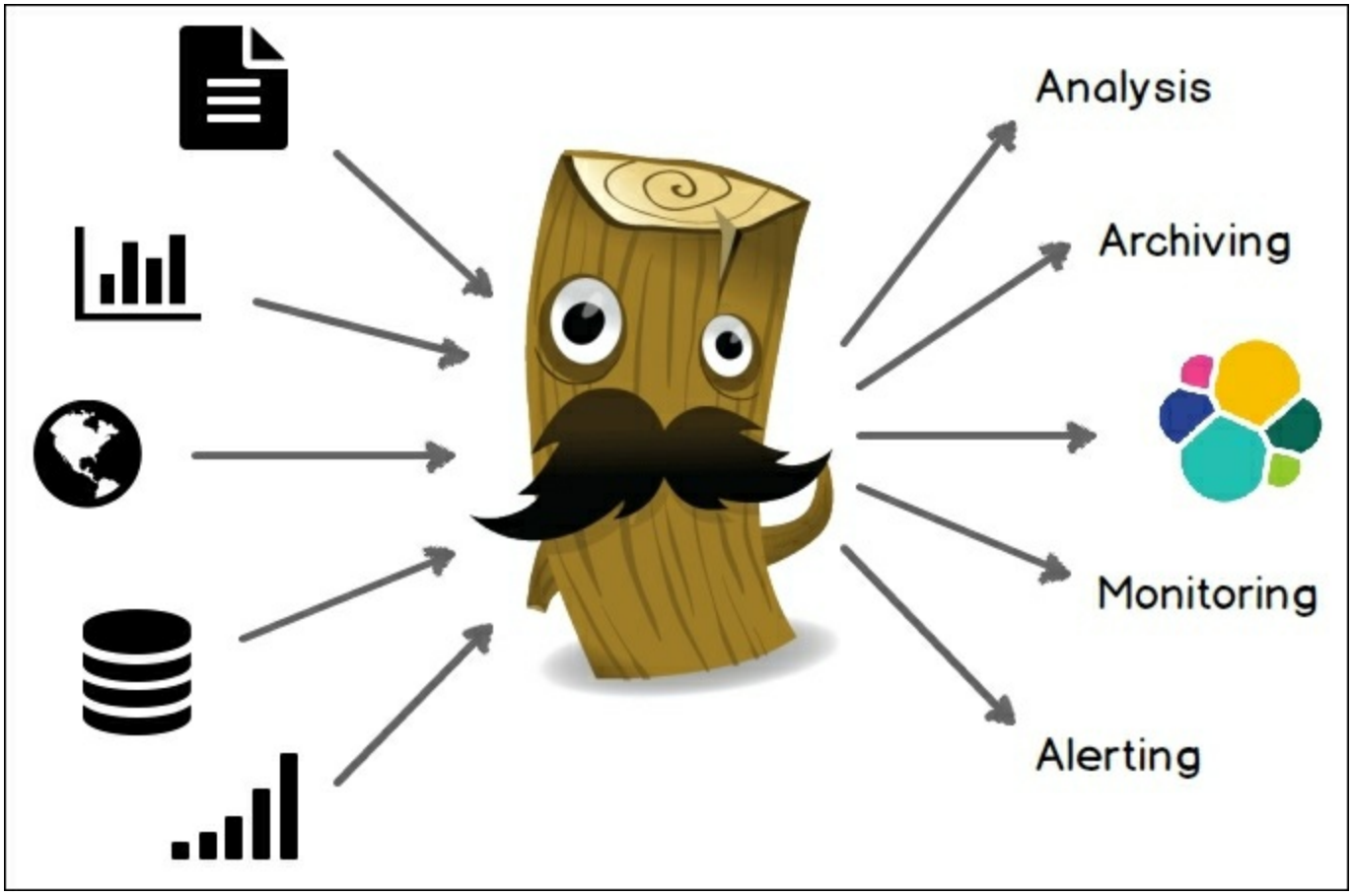
It is readily scalable and can provide near real-time searches with a latency of 1 second. It is developed in Java using Apache Lucene. Apache Lucene is also free, open sourced, and it provides the core of Elasticsearch, aka the informational retrieval software library.

Elasticsearch APIs are extensive in nature and very elaborative. Elasticsearch provides a JSON-based schema, less storage, and represents data models in JSON. Elasticsearch APIs use JSON documents for HTTP requests and responses.

Logstash

Logstash is an open source data collection engine with real-time pipeline capabilities. In simple words, it collects, parses, processes, and stores the data. Since Logstash has data pipeline capabilities, helping you to process any event data, like logs, from a variety of systems. Logstash runs as an agent that collects the data, parses it, filters it, and sends the output to a designated app, such as Elasticsearch, or simple standard output on a console.

It is also has a very good plugin ecosystem (image sourced from www.elastic.co):



Logstash ecosystem

Kibana

Kibana is an open source analytics and visualization web application. It is designed to work with Elasticsearch. You use Kibana to search, view, and interact with data stored in Elasticsearch indices.

It is a browser-based web application that lets you perform advanced data analysis and visualize your data in a variety of charts, tables, and maps. Moreover, it is a zero-configuration application. Therefore, it neither needs any coding nor additional infrastructure after installation.

ELK stack setup

Generally, these tools are installed individually and then configured to communicate with each other. The installation of these components is pretty straight forward. Download the installable artifact from the designated location and follow the installation steps as shown in the next section.

The installation steps provided below are part of a basic setup, which is required for setting up the ELK stack you want to run. Since this installation was done on my localhost machine, I have used the host localhost. It can be changed easily with any respective host name that you want.

Installing Elasticsearch

We can install Elasticsearch by following these steps:

1. Download the latest Elasticsearch distribution from https://www.elastic.co/downloads/elasticsearch.
2. Unzip it to the desired location in your system.
3. Make sure the latest Java version is installed and the `JAVA_HOME` environment variable is set.
4. Go to Elasticsearch home and run `bin/elasticsearch` on Unix-based systems and `bin/elasticsearch.bat` on Windows.
5. Open any browser and hit `http://localhost:9200/`. On successful installation it should provide you a JSON object similar to that shown as follows:

```
{
  "name" : "Leech",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.3.1",
    "build_hash" : "bd980929010aef404e7cb0843e61d0665269fc39",
    "build_timestamp" : "2016-04-04T12:25:05Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.0"
  },
  "tagline" : "You Know, for Search"
}
```

   By default, the GUI is not installed. You can install one by executing the following command from the `bin` directory; make sure the system is connected to the Internet:

```
plugin -install mobz/elasticsearch-head
```

6. Now you can access the GUI interface with the URL `http://localhost:9200/_plugin/head/`.

   You can replace `localhost` and **9200** with your respective hostname and port number.

Installing Logstash

We can install Logstash by following the given steps:

1. Download the latest Logstash distribution from https://www.elastic.co/downloads/logstash.
2. Unzip it to the desired location in your system.

   Prepare a configuration file, as shown below. It instructs Logstash to read input from given files and passes it to Elasticsearch (see the following `config` file; Elasticsearch is represented by localhost and `9200` port). It is the simplest configuration file. To add filters and learn more about Logstash, you can explore the Logstash reference documentation available at https://www.elastic.co/guide/en/logstash/current/index.html.

   ### Note

   As you can see, the OTRS `service` log and `edge-server` log are added as input. Similarly, you can also add log files of other microservices.

```
input {
  ### OTRS ###
  file {
    path => "\logs\otrs-service.log"
    type => "otrs-api"
    codec => "json"
    start_position => "beginning"
  }

  ### edge ###
  file {
    path => "/logs/edge-server.log"
    type => "edge-server"
    codec => "json"
  }
}

output {
  stdout {
    codec => rubydebug
  }
  elasticsearch {
    hosts => "localhost:9200"
  }
}
```

3. Go to Logstash home and run `bin/logstash agent -f logstash.conf` on Unix-based systems and `bin/logstash.bat agent -f logstash.conf` on Windows. Here, Logstash is executed using the `agent` command. Logstash agent collects data from the sources provided in the input field in the configuration file and sends the output to Elasticsearch. Here, we have not used the filters, because otherwise it may process the input data before providing it to Elasticsearch.

Installing Kibana

We can install the Kibana web application by following the given steps:

1. Download the latest Kibana distribution from https://www.elastic.co/downloads/kibana.
2. Unzip it to the desired location in your system.
3. Open the configuration file `config/kibana.yml` from the Kibana home directory and point the `elasticsearch.url` to the previously configured Elasticsearch instance:

   ```
   elasticsearch.url: "http://localhost:9200"
   ```

4. Go to Kibana home and run `bin/kibana agent -f logstash.conf` on Unix-based systems and `bin/kibana.bat agent -f logstash.conf` on Windows.
5. Now you can access the Kibana app from your browser using the URL `http://localhost:5601/`.

   To learn more about Kibana, explore the Kibana reference documentation at https://www.elastic.co/guide/en/kibana/current/getting-started.html.

As we followed the above steps, you may have noticed that it requires some amount of effort. If you want to avoid a manual setup, you can Dockerize it. If you don't want to put effort into creating the Docker container of the ELK stack, you can choose one from Docker Hub. On Docker Hub there are many ready-made ELK stack Docker images. You can try different ELK containers and choose the one that suits you the most. willdurand/elk is the most downloaded container and is easy to start, working well with Docker Compose.

Tips for ELK stack implementation

- To avoid any data loss and handle the sudden spike of input load, using a broker, such as Redis or RabbitMQ, is recommended between Logstash and Elasticsearch.
- Use an odd number of nodes for Elasticsearch if you are using clustering to prevent the split-brain problem.
- In Elasticsearch, always use the appropriate field type for given data. This will allow you to perform different checks, for example, the `int` field type will allow you to perform (`"http_status: <400"`) or (`"http_status:=200"`). Similarly, other field types also allow you to perform similar checks.

Use of correlation ID for service calls

When you make a call to any REST endpoint and if any issue pops up, it is difficult to trace the issue and its root origin because each call is made to server, and this call may call another and so on and so forth. This makes it very difficult to figure out how one particular request was transformed and what it was called. Normally, an issue that is caused by one service can cause service elsewhere. It is very difficult to track and may require an enormous amount of effort. If it is monolithic, you know that you are looking in the right direction but microservices make it difficult to understand what the source of the issue is and where you should get your data.

Let's see how we can tackle this problem

By using a correlation ID that is passed across all calls, it allows you to track each request and track the route easily. Each request will have its unique correlation ID. Therefore, when we debug any issue, the correlation ID is our starting point. We can follow it, and along the way, we can find out what went wrong.

The correlation ID requires some extra development effort, but it's effort well spent as it helps a lot in the long run. When a request travels between different microservices, you will be able to see all interactions and which service has problems.

This is not something new or invented for microservices. This pattern is already being used by many popular products such as Microsoft SharePoint.

Dependencies and versions

Two common problems that we face in product development are cyclic dependencies and API versions. We'll discuss them in terms of microservice based architectures.

Cyclic dependencies and their impact

Generally, monolithic architecture has a typical layer model, whereas microservices carry the graph model. Therefore, microservices may have cyclic dependencies.

Therefore, it is necessary to keep a dependency check on microservice relationships.

Let us have a look at the following two cases:

- If you have a cycle of dependencies between your microservices, you are vulnerable to distributed stack overflow errors when a certain transaction might be stuck in a loop. For example, when a restaurant table is being reserved by a person. In this case, the restaurant needs to know the person (`findBookedUser`), and the person needs to know the restaurant at a given time (`findBookedRestaurant`). If it is not designed well, these services may call each other in loop. The result may be a stack overflow generated by JVM.
- If two services share a dependency and you update that other service's API in a way that could affect them, you'll need to updated all three at once. This brings up questions like, which should you update first? In addition, how do you make this a safe transition?

It needs to be analyzed while designing the system

Therefore, it is important while designing the microservices to establish the proper relationship between different services internally to avoid any cyclic dependencies. It is a design issue and must be addressed even if it requires a refactoring of the code.

Maintaining different versions

When you have more services, it means different release cycles for each of them, which adds to this complexity by introducing different versions of services, in that there will be different versions of the same REST services. Reproducing the solution to a problem will prove to be very difficult when it has gone in one version and returns in a newer one.

Let's explore more

The versioning of APIs is important because with time APIs change. Your knowledge and experience improves with time, and that leads to changes in APIs. Changing APIs may break existing client integrations.

Therefore, there are various ways for managing the API versions. One of these is using the version in the path that we have used in this book; some also use the HTTP header. The HTTP header could be a custom request header or you could use the *Accept Header* for representing the calling API version. For more information on how versions are handled using HTTP headers, please refer to *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing: https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices.

It is very important while troubleshooting any issue that your microservices are implemented to produce the version numbers in logs. In addition, ideally, you should avoid any instance where you have too many versions of any microservice.

References

This following links will have more information:

- Elasticsearch: https://www.elastic.co/products/elasticsearch
- Logstash: https://www.elastic.co/products/logstash
- Kibana: https://www.elastic.co/products/kibana
- `willdurand/elk`: ELK Docker image
- *Mastering Elasticsearch – Second Edition*: https://www.packtpub.com/web-development/mastering-elasticsearch-second-edition

Summary

In this chapter, we have explored the ELK stack overview and installation. In the ELK stack, Elasticsearch is used for storing the logs and service queries from Kibana. Logstash is an agent that runs on each server that you wish to collect logs from. Logstash reads the logs, filters/transforms them, and provides them to Elasticsearch. Kibana reads/queries the data from Elasticsearch and presents it in tabular or graphical visualizations.

We also understand the utility of having the correlation ID while debugging issues. At the end of this chapter, we also discovered the shortcomings of a few microservice designs. It was a challenging task to cover all the topics relating to microservices in this book, so I tried to include as much relevant information as possible with precise sections with references, which allow you to explore more. Now I would like to let you start implementing the concepts we have learned in this chapter to your workplace or in your personal projects. This will not only give you hands-on experience, but may also allow you to master microservices. In addition, you will also be able to participate in local meetups and conferences.

Part 2. Module 2

Spring Microservices

Build scalable microservices with Spring, Docker, and Mesos

Chapter 1. Demystifying Microservices

Microservices are an architecture style and an approach for software development to satisfy modern business demands. Microservices are not invented; they are more of an evolution from the previous architecture styles.

We will start the chapter by taking a closer look at the evolution of the microservices architecture from the traditional monolithic architectures. We will also examine the definition, concepts, and characteristics of microservices. Finally, we will analyze typical use cases of microservices and establish the similarities and relationships between microservices and other architecture approaches such as **Service Oriented Architecture** (**SOA**) and Twelve-Factor Apps. Twelve-Factor Apps defines a set of software engineering principles of developing applications targeting the cloud.

In this chapter you, will learn about:

- The evolution of microservices
- The definition of the microservices architecture with examples
- Concepts and characteristics of the microservices architecture
- Typical use cases of the microservices architecture
- The relationship of microservices with SOA and Twelve-Factor Apps

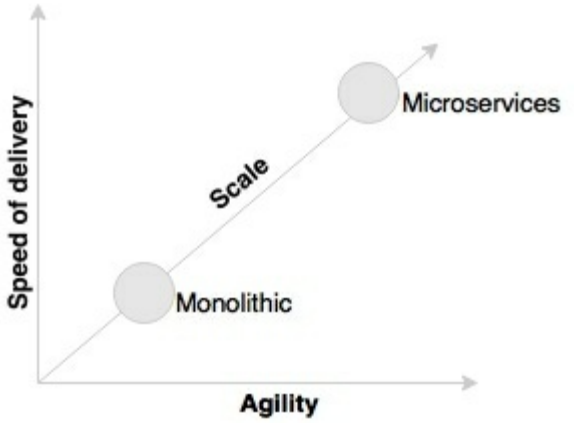
The evolution of microservices

Microservices are one of the increasingly popular architecture patterns next to SOA, complemented by DevOps and cloud. The microservices evolution is greatly influenced by the disruptive digital innovation trends in modern business and the evolution of technologies in the last few years. We will examine these two factors in this section.

Business demand as a catalyst for microservices evolution

In this era of digital transformation, enterprises increasingly adopt technologies as one of the key enablers for radically increasing their revenue and customer base. Enterprises primarily use social media, mobile, cloud, big data, and Internet of Things as vehicles to achieve the disruptive innovations. Using these technologies, enterprises find new ways to quickly penetrate the market, which severely pose challenges to the traditional IT delivery mechanisms.

The following graph shows the state of traditional development and microservices against the new enterprise challenges such as agility, speed of delivery, and scale.

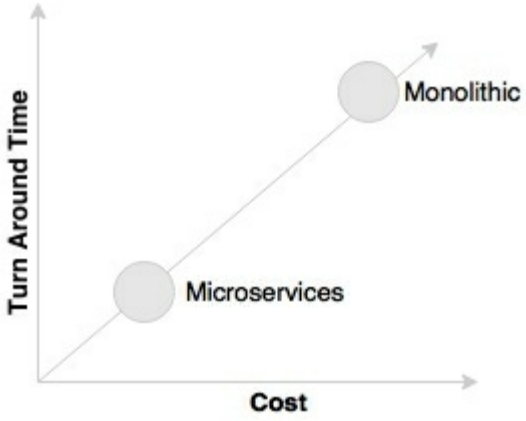


Tip

Microservices promise more agility, speed of delivery, and scale compared to traditional monolithic applications.

Gone are the days when businesses invested in large application developments with the turnaround time of a few years. Enterprises are no longer interested in developing consolidated applications to manage their end-to-end business functions as they did a few years ago.

The following graph shows the state of traditional monolithic applications and microservices in comparison with the turnaround time and cost.

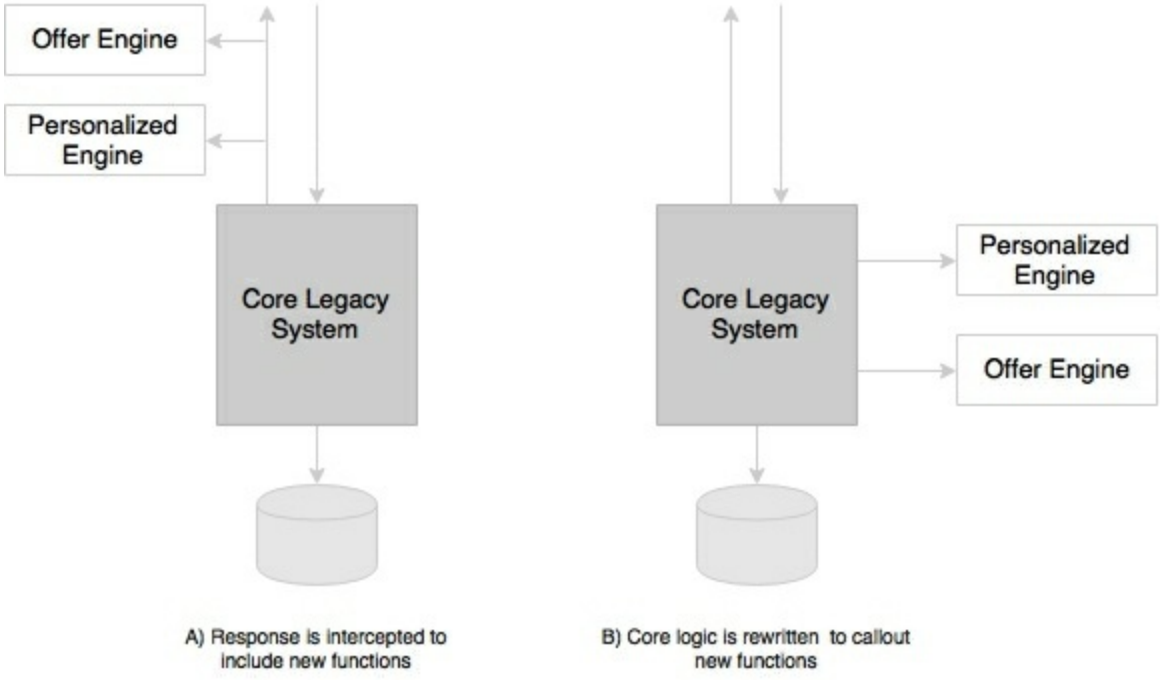


Tip

Microservices provide an approach for developing quick and agile applications, resulting in less overall cost.

Today, for instance, airlines or financial institutions do not invest in rebuilding their core mainframe systems as another monolithic monster. Retailers and other industries do not rebuild heavyweight supply chain management applications, such as their traditional ERPs. Focus has shifted to building quick-win point solutions that cater to specific needs of the business in the most agile way possible.

Let's take an example of an online retailer running with a legacy monolithic application. If the retailer wants to innovate his/her sales by offering their products personalized to a customer based on the customer's past shopping, preferences, and so on and also wants to enlighten customers by offering products based on their propensity to buy them, they will quickly develop a personalization engine or offers based on their immediate needs and plug them into their legacy application.



As shown in the preceding diagram, rather than investing in rebuilding the core legacy system, this will be either done by passing the responses through the new functions, as shown in the diagram marked **A**, or by modifying the core legacy system to call out these functions as part of the processing, as shown in the diagram marked **B**. These functions are typically written as microservices.

This approach gives organizations a plethora of opportunities to quickly try out new functions with lesser cost in an experimental mode. Businesses can later validate key performance indicators and alter or replace these implementations if required.

Tip

Modern architectures are expected to maximize the ability to replace their parts and minimize the cost of replacing their parts. The microservices approach is a means to achieving this.

Technology as a catalyst for the microservices evolution

Emerging technologies have also made us rethink the way we build software systems. For example, a few decades back, we couldn't even imagine a distributed application without a two-phase commit. Later, NoSQL databases made us think differently.

Similarly, these kinds of paradigm shifts in technology have reshaped all the layers of the software architecture.

The emergence of HTML 5 and CSS3 and the advancement of mobile applications repositioned user interfaces. Client-side JavaScript frameworks such as Angular, Ember, React, Backbone, and so on are immensely popular due to their client-side rendering and responsive designs.

With cloud adoptions steamed into the mainstream, **Platform as a Services** (**PaaS**) providers such as Pivotal CF, AWS, Salesforce.com, IBMs Bluemix, RedHat OpenShift, and so on made us rethink the way we build middleware components. The container revolution created by Docker radically influenced the infrastructure space. These days, an infrastructure is treated as a commodity service.

The integration landscape has also changed with **Integration Platform as a Service** (**iPaaS**), which is emerging. Platforms such as Dell Boomi, Informatica, MuleSoft, and so on are examples of iPaaS. These tools helped organizations stretch integration boundaries beyond the traditional enterprise.

NoSQLs have revolutionized the databases space. A few years ago, we had only a few popular databases, all based on relational data modeling principles. We have a long list of databases today: Hadoop, Cassandra, CouchDB, and Neo 4j to name a few. Each of these databases addresses certain specific architectural problems.

Imperative architecture evolution

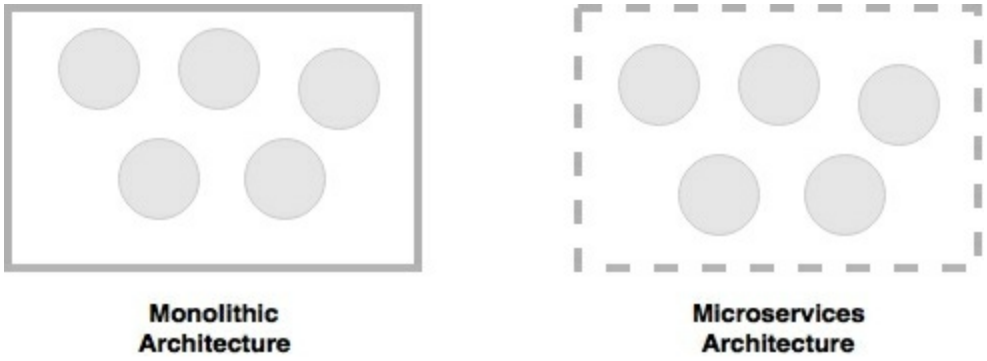
Application architecture has always been evolving alongside demanding business requirements and the evolution of technologies. Architectures have gone through the evolution of age-old mainframe systems to fully abstract cloud services such as AWS Lambda.

Tip

Using AWS Lambda, developers can now drop their "functions" into a fully managed compute service.

Read more about Lambda at: https://aws.amazon.com/documentation/lambda/

Different architecture approaches and styles such as mainframes, client server, N-tier, and service-oriented were popular at different timeframes. Irrespective of the choice of architecture styles, we always used to build one or the other forms of monolithic architectures. The microservices architecture evolved as a result of modern business demands such as agility and speed of delivery, emerging technologies, and learning from previous generations of architectures.



Microservices help us break the boundaries of monolithic applications and build a logically independent smaller system of systems, as shown in the preceding diagram.

Tip

If we consider monolithic applications as a set of logical subsystems encompassed with a physical boundary, microservices are a set of independent subsystems with no enclosing physical boundary.

What are microservices?

Microservices are an architecture style used by many organizations today as a game changer to achieve a high degree of agility, speed of delivery, and scale. Microservices give us a way to develop more physically separated modular applications.

Microservices are not invented. Many organizations such as Netflix, Amazon, and eBay successfully used the divide-and-conquer technique to functionally partition their monolithic applications into smaller atomic units, each performing a single function. These organizations solved a number of prevailing issues they were experiencing with their monolithic applications.

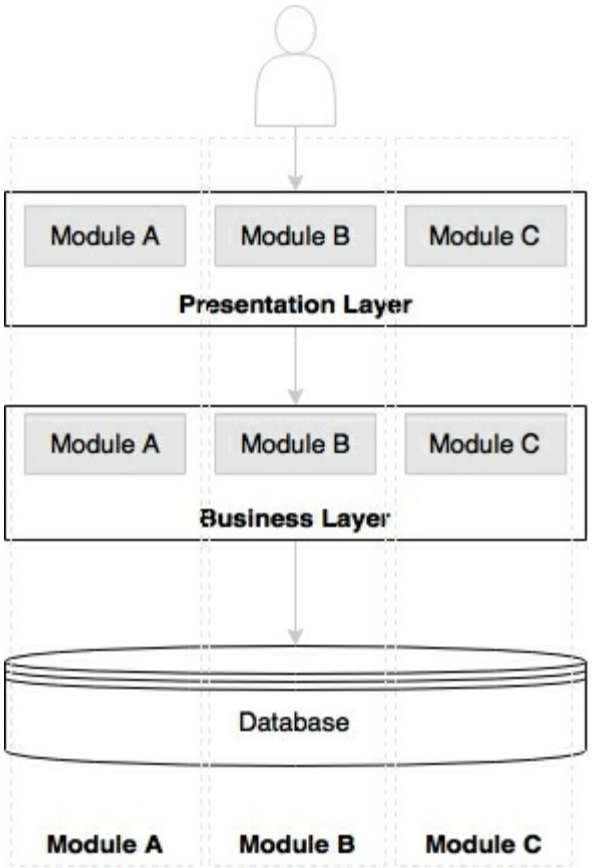
Following the success of these organizations, many other organizations started adopting this as a common pattern to refactor their monolithic applications. Later, evangelists termed this pattern as the microservices architecture.

Microservices originated from the idea of hexagonal architecture coined by Alistair Cockburn. Hexagonal architecture is also known as the Ports and Adapters pattern.

Tip

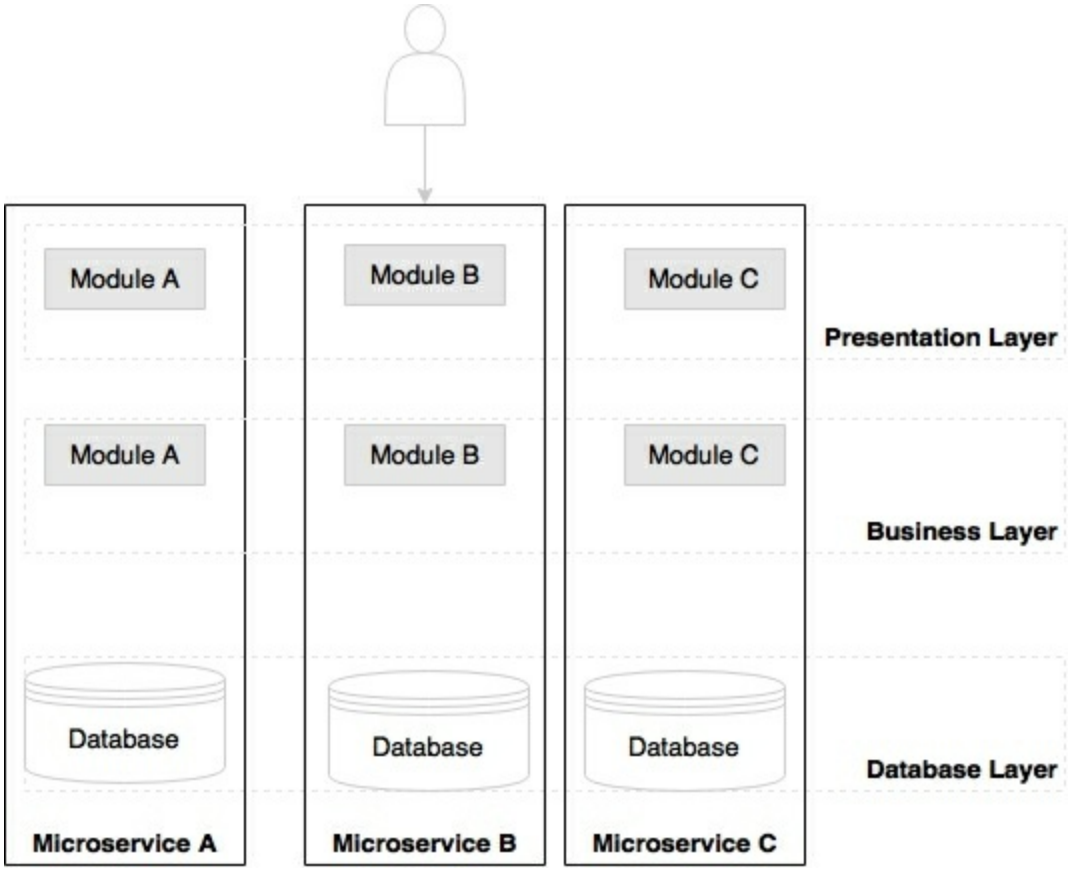
Read more about hexagonal architecture at http://alistair.cockburn.us/Hexagonal+architecture.

Microservices are an architectural style or an approach to building IT systems as a set of business capabilities that are autonomous, self-contained, and loosely coupled:



The preceding diagram depicts a traditional N-tier application architecture having a presentation layer, business layer, and database layer. The modules **A**, **B**, and **C** represent three different business capabilities. The layers in the diagram represent a separation of architecture concerns. Each layer holds all three business capabilities pertaining to this layer. The presentation layer has web components of all the three modules, the business layer has business components of all the three modules, and the database hosts tables of all the three modules. In most cases, layers are physically spreadable, whereas modules within a layer are hardwired.

Let's now examine a microservices-based architecture.



As we can note in the preceding diagram, the boundaries are inversed in the microservices architecture. Each vertical slice represents a microservice. Each microservice has its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities. By doing so, changes to one microservice do not impact others.

There is no standard for communication or transport mechanisms for microservices. In general, microservices communicate with each other using widely adopted lightweight protocols, such as HTTP and REST, or messaging protocols, such as JMS or AMQP. In specific cases, one might choose more optimized communication protocols, such as Thrift, ZeroMQ, Protocol Buffers, or Avro.

As microservices are more aligned to business capabilities and have independently manageable life

cycles, they are the ideal choice for enterprises embarking on DevOps and cloud. DevOps and cloud are two facets of microservices.

Tip

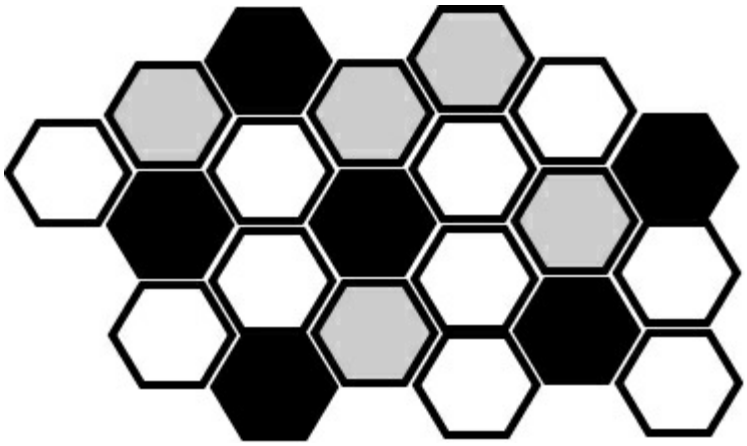
DevOps is an IT realignment to narrow the gap between traditional IT development and operations for better efficiency.

Read more about DevOps:

http://dev2ops.org/2010/02/what-is-devops/

Microservices – the honeycomb analogy

The honeycomb is an ideal analogy for representing the evolutionary microservices architecture.



In the real world, bees build a honeycomb by aligning hexagonal wax cells. They start small, using different materials to build the cells. Construction is based on what is available at the time of building. Repetitive cells form a pattern and result in a strong fabric structure. Each cell in the honeycomb is independent but also integrated with other cells. By adding new cells, the honeycomb grows organically to a big, solid structure. The content inside each cell is abstracted and not visible outside. Damage to one cell does not damage other cells, and bees can reconstruct these cells without impacting the overall honeycomb.

Principles of microservices

In this section, we will examine some of the principles of the microservices architecture. These principles are a "must have" when designing and developing microservices.

Single responsibility per service

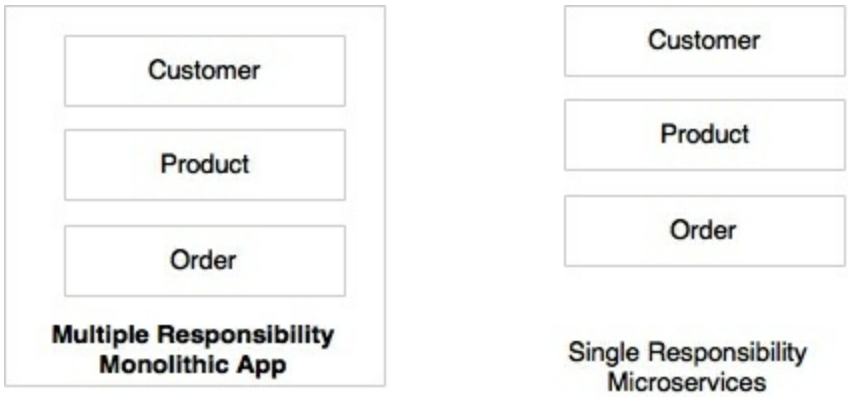
The single responsibility principle is one of the principles defined as part of the SOLID design pattern. It states that a unit should only have one responsibility.

Tip

Read more about the SOLID design pattern at:

http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

This implies that a unit, either a class, a function, or a service, should have only one responsibility. At no point should two units share one responsibility or one unit have more than one responsibility. A unit with more than one responsibility indicates tight coupling.



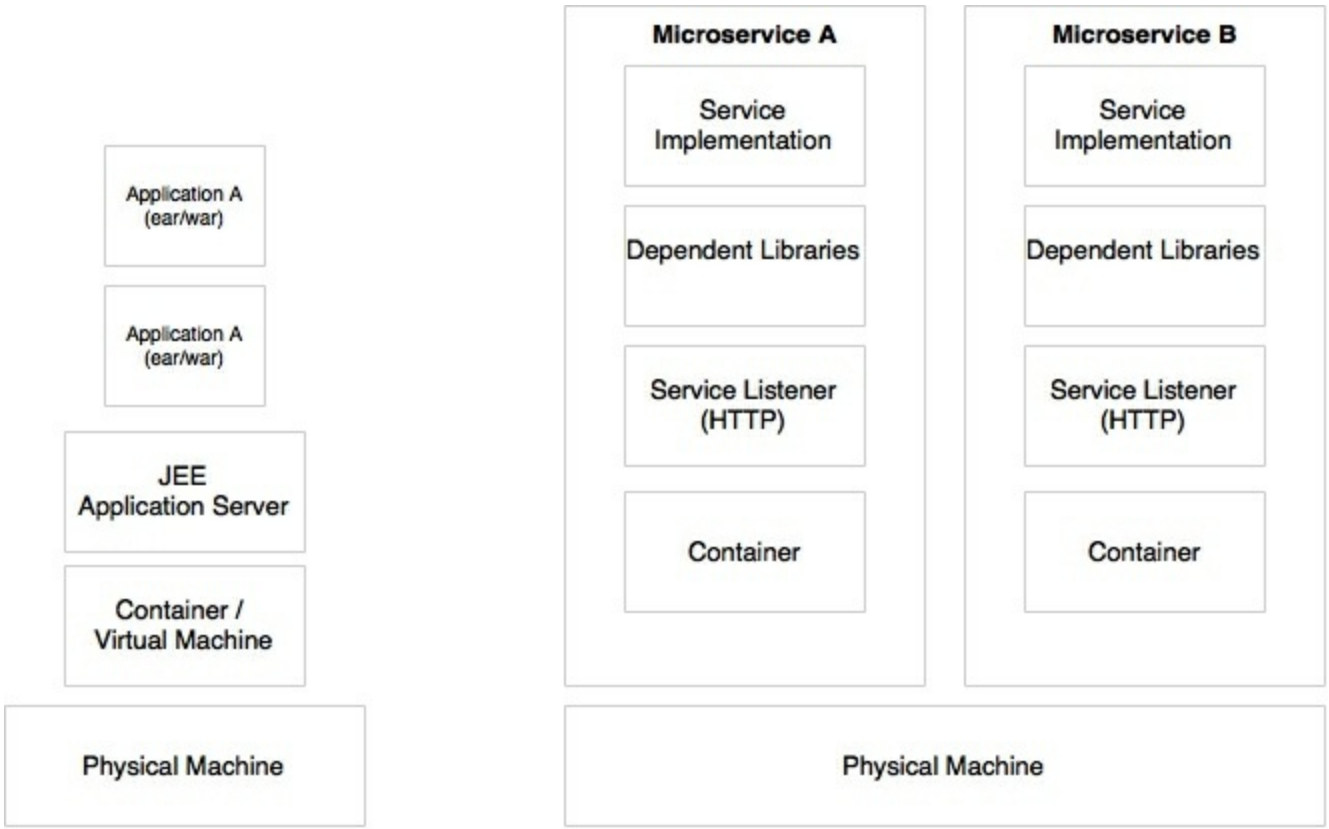
As shown in the preceding diagram, **Customer**, **Product**, and **Order** are different functions of an e-commerce application. Rather than building all of them into one application, it is better to have three different services, each responsible for exactly one business function, so that changes to one responsibility will not impair others. In the preceding scenario, **Customer**, **Product**, and **Order** will be treated as three independent microservices.

Microservices are autonomous

Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution. They bundle all dependencies, including library dependencies, and execution environments such as web servers and containers or virtual machines that abstract physical resources.

One of the major differences between microservices and SOA is in their level of autonomy. While most SOA implementations provide service-level abstraction, microservices go further and abstract the realization and execution environment.

In traditional application developments, we build a WAR or an EAR, then deploy it into a JEE application server, such as with JBoss, WebLogic, WebSphere, and so on. We may deploy multiple applications into the same JEE container. In the microservices approach, each microservice will be built as a fat Jar, embedding all dependencies and run as a standalone Java process.



Microservices may also get their own containers for execution, as shown in the preceding diagram. Containers are portable, independently manageable, lightweight runtime environments. Container technologies, such as Docker, are an ideal choice for microservices deployment.

Characteristics of microservices

The microservices definition discussed earlier in this chapter is arbitrary. Evangelists and practitioners have strong but sometimes different opinions on microservices. There is no single, concrete, and universally accepted definition for microservices. However, all successful microservices implementations exhibit a number of common characteristics. Therefore, it is important to understand these characteristics rather than sticking to theoretical definitions. Some of the common characteristics are detailed in this section.

Services are first-class citizens

In the microservices world, services are first-class citizens. Microservices expose service endpoints as APIs and abstract all their realization details. The internal implementation logic, architecture, and technologies (including programming language, database, quality of services mechanisms, and so on) are completely hidden behind the service API.

Moreover, in the microservices architecture, there is no more application development; instead, organizations focus on service development. In most enterprises, this requires a major cultural shift in the way that applications are built.

In a **Customer Profile** microservice, internals such as the data structure, technologies, business logic, and so on are hidden. They aren't exposed or visible to any external entities. Access is restricted through the service endpoints or APIs. For instance, Customer Profile microservices may expose **Register Customer** and **Get Customer** as two APIs for others to interact with.

Characteristics of services in a microservice

As microservices are more or less like a flavor of SOA, many of the service characteristics defined in the SOA are applicable to microservices as well.

The following are some of the characteristics of services that are applicable to microservices as well:

- **Service contract**: Similar to SOA, microservices are described through well-defined service contracts. In the microservices world, JSON and REST are universally accepted for service communication. In the case of JSON/REST, there are many techniques used to define service contracts. JSON Schema, WADL, Swagger, and RAML are a few examples.
- **Loose coupling**: Microservices are independent and loosely coupled. In most cases, microservices accept an event as input and respond with another event. Messaging, HTTP, and REST are commonly used for interaction between microservices. Message-based endpoints provide higher levels of decoupling.
- **Service abstraction**: In microservices, service abstraction is not just an abstraction of service realization, but it also provides a complete abstraction of all libraries and environment details, as discussed earlier.
- **Service reuse**: Microservices are course-grained reusable business services. These are accessed by mobile devices and desktop channels, other microservices, or even other systems.
- **Statelessness**: Well-designed microservices are stateless and share nothing with no shared state or conversational state maintained by the services. In case there is a requirement to maintain state, they are maintained in a database, perhaps in memory.
- **Services are discoverable**: Microservices are discoverable. In a typical microservices environment, microservices self-advertise their existence and make themselves available for discovery. When services die, they automatically take themselves out from the microservices ecosystem.

- **Service interoperability**: Services are interoperable as they use standard protocols and message exchange standards. Messaging, HTTP, and so on are used as transport mechanisms. REST/JSON is the most popular method for developing interoperable services in the microservices world. In cases where further optimization is required on communications, other protocols such as Protocol Buffers, Thrift, Avro, or Zero MQ could be used. However, the use of these protocols may limit the overall interoperability of the services.
- **Service composeability**: Microservices are composeable. Service composeability is achieved either through service orchestration or service choreography.

Tip

More detail on SOA principles can be found at:

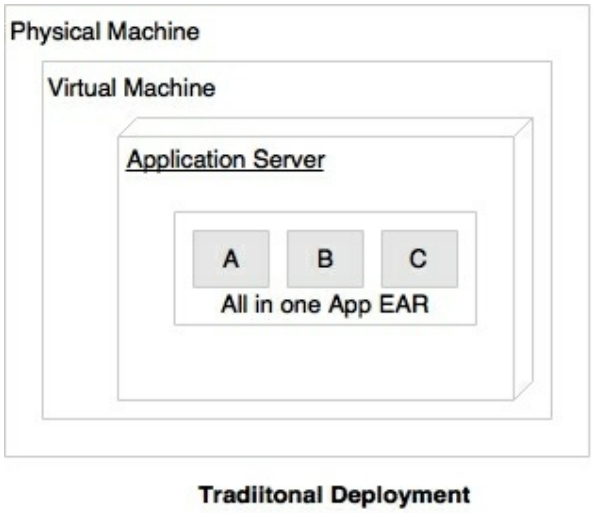
http://serviceorientation.com/serviceorientation/index

Microservices are lightweight

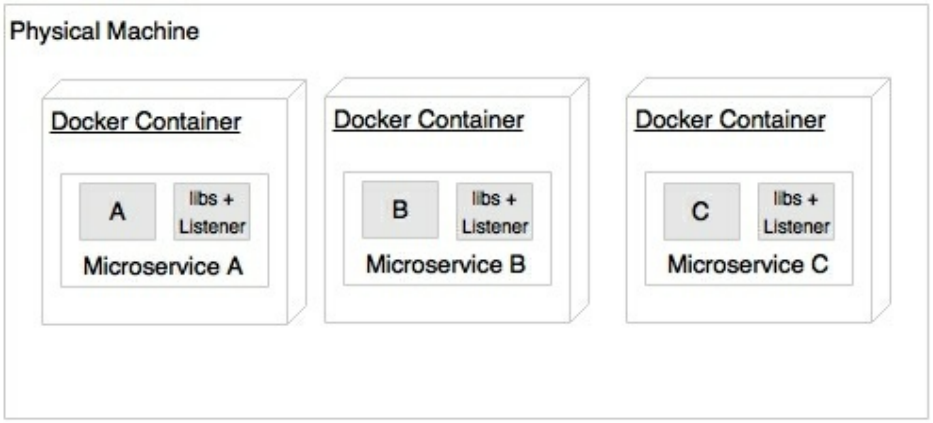
Well-designed microservices are aligned to a single business capability, so they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints.

When selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight so that the overall footprint remains manageable. For example, Jetty or Tomcat are better choices as application containers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.

Container technologies such as Docker also help us keep the infrastructure footprint as minimal as possible compared to hypervisors such as VMWare or Hyper-V.



Traditonal Deployment



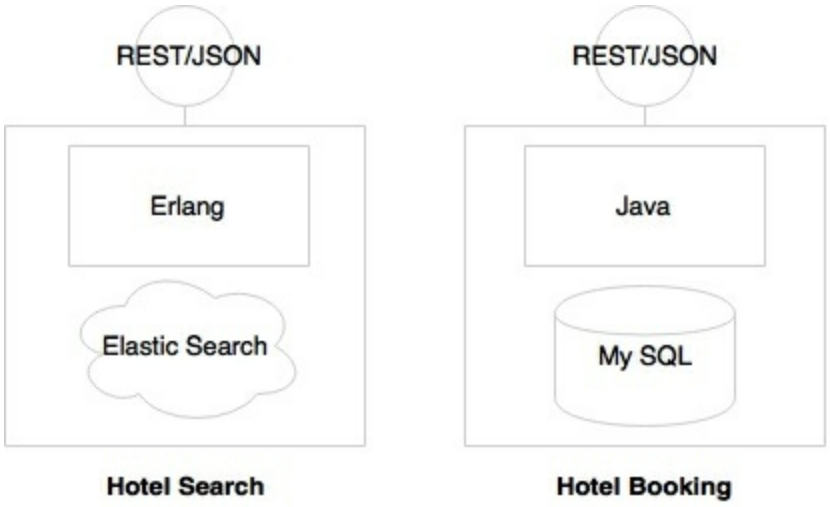
Microservices Deployment

As shown in the preceding diagram, microservices are typically deployed in Docker containers, which encapsulate the business logic and needed libraries. This help us quickly replicate the entire setup on a new machine or on a completely different hosting environment or even to move across different cloud providers. As there is no physical infrastructure dependency, containerized microservices are easily portable.

Microservices with polyglot architecture

As microservices are autonomous and abstract everything behind service APIs, it is possible to have different architectures for different microservices. A few common characteristics that we see in microservices implementations are:

- Different services use different versions of the same technologies. One microservice may be written on Java 1.7, and another one could be on Java 1.8.
- Different languages are used to develop different microservices, such as one microservice is developed in Java and another one in Scala.
- Different architectures are used, such as one microservice using the Redis cache to serve data, while another microservice could use MySQL as a persistent data store.

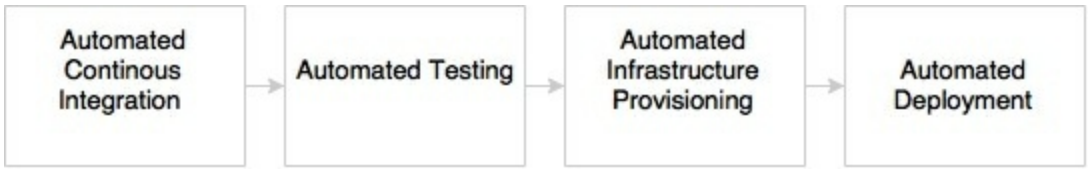


In the preceding example, as **Hotel Search** is expected to have high transaction volumes with stringent performance requirements, it is implemented using Erlang. In order to support predictive searching, Elasticsearch is used as the data store. At the same time, **Hotel Booking** needs more ACID transactional characteristics. Therefore, it is implemented using MySQL and Java. The internal implementations are hidden behind service endpoints defined as REST/JSON over HTTP.

Automation in a microservices environment

Most of the microservices implementations are automated to a maximum from development to production.

As microservices break monolithic applications into a number of smaller services, large enterprises may see a proliferation of microservices. A large number of microservices is hard to manage until and unless automation is in place. The smaller footprint of microservices also helps us automate the microservices development to the deployment life cycle. In general, microservices are automated end to end—for example, automated builds, automated testing, automated deployment, and elastic scaling.

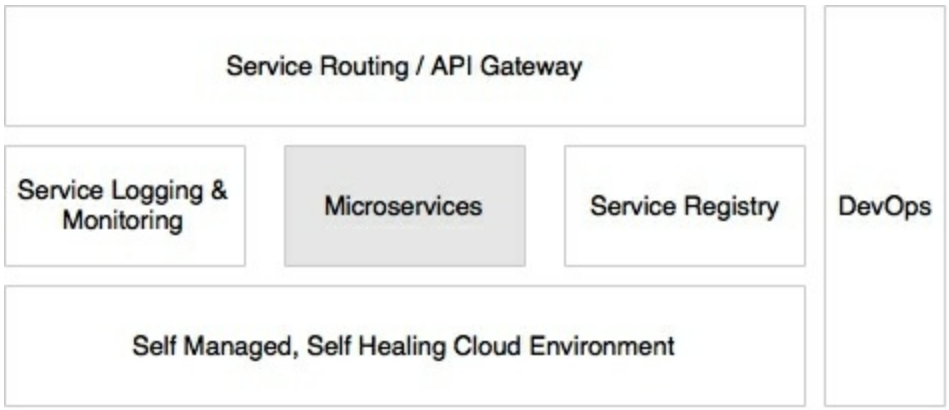


As indicated in the preceding diagram, automations are typically applied during the development, test, release, and deployment phases:

- The development phase is automated using version control tools such as Git together with **Continuous Integration** (**CI**) tools such as Jenkins, Travis CI, and so on. This may also include code quality checks and automation of unit testing. Automation of a full build on every code check-in is also achievable with microservices.
- The testing phase will be automated using testing tools such as Selenium, Cucumber, and other AB testing strategies. As microservices are aligned to business capabilities, the number of test cases to automate is fewer compared to monolithic applications, hence regression testing on every build also becomes possible.
- Infrastructure provisioning is done through container technologies such as Docker, together with release management tools such as Chef or Puppet, and configuration management tools such as Ansible. Automated deployments are handled using tools such as Spring Cloud, Kubernetes, Mesos, and Marathon.

Microservices with a supporting ecosystem

Most of the large-scale microservices implementations have a supporting ecosystem in place. The ecosystem capabilities include DevOps processes, centralized log management, service registry, API gateways, extensive monitoring, service routing, and flow control mechanisms.

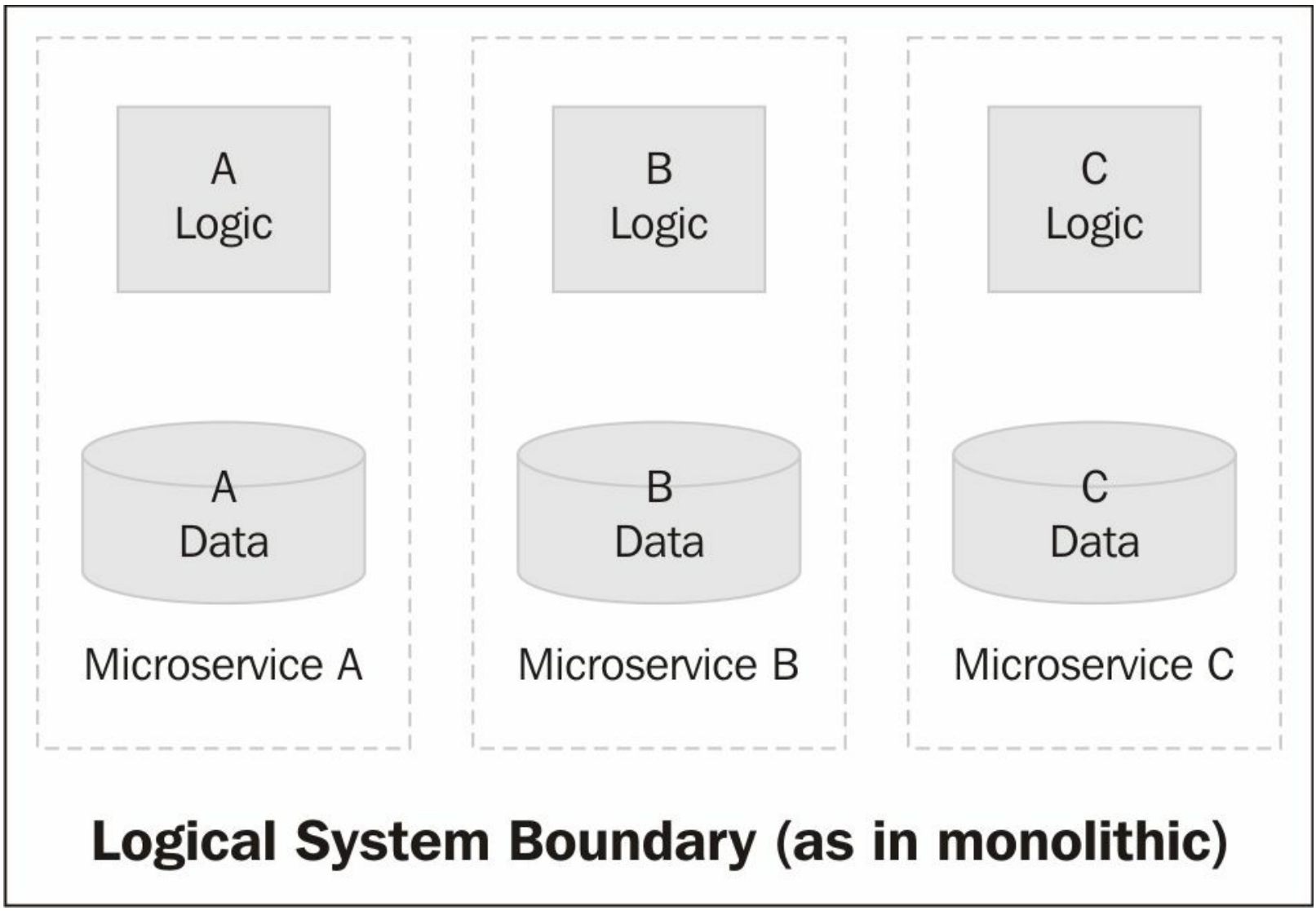


Microservices work well when supporting capabilities are in place, as represented in the preceding diagram.

Microservices are distributed and dynamic

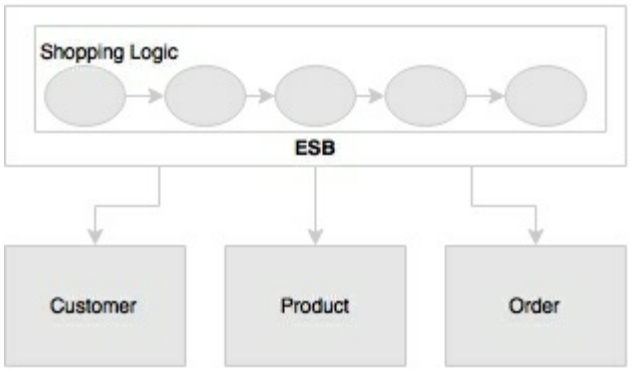
Successful microservices implementations encapsulate logic and data within the service. This results in two unconventional situations: distributed data and logic and decentralized governance.

Compared to traditional applications, which consolidate all logic and data into one application boundary, microservices decentralize data and logic. Each service, aligned to a specific business capability, owns its data and logic.



The outer rectangle in the preceding diagram implies the logical monolithic application boundary. When we migrate this to microservices, each microservice **A**, **B**, and **C** creates its own physical boundaries.

Microservices don't typically use centralized governance mechanisms the way they are used in SOA. One of the common characteristics of microservices implementations is that they do not relay on heavyweight enterprise-level products, such as **Enterprise Service Bus** (**ESB**). Instead, the business logic and intelligence are embedded as a part of the services themselves.



A typical SOA implementation is shown in the preceding diagram. Shopping logic is fully implemented in ESB by orchestrating different services exposed by Customer, Order, and Product. In the microservices approach, on the other hand, Shopping itself will run as a separate microservice, which interacts with Customer, Product, and Order in a fairly decoupled way.

SOA implementations heavily relay on static registry and repository configurations to manage services and other artifacts. Microservices bring a more dynamic nature into this. Hence, a static governance approach is seen as an overhead in maintaining up-to-date information. This is why most of the microservices implementations use automated mechanisms to build registry information dynamically from the runtime topologies.

Antifragility, fail fast, and self-healing

Antifragility is a technique successfully experimented at Netflix. It is one of the most powerful approaches to building fail-safe systems in modern software development.

Tip

The antifragility concept is introduced by Nassim Nicholas Taleb in his book *Antifragile: Things That Gain from Disorder*.

In the antifragility practice, software systems are consistently challenged. Software systems evolve through these challenges and, over a period of time, get better and better at withstanding these challenges. Amazon's GameDay exercise and Netflix' Simian Army are good examples of such antifragility experiments.

Fail fast is another concept used to build fault-tolerant, resilient systems. This philosophy advocates systems that expect failures versus building systems that never fail. Importance should be given to how quickly the system can fail and if it fails, how quickly it can recover from this failure. With this approach, the focus is shifted from **Mean Time Between Failures** (**MTBF**) to **Mean Time To Recover** (**MTTR**). A key advantage of this approach is that if something goes wrong, it kills itself, and downstream functions aren't stressed.

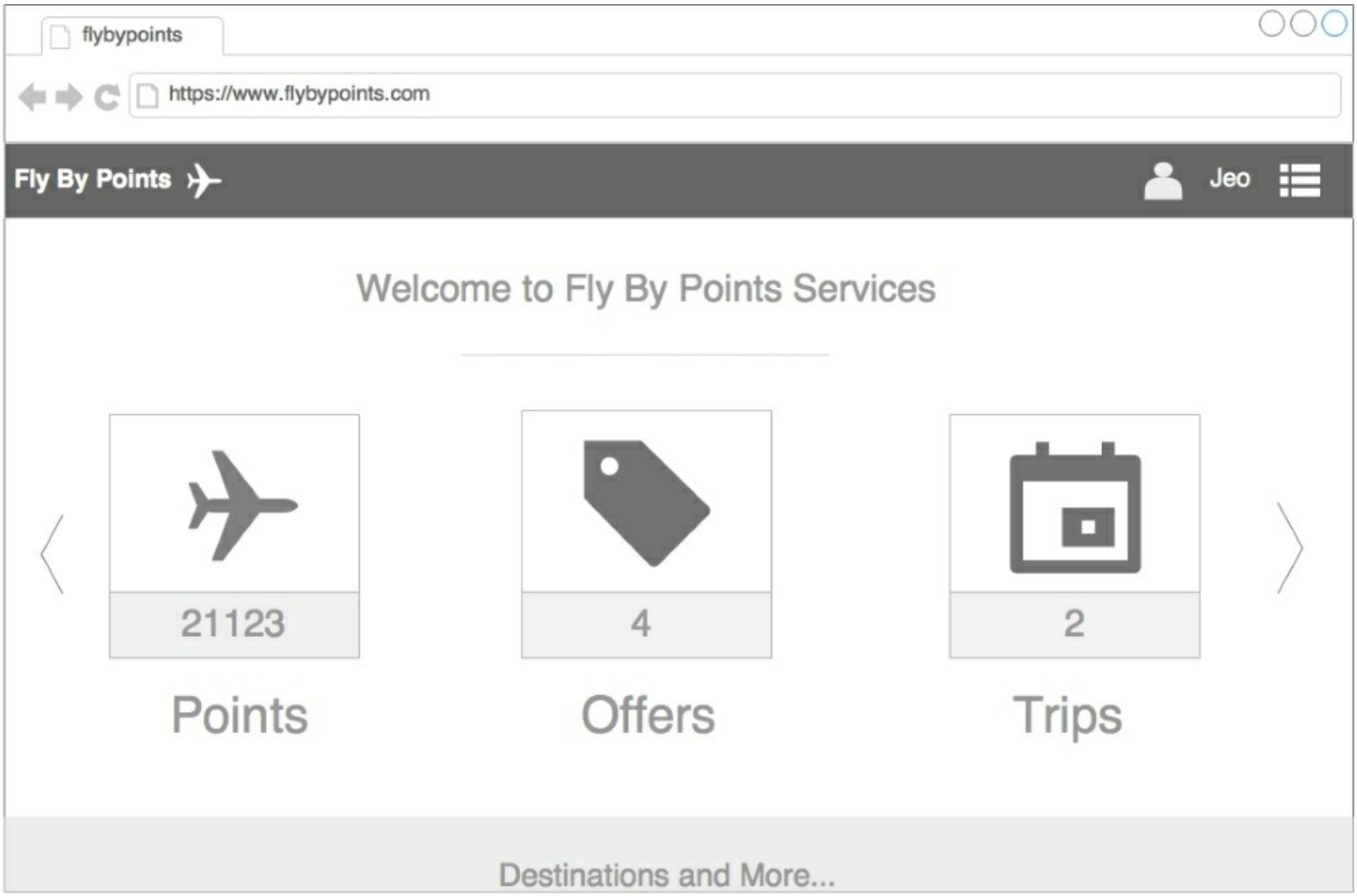
Self-healing is commonly used in microservices deployments, where the system automatically learns from failures and adjusts itself. These systems also prevent future failures.

Microservices examples

There is no "one size fits all" approach when implementing microservices. In this section, different examples are analyzed to crystalize the microservices concept.

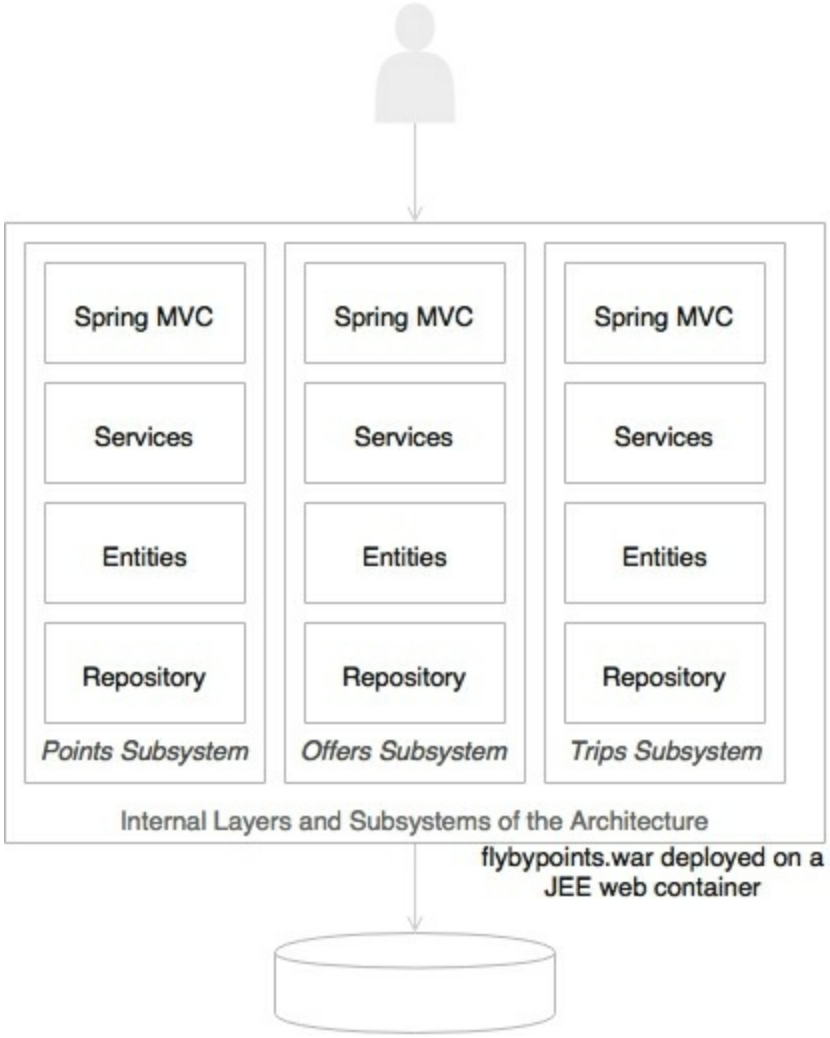
An example of a holiday portal

In the first example, we will review a holiday portal, **Fly By Points**. Fly By Points collects points that are accumulated when a customer books a hotel, flight, or car through the online website. When the customer logs in to the Fly By Points website, he/she is able to see the points accumulated, personalized offers that can be availed of by redeeming the points, and upcoming trips if any.

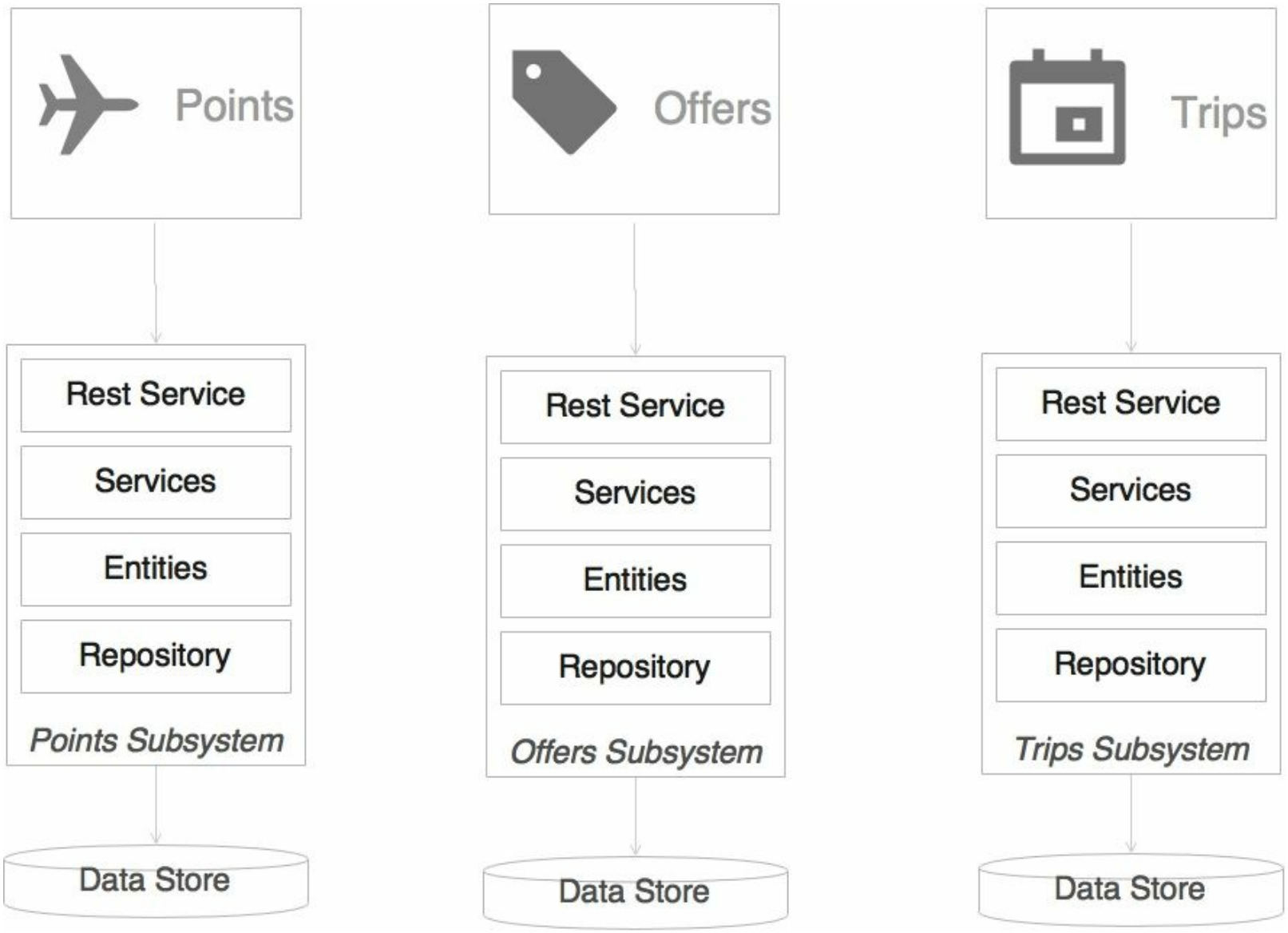


Let's assume that the preceding page is the home page after login. There are two upcoming trips for **Jeo**, four personalized offers, and 21,123 loyalty points. When the user clicks on each of the boxes, the details are queried and displayed.

The holiday portal has a Java Spring-based traditional monolithic application architecture, as shown in the following:



As shown in the preceding diagram, the holiday portal's architecture is web-based and modular, with a clear separation between layers. Following the usual practice, the holiday portal is also deployed as a single WAR file on a web server such as Tomcat. Data is stored on an all-encompassing backing relational database. This is a good fit for the purpose architecture when the complexities are few. As the business grows, the user base expands, and the complexity also increases. This results in a proportional increase in transaction volumes. At this point, enterprises should look to rearchitecting the monolithic application to microservices for better speed of delivery, agility, and manageability.



Examining the simple microservices version of this application, we can immediately note a few things in this architecture:

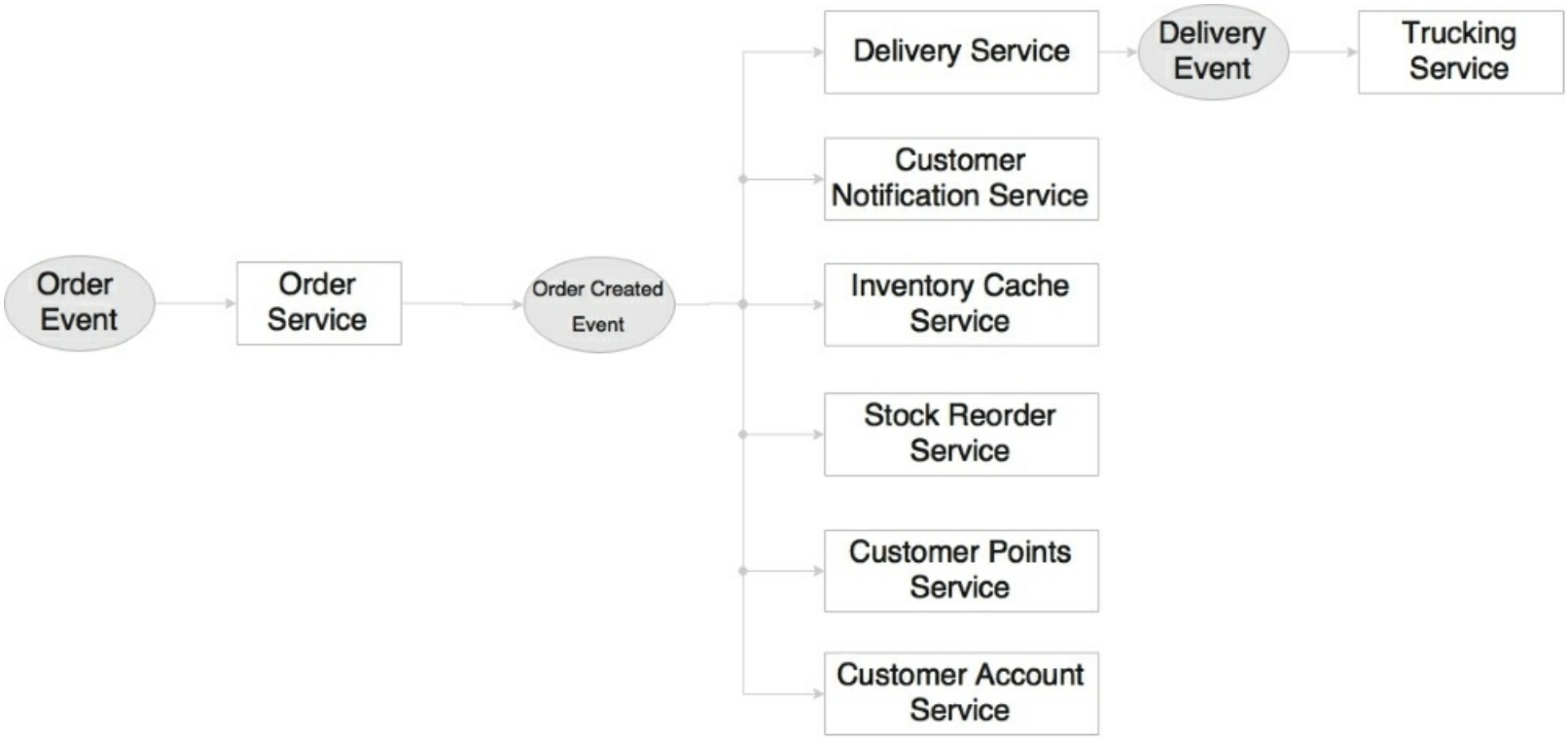
- Each subsystem has now become an independent system by itself, a microservice. There are three microservices representing three business functions: **Trips**, **Offers**, and **Points**. Each one has its internal data store and middle layer. The internal structure of each service remains the same.
- Each service encapsulates its own database as well as its own HTTP listener. As opposed to the previous model, there is no web server or WAR. Instead, each service has its own embedded HTTP listener, such as Jetty, Tomcat, and so on.
- Each microservice exposes a REST service to manipulate the resources/entity that belong to this service.

It is assumed that the presentation layer is developed using a client-side JavaScript MVC framework such as Angular JS. These client-side frameworks are capable of invoking REST calls directly.

When the web page is loaded, all the three boxes, Trips, Offers, and Points will be displayed with details such as points, the number of offers, and the number of trips. This will be done by each box independently making asynchronous calls to the respective backend microservices using REST. There is no dependency between the services at the service layer. When the user clicks on any of the boxes, the screen will be transitioned and will load the details of the item clicked on. This will be done by making another call to the respective microservice.

A microservice-based order management system

Let's examine another microservices example: an online retail website. In this case, we will focus more on the backend services, such as the Order Service which processes the Order Event generated when a customer places an order through the website:



This microservices system is completely designed based on reactive programming practices.

Tip

Read more on reactive programming at:

http://www.reactivemanifesto.org

When an event is published, a number of microservices are ready to kick-start upon receiving the event. Each one of them is independent and does not rely on other microservices. The advantage of this model is that we can keep adding or replacing microservices to achieve specific needs.

In the preceding diagram, there are eight microservices shown. The following activities take place upon the arrival of **Order Event**:

1. Order Service kicks off when Order Event is received. Order Service creates an order and saves the details to its own database.
2. If the order is successfully saved, Order Successful Event is created by Order Service and published.
3. A series of actions take place when Order Successful Event arrives.

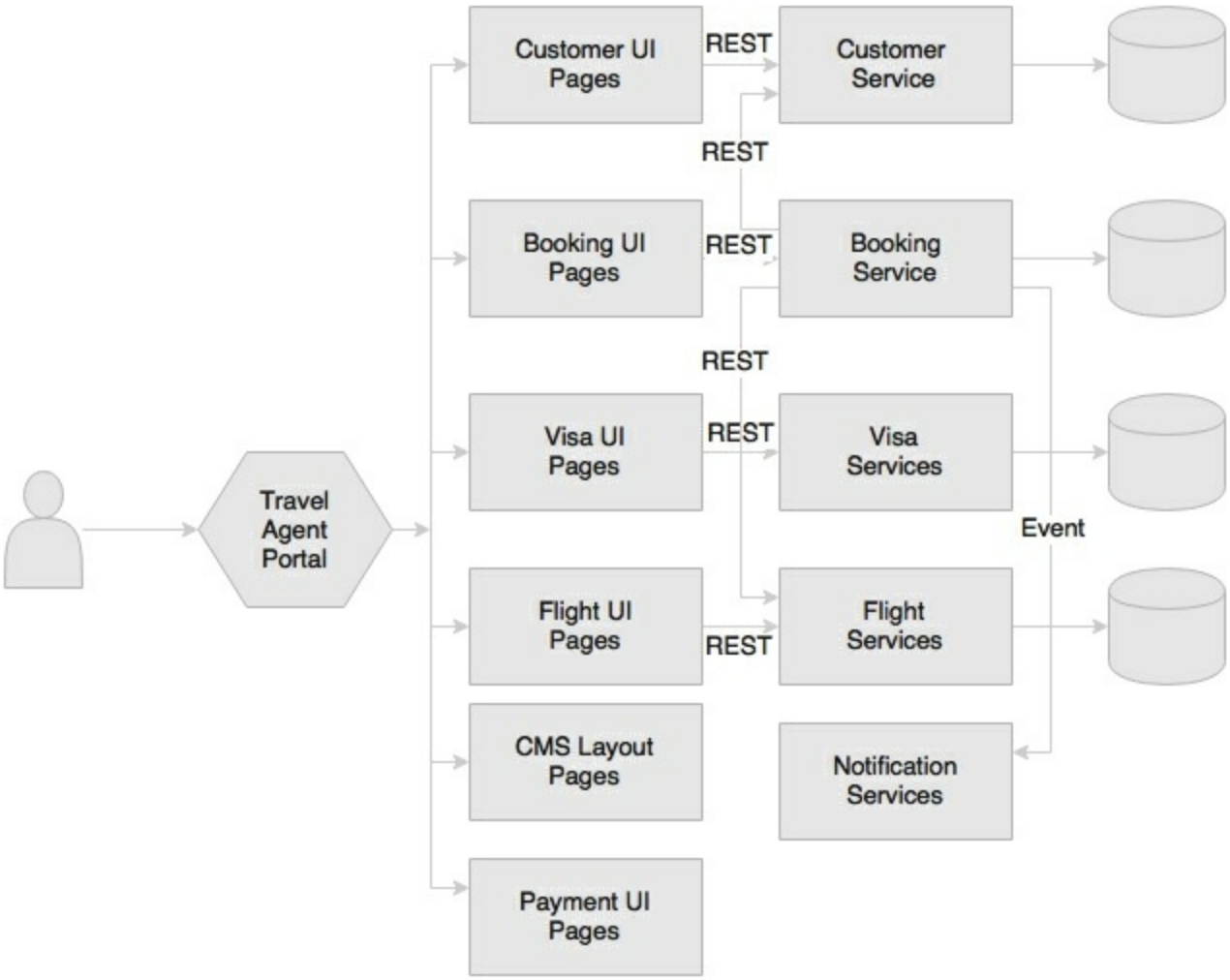
4. Delivery Service accepts the event and places Delivery Record to deliver the order to the customer. This, in turn, generates Delivery Event and publishes the event.
5. Trucking Service picks up Delivery Event and processes it. For instance, Trucking Service creates a trucking plan.
6. Customer Notification Service sends a notification to the customer informing the customer that an order is placed.
7. Inventory Cache Service updates the inventory cache with the available product count.
8. Stock Reorder Service checks whether the stock limits are adequate and generates Replenish Event if required.
9. Customer Points Service recalculates the customer's loyalty points based on this purchase.
10. **Customer Account Service** updates the order history in the customer's account.

In this approach, each service is responsible for only one function. Services accept and generate events. Each service is independent and is not aware of its neighborhood. Hence, the neighborhood can organically grow as mentioned in the honeycomb analogy. New services can be added as and when necessary. Adding a new service does not impact any of the existing services.

An example of a travel agent portal

This third example is a simple travel agent portal application. In this example, we will see both synchronous REST calls as well as asynchronous events.

In this case, the portal is just a container application with multiple menu items or links in the portal. When specific pages are requested—for example, when the menu or a link is clicked on—they will be loaded from the specific microservices.



When a customer requests a booking, the following events take place internally:

1. The travel agent opens the flight UI, searches for a flight, and identifies the right flight for the customer. Behind the scenes, the flight UI is loaded from the Flight microservice. The flight UI only interacts with its own backend APIs within the Flight microservice. In this case, it makes a REST call to the Flight microservice to load the flights to be displayed.
2. The travel agent then queries the customer details by accessing the customer UI. Similar to the flight UI, the customer UI is loaded from the Customer microservice. Actions in the customer UI will invoke REST calls on the Customer microservice. In this case, customer details are loaded by invoking appropriate APIs on the Customer microservice.
3. Then, the travel agent checks the visa details for the customer's eligibility to travel to the

selected country. This also follows the same pattern as mentioned in the previous two points.

4. Next, the travel agent makes a booking using the booking UI from the Booking microservice, which again follows the same pattern.
5. The payment pages are loaded from the Payment microservice. In general, the payment service has additional constraints such as PCIDSS compliance (protecting and encrypting data in motion and data at rest). The advantage of the microservices approach is that none of the other microservices need to be considered under the purview of PCIDSS as opposed to the monolithic application, where the complete application comes under the governing rules of PCIDSS. Payment also follows the same pattern as described earlier.
6. Once the booking is submitted, the Booking microservice calls the flight service to validate and update the flight booking. This orchestration is defined as part of the Booking microservice. Intelligence to make a booking is also held within the Booking microservice. As part of the booking process, it also validates, retrieves, and updates the Customer microservice.
7. Finally, the Booking microservice sends the Booking Event, which the Notification service picks up and sends a notification of to the customer.

The interesting factor here is that we can change the user interface, logic, and data of a microservice without impacting any other microservices.

This is a clean and neat approach. A number of portal applications can be built by composing different screens from different microservices, especially for different user communities. The overall behavior and navigation will be controlled by the portal application.

The approach has a number of challenges unless the pages are designed with this approach in mind. Note that the site layouts and static content will be loaded by the **Content Management System** (**CMS**) as layout templates. Alternately, this could be stored in a web server. The site layout may have fragments of UIs that will be loaded from the microservices at runtime.

Microservices benefits

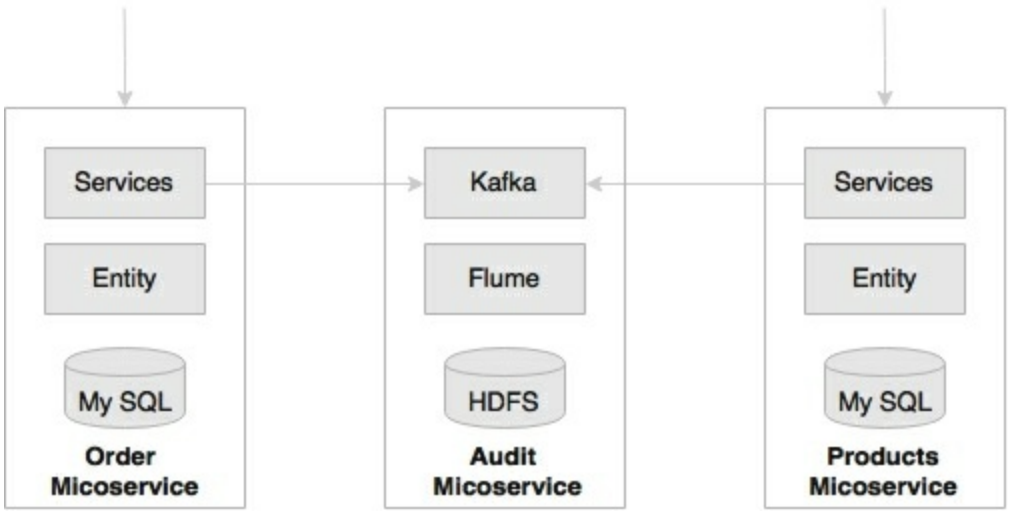
Microservices offer a number of benefits over the traditional multitier, monolithic architectures. This section explains some key benefits of the microservices architecture approach.

Supports polyglot architecture

With microservices, architects and developers can choose fit for purpose architectures and technologies for each microservice. This gives the flexibility to design better-fit solutions in a more cost-effective way.

As microservices are autonomous and independent, each service can run with its own architecture or technology or different versions of technologies.

The following shows a simple, practical example of a polyglot architecture with microservices.



There is a requirement to audit all system transactions and record transaction details such as request and response data, the user who initiated the transaction, the service invoked, and so on.

As shown in the preceding diagram, while core services such as the Order and Products microservices use a relational data store, the Audit microservice persists data in Hadoop File System (HDFS). A relational data store is neither ideal nor cost effective in storing large data volumes such as in the case of audit data. In the monolithic approach, the application generally uses a shared, single database that stores Order, Products, and Audit data.

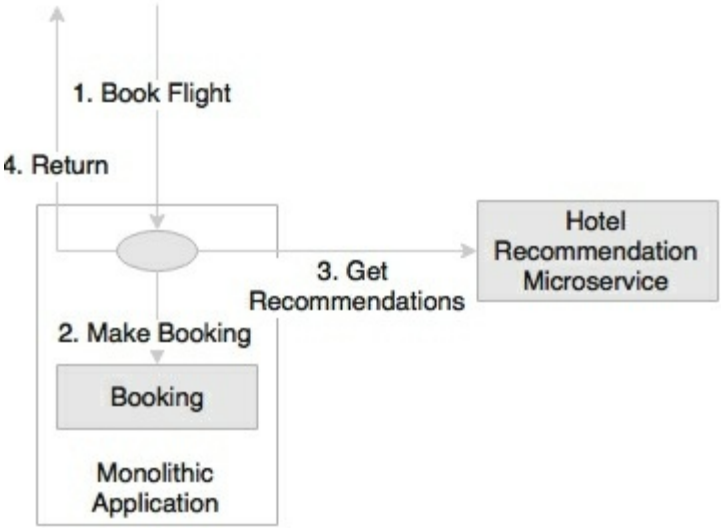
In this example, the audit service is a technical microservice using a different architecture. Similarly, different functional services could also use different architectures.

In another example, there could be a Reservation microservice running on Java 7, while a Search microservice could be running on Java 8. Similarly, an Order microservice could be written on Erlang, whereas a Delivery microservice could be on the Go language. None of these are possible with a monolithic architecture.

Enabling experimentation and innovation

Modern enterprises are thriving towards quick wins. Microservices are one of the key enablers for enterprises to do disruptive innovation by offering the ability to experiment and fail fast.

As services are fairly simple and smaller in size, enterprises can afford to experiment new processes, algorithms, business logics, and so on. With large monolithic applications, experimentation was not easy; nor was it straightforward or cost effective. Businesses had to spend huge money to build or change an application to try out something new. With microservices, it is possible to write a small microservice to achieve the targeted functionality and plug it into the system in a reactive style. One can then experiment with the new function for a few months, and if the new microservice does not work as expected, we can change or replace it with another one. The cost of change will be considerably less compared to that of the monolithic approach.



In another example of an airline booking website, the airline wants to show personalized hotel recommendations in their booking page. The recommendations must be displayed on the booking confirmation page.

As shown in the preceding diagram, it is convenient to write a microservice that can be plugged into the monolithic applications booking flow rather than incorporating this requirement in the monolithic application itself. The airline may choose to start with a simple recommendation service and keep replacing it with newer versions till it meets the required accuracy.

Elastically and selectively scalable

As microservices are smaller units of work, they enable us to implement selective scalability.

Scalability requirements may be different for different functions in an application. A monolithic application, packaged as a single WAR or an EAR, can only be scaled as a whole. An I/O-intensive function when streamed with high velocity data could easily bring down the service levels of the entire application.

In the case of microservices, each service could be independently scaled up or down. As scalability can be selectively applied at each service, the cost of scaling is comparatively less with the microservices approach.

In practice, there are many different ways available to scale an application and is largely subject to the architecture and behavior of the application. **Scale Cube** defines primarily three approaches to scaling an application:

- Scaling the *x* axis by horizontally cloning the application
- Scaling the *y* axis by splitting different functionality
- Scaling the *z* axis by partitioning or sharding the data

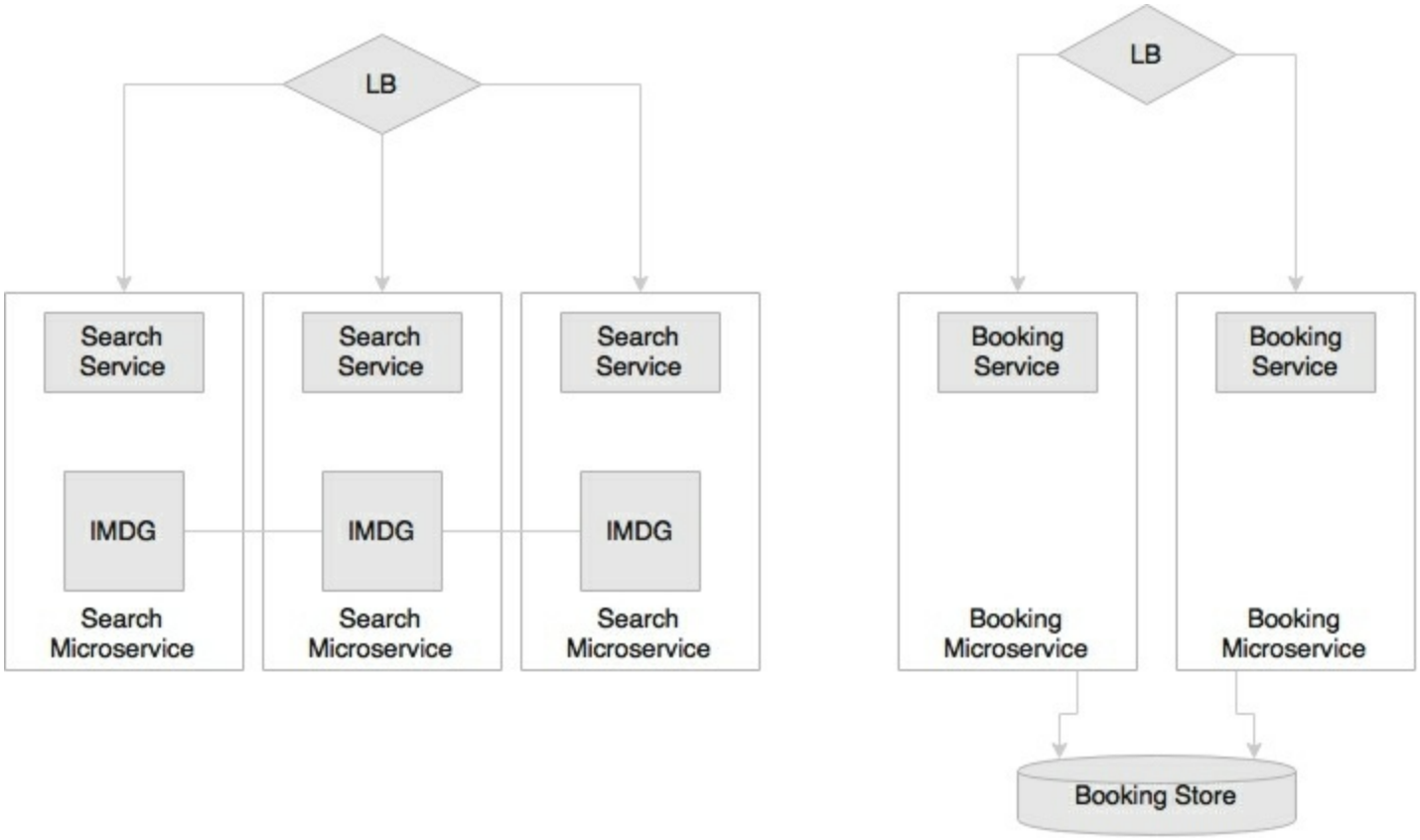
Tip

Read more about Scale Cube in the following site:

http://theartofscalability.com/

When *y* axis scaling is applied to monolithic applications, it breaks the monolithic to smaller units aligned with business functions. Many organizations successfully applied this technique to move away from monolithic applications. In principle, the resulting units of functions are in line with the microservices characteristics.

For instance, in a typical airline website, statistics indicate that the ratio of flight searching to flight booking could be as high as 500:1. This means one booking transaction for every 500 search transactions. In this scenario, the search needs 500 times more scalability than the booking function. This is an ideal use case for selective scaling.



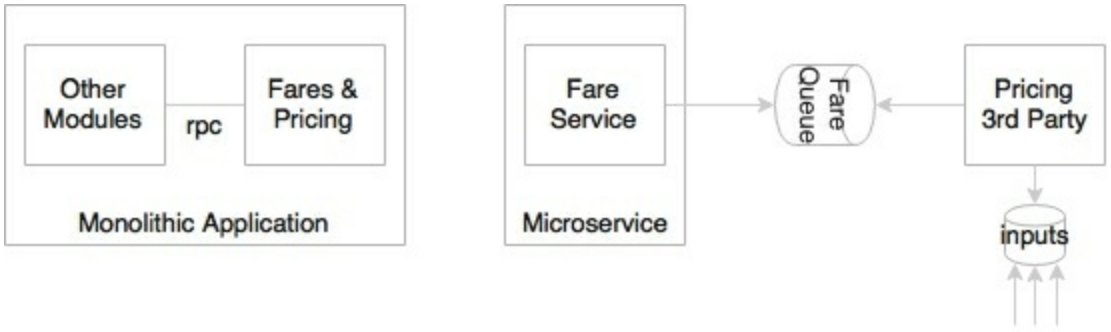
The solution is to treat search requests and booking requests differently. With a monolithic architecture, this is only possible with *z* scaling in the scale cube. However, this approach is expensive because in the *z* scale, the entire code base is replicated.

In the preceding diagram, Search and Booking are designed as different microservices so that Search can be scaled differently from Booking. In the diagram, Search has three instances, and Booking has two instances. Selective scalability is not limited to the number of instances, as shown in the diagram, but also in the way in which the microservices are architected. In the case of Search, an **in-memory data grid** (**IMDG**) such as Hazelcast can be used as the data store. This will further increase the performance and scalability of Search. When a new Search microservice instance is instantiated, an additional IMDG node is added to the IMDG cluster. Booking does not require the same level of scalability. In the case of Booking, both instances of the Booking microservice are connected to the same instance of the database.

Allowing substitution

Microservices are self-contained, independent deployment modules enabling the substitution of one microservice with another similar microservice.

Many large enterprises follow buy-versus-build policies to implement software systems. A common scenario is to build most of the functions in house and buy certain niche capabilities from specialists outside. This poses challenges in traditional monolithic applications as these application components are highly cohesive. Attempting to plug in third-party solutions to the monolithic applications results in complex integrations. With microservices, this is not an afterthought. Architecturally, a microservice can be easily replaced by another microservice developed either in-house or even extended by a microservice from a third party.



A pricing engine in the airline business is complex. Fares for different routes are calculated using complex mathematical formulas known as the pricing logic. Airlines may choose to buy a pricing engine from the market instead of building the product in house. In the monolithic architecture, Pricing is a function of Fares and Booking. In most cases Pricing, Fares, and Booking are hardwired, making it almost impossible to detach.

In a well-designed microservices system, Booking, Fares, and Pricing would be independent microservices. Replacing the Pricing microservice will have only a minimal impact on any other services as they are all loosely coupled and independent. Today, it could be a third-party service; tomorrow, it could be easily substituted by another third-party or home-grown service.

Enabling to build organic systems

Microservices help us build systems that are organic in nature. This is significantly important when migrating monolithic systems gradually to microservices.

Organic systems are systems that grow laterally over a period of time by adding more and more functions to it. In practice, an application grows unimaginably large in its lifespan, and in most cases, the manageability of the application reduces dramatically over this same period of time.

Microservices are all about independently manageable services. This enable us to keep adding more and more services as the need arises with minimal impact on the existing services. Building such systems does not need huge capital investments. Hence, businesses can keep building as part of their operational expenditure.

A loyalty system in an airline was built years ago, targeting individual passengers. Everything was fine until the airline started offering loyalty benefits to their corporate customers. Corporate customers are individuals grouped under corporations. As the current systems core data model is flat, targeting individuals, the corporate environment needs a fundamental change in the core data model, and hence huge reworking, to incorporate this requirement.



As shown in the preceding diagram, in a microservices-based architecture, customer information would be managed by the Customer microservice and loyalty by the Loyalty Points microservice.

In this situation, it is easy to add a new Corporate Customer microservice to manage corporate customers. When a corporation is registered, individual members will be pushed to the Customer microservice to manage them as usual. The Corporate Customer microservice provides a corporate view by aggregating data from the Customer microservice. It will also provide services to support corporate-specific business rules. With this approach, adding new services will have only a minimal impact on the existing services.

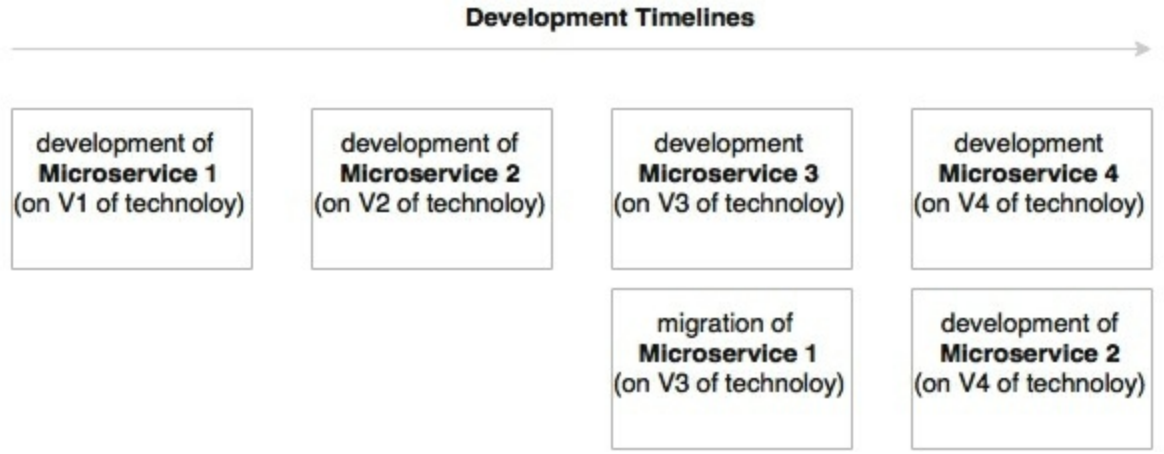
Helping reducing technology debt

As microservices are smaller in size and have minimal dependencies, they allow the migration of services that use end-of-life technologies with minimal cost.

Technology changes are one of the barriers in software development. In many traditional monolithic applications, due to the fast changes in technologies, today's next-generation applications could easily become legacy even before their release to production. Architects and developers tend to add a lot of protection against technology changes by adding layers of abstractions. However, in reality, this approach does not solve the issue but, instead, results in over-engineered systems. As technology upgrades are often risky and expensive with no direct returns to business, the business may not be happy to invest in reducing the technology debt of the applications.

With microservices, it is possible to change or upgrade technology for each service individually rather than upgrading an entire application.

Upgrading an application with, for instance, five million lines written on EJB 1.1 and Hibernate to the Spring, JPA, and REST services is almost similar to rewriting the entire application. In the microservices world, this could be done incrementally.



As shown in the preceding diagram, while older versions of the services are running on old versions of technologies, new service developments can leverage the latest technologies. The cost of migrating microservices with end-of-life technologies is considerably less compared to enhancing monolithic applications.

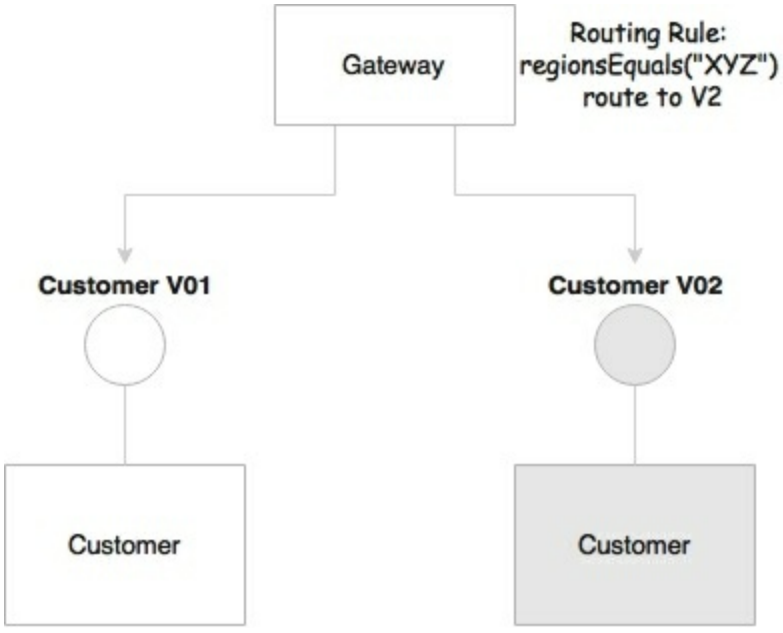
Allowing the coexistence of different versions

As microservices package the service runtime environment along with the service itself, this enables having multiple versions of the service to coexist in the same environment.

There will be situations where we will have to run multiple versions of the same service at the same time. Zero downtime promote, where one has to gracefully switch over from one version to another, is one example of a such a scenario as there will be a time window where both services will have to be up and running simultaneously. With monolithic applications, this is a complex procedure because upgrading new services in one node of the cluster is cumbersome as, for instance, this could lead to class loading issues. A canary release, where a new version is only released to a few users to validate the new service, is another example where multiple versions of the services have to coexist.

With microservices, both these scenarios are easily manageable. As each microservice uses independent environments, including service listeners such as Tomcat or Jetty embedded, multiple versions can be released and gracefully transitioned without many issues. When consumers look up services, they look for specific versions of services. For example, in a canary release, a new user interface is released to user A. When user A sends a request to the microservice, it looks up the canary release version, whereas all other users will continue to look up the last production version.

Care needs to be taken at the database level to ensure the database design is always backward compatible to avoid breaking the changes.



As shown in the preceding diagram, version 1 and 2 of the **Customer** service can coexist as they are not interfering with each other, given their respective deployment environments. Routing rules can be set at the gateway to divert traffic to specific instances, as shown in the diagram. Alternatively, clients can request specific versions as part of the request itself. In the diagram, the gateway selects

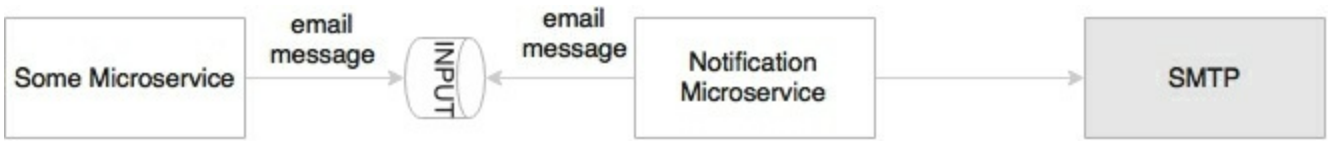
the version based on the region from which the request is originated.

Supporting the building of self-organizing systems

Microservices help us build self-organizing systems. A self-organizing system support will automate deployment, be resilient, and exhibit self-healing and self-learning capabilities.

In a well-architected microservices system, a service is unaware of other services. It accepts a message from a selected queue and processes it. At the end of the process, it may send out another message, which triggers other services. This allows us to drop any service into the ecosystem without analyzing the impact on the overall system. Based on the input and output, the service will self-organize into the ecosystem. No additional code changes or service orchestration is required. There is no central brain to control and coordinate the processes.

Imagine an existing notification service that listens to an **INPUT** queue and sends notifications to an **SMTP** server, as shown in the following figure:



Let's assume, later, a personalization engine, responsible for changing the language of the message to the customer's native language, needs to be introduced to personalize messages before sending them to the customer, the personalization engine is responsible for changing the language of the message to the customer's native language.



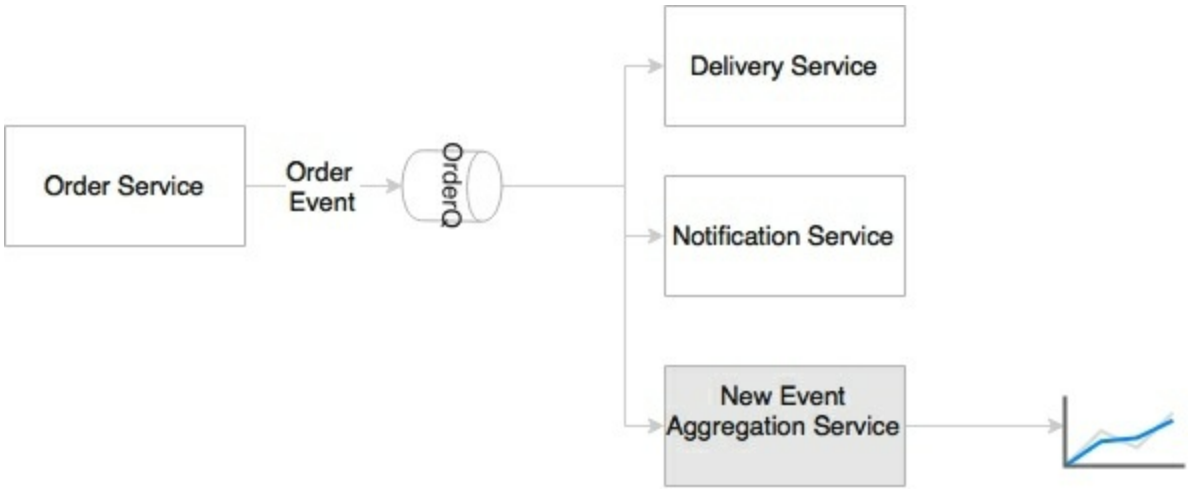
With microservices, a new personalization microservice will be created to do this job. The input queue will be configured as INPUT in an external configuration server, and the personalization service will pick up the messages from the INPUT queue (earlier, this was used by the notification service) and send the messages to the OUTPUT queue after completing process. The notification service will read messages from OUTPUT and send to SMTP. From the very next moment onward, the system automatically adopts this new message flow.

Supporting event-driven architecture

Microservices enable us to develop transparent software systems. Traditional systems communicate with each other through native protocols and hence behave like a black box application. Business events and system events, unless published explicitly, are hard to understand and analyze. Modern applications require data for business analysis, to understand dynamic system behaviors, and analyze market trends, and they also need to respond to real-time events. Events are useful mechanisms for data extraction.

A well-architected microservice always works with events for both input and output. These events can be tapped by any service. Once extracted, events can be used for a variety of use cases.

For example, the business wants to see the velocity of orders categorized by product type in real time. In a monolithic system, we need to think about how to extract these events. This may impose changes in the system.



In the microservices world, **Order Event** is already published whenever an order is created. This means that it is just a matter of adding a new service to subscribe to the same topic, extract the event, perform the requested aggregations, and push another event for the dashboard to consume.

Enabling DevOps

Microservices are one of the key enablers of DevOps. DevOps is widely adopted as a practice in many enterprises, primarily to increase the speed of delivery and agility. A successful adoption of DevOps requires cultural changes, process changes, as well as architectural changes. DevOps advocates to have agile development, high-velocity release cycles, automatic testing, automatic infrastructure provisioning, and automated deployment.

Automating all these processes is extremely hard to achieve with traditional monolithic applications. Microservices are not the ultimate answer, but microservices are at the center stage in many DevOps implementations. Many DevOps tools and techniques are also evolving around the use of microservices.

Consider a monolithic application that takes hours to complete a full build and 20 to 30 minutes to start the application; one can see that this kind of application is not ideal for DevOps automation. It is hard to automate continuous integration on every commit. As large, monolithic applications are not automation friendly, continuous testing and deployments are also hard to achieve.

On the other hand, small footprint microservices are more automation-friendly and therefore can more easily support these requirements.

Microservices also enable smaller, focused agile teams for development. Teams will be organized based on the boundaries of microservices.

Relationship with other architecture styles

Now that we have seen the characteristics and benefits of microservices, in this section, we will explore the relationship of microservices with other closely related architecture styles such as SOA and Twelve-Factor Apps.

Relations with SOA

SOA and microservices follow similar concepts. Earlier in this chapter, we discussed that microservices are evolved from SOA, and many service characteristics are common in both approaches.

However, are they the same or are they different?

As microservices are evolved from SOA, many characteristics of microservices are similar to SOA. Let's first examine the definition of SOA.

The definition of SOA from *The Open Group* consortium is as follows:

> *"Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.*
>
> *A service:*
>
> *Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)*
>
> *It is self-contained.*
>
> *It may be composed of other services.*
>
> *It is a "black box" to consumers of the service."*

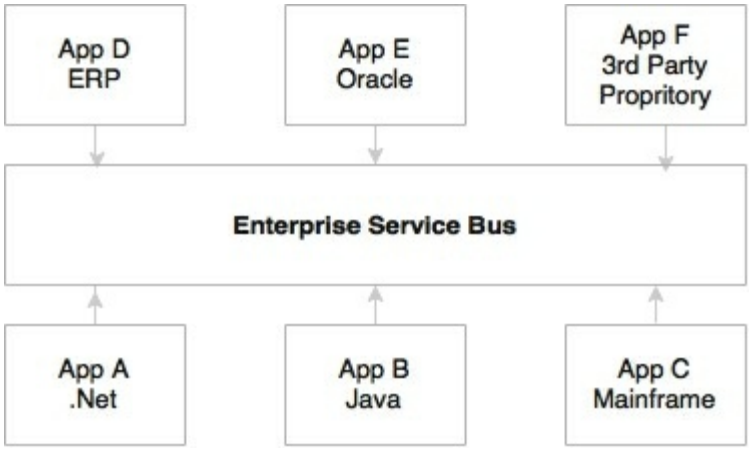
We observed similar aspects in microservices as well. So, in what way are microservices different? The answer is: it depends.

The answer to the previous question could be yes or no, depending upon the organization and its adoption of SOA. SOA is a broader term, and different organizations approached SOA differently to solve different organizational problems. The difference between microservices and SOA is in a way based on how an organization approaches SOA.

In order to get clarity, a few cases will be examined.

Service-oriented integration

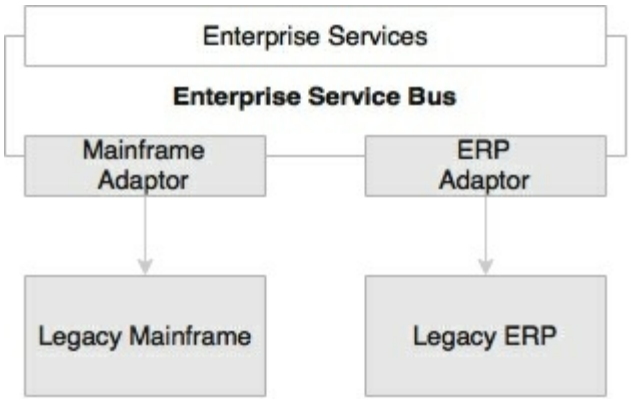
Service-oriented integration refers to a service-based integration approach used by many organizations.



Many organizations would have used SOA primarily to solve their integration complexities, also known as integration spaghetti. Generally, this is termed as **Service-Oriented Integration** (**SOI**). In such cases, applications communicate with each other through a common integration layer using standard protocols and message formats such as SOAP/XML-based web services over HTTP or JMS. These types of organizations focus on **Enterprise Integration Patterns** (**EIP**) to model their integration requirements. This approach strongly relies on heavyweight ESB such as TIBCO Business Works, WebSphere ESB, Oracle ESB, and the likes. Most ESB vendors also packed a set of related products such as rules engines, business process management engines, and so on as an SOA suite. Such organizations' integrations are deeply rooted into their products. They either write heavy orchestration logic in the ESB layer or the business logic itself in the service bus. In both cases, all enterprise services are deployed and accessed via ESB. These services are managed through an enterprise governance model. For such organizations, microservices are altogether different from SOA.

Legacy modernization

SOA is also used to build service layers on top of legacy applications.

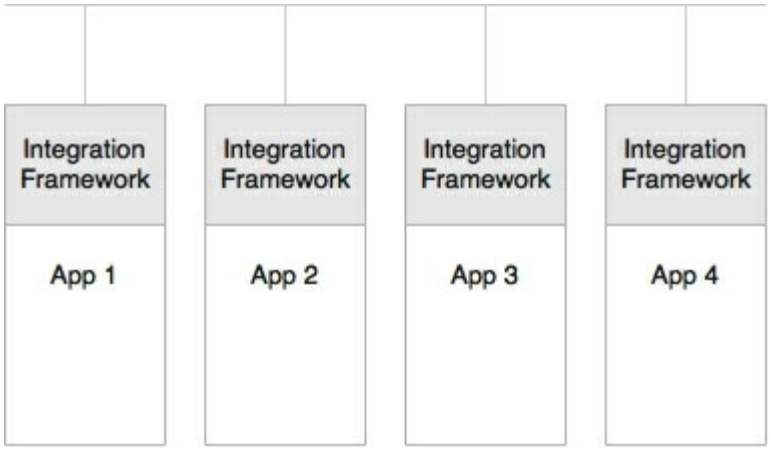


Another category of organizations would use SOA in transformation projects or legacy modernization projects. In such cases, the services are built and deployed in the ESB layer connecting to backend

systems using ESB adapters. For these organizations, microservices are different from SOA.

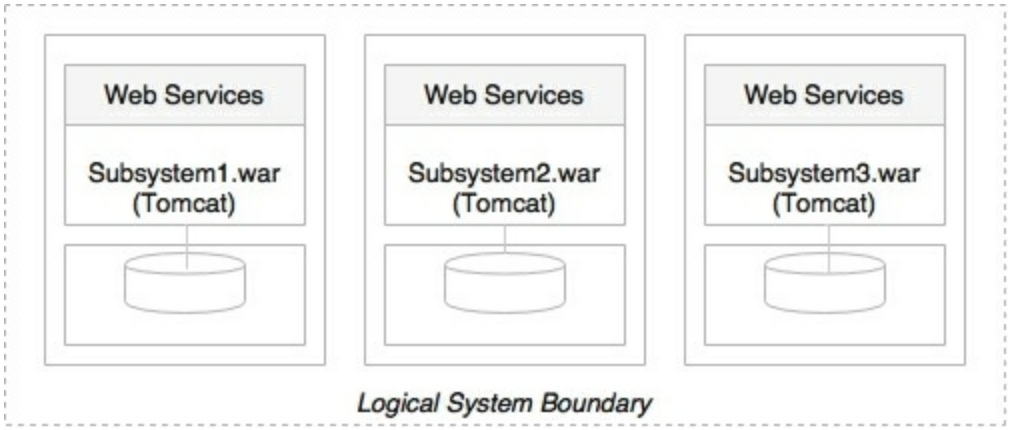
Service-oriented application

Some organizations adopt SOA at an application level.



In this approach, lightweight integration frameworks, such as Apache Camel or Spring Integration, are embedded within applications to handle service-related cross-cutting capabilities such as protocol mediation, parallel execution, orchestration, and service integration. As some of the lightweight integration frameworks have native Java object support, such applications would even use native **Plain Old Java Objects** (**POJO**) services for integration and data exchange between services. As a result, all services have to be packaged as one monolithic web archive. Such organizations could see microservices as the next logical step of their SOA.

Monolithic migration using SOA



The last possibility is transforming a monolithic application into smaller units after hitting the breaking point with the monolithic system. They would break the application into smaller, physically deployable subsystems, similar to the *y* axis scaling approach explained earlier, and deploy them as web archives on web servers or as JARs deployed on some home-grown containers. These subsystems as service would use web services or other lightweight protocols to exchange data

between services. They would also use SOA and service design principles to achieve this. For such organizations, they may tend to think that microservices are the same old wine in a new bottle.

Relations with Twelve-Factor apps

Cloud computing is one of the rapidly evolving technologies. Cloud computing promises many benefits, such as cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers offering different services. They lower the cost models to make it more attractive to the enterprises. Different cloud providers such as AWS, Microsoft, Rackspace, IBM, Google, and so on use different tools, technologies, and services. On the other hand, enterprises are aware of this evolving battlefield and, therefore, they are looking for options for de-risking from lockdown to a single vendor.

Many organizations do lift and shift their applications to the cloud. In such cases, the applications may not realize all the benefits promised by cloud platforms. Some applications need to undergo overhaul, whereas some may need minor tweaking before moving to cloud. This by and large depends upon how the application is architected and developed.

For example, if the application has its production database server URLs hardcoded as part of the applications WAR, it needs to be modified before moving the application to cloud. In the cloud, the infrastructure is transparent to the application, and especially, the physical IP addresses cannot be assumed.

How do we ensure that an application, or even microservices, can run seamlessly across multiple cloud providers and take advantages of cloud services such as elasticity?

It is important to follow certain principles while developing cloud native applications.

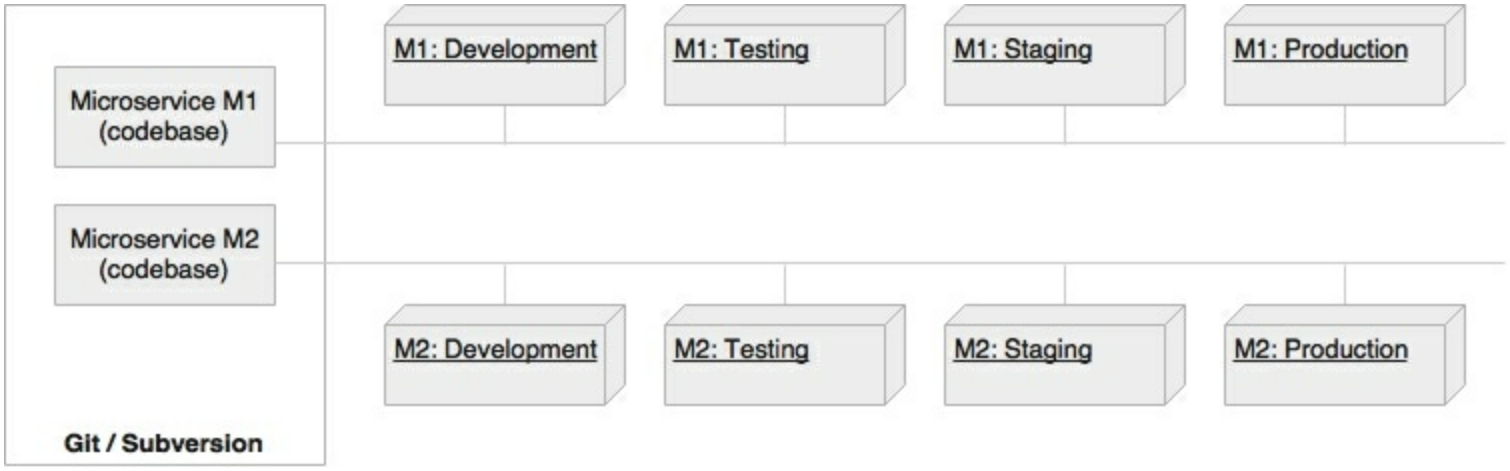
Tip

Cloud native is a term used for developing applications that can work efficiently in a cloud environment, understanding and utilizing cloud behaviors such as elasticity, utilization based charging, fail aware, and so on.

Twelve-Factor App, forwarded by Heroku, is a methodology describing the characteristics expected from modern cloud-ready applications. Twelve-Factor App is equally applicable for microservices as well. Hence, it is important to understand Twelve-Factor App.

A single code base

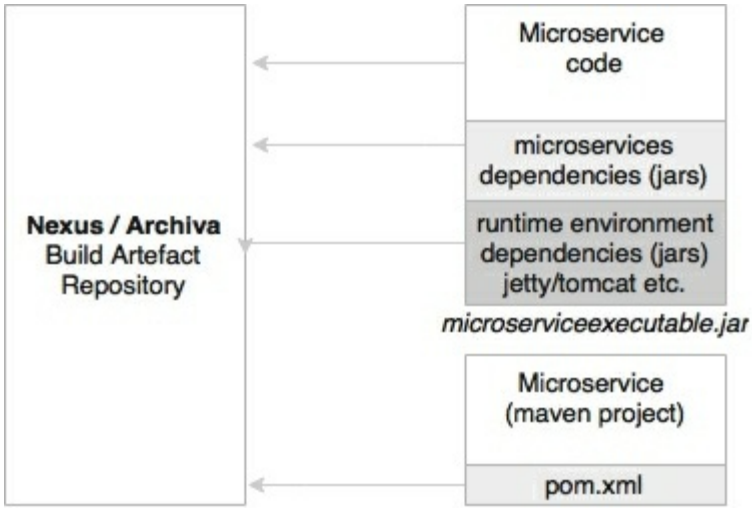
The code base principle advises that each application has a single code base. There can be multiple instances of deployment of the same code base, such as development, testing, and production. Code is typically managed in a source control system such as Git, Subversion, and so on.



Extending the same philosophy for microservices, each microservice should have its own code base, and this code base is not shared with any other microservice. It also means that one microservice has exactly one code base.

Bundling dependencies

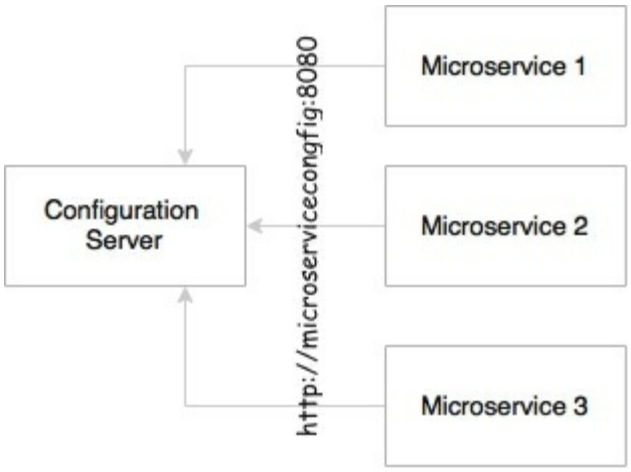
As per this principle, all applications should bundle their dependencies along with the application bundle. With build tools such as Maven and Gradle, we explicitly manage dependencies in a `pom.xml` or the `.gradle` file and link them using a central build artifact repository such as Nexus or Archiva. This ensures that the versions are managed correctly. The final executables will be packaged as a WAR file or an executable JAR file, embedding all the dependencies.



In the context of microservices, this is one of the fundamental principles to be followed. Each microservice should bundle all the required dependencies and execution libraries such as the HTTP listener and so on in the final executable bundle.

Externalizing configurations

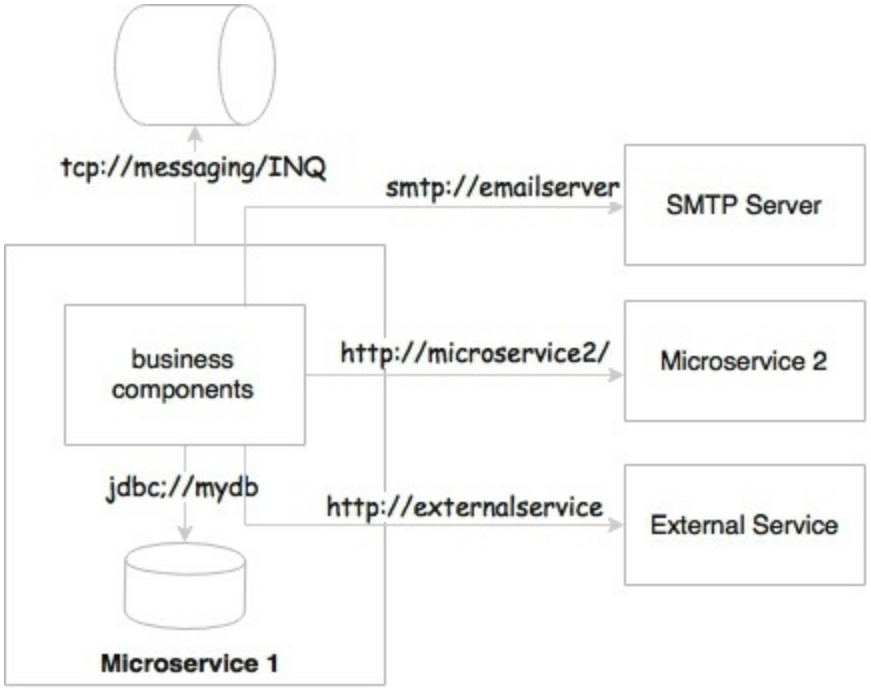
This principle advises the externalization of all configuration parameters from the code. An application's configuration parameters vary between environments, such as support to the e-mail IDs or URL of an external system, username, passwords, queue name, and so on. These will be different for development, testing, and production. All service configurations should be externalized.



The same principle is obvious for microservices as well. The microservices configuration parameters should be loaded from an external source. This will also help to automate the release and deployment process as the only difference between these environments is the configuration parameters.

Backing services are addressable

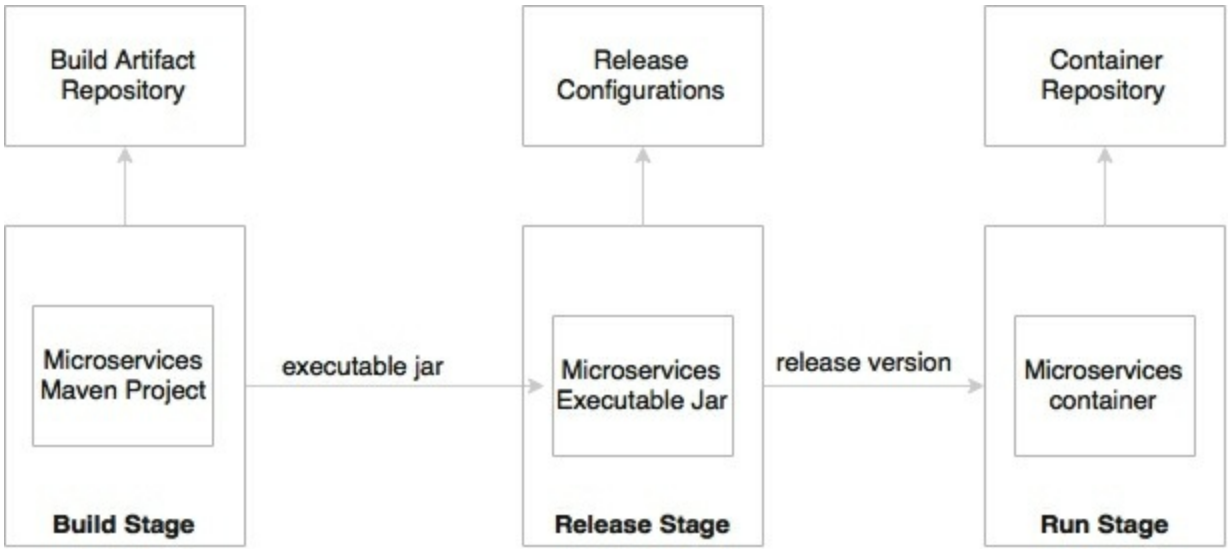
All backing services should be accessible through an addressable URL. All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening or sending messages to a messaging system, sending an e-mail, persisting data to database, and so on. All these services should be reachable through a URL without complex communication requirements.



In the microservices world, microservices either talk to a messaging system to send or receive messages, or they could accept or send messages to other service APIs. In a regular case, these are either HTTP endpoints using REST and JSON or TCP- or HTTP-based messaging endpoints.

Isolation between build, release, and run

This principle advocates a strong isolation between the build, release, and run stages. The build stage refers to compiling and producing binaries by including all the assets required. The release stage refers to combining binaries with environment-specific configuration parameters. The run stage refers to running application on a specific execution environment. The pipeline is unidirectional, so it is not possible to propagate changes from the run stages back to the build stage. Essentially, it also means that it is not recommended to do specific builds for production; rather, it has to go through the pipeline.



In microservices, the build will create executable JAR files, including the service runtime such as an HTTP listener. During the release phase, these executables will be combined with release configurations such as production URLs and so on and create a release version, most probably as a container similar to Docker. In the run stage, these containers will be deployed on production via a container scheduler.

Stateless, shared nothing processes

This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it is fault tolerant and can be scaled out easily.

All microservices should be designed as stateless functions. If there is any requirement to store a state, it should be done with a backing database or in an in-memory cache.

Exposing services through port bindings

A Twelve-Factor application is expected to be self-contained. Traditionally, applications are deployed to a server: a web server or an application server such as Apache Tomcat or JBoss. A Twelve-Factor application does not rely on an external web server. HTTP listeners such as Tomcat or Jetty have to be embedded in the service itself.

Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservices embed service listeners as a part of the service itself.

Concurrency to scale out

This principle states that processes should be designed to scale out by replicating the processes. This is in addition to the use of threads within the process.

In the microservices world, services are designed to scale out rather than scale up. The *x* axis scaling technique is primarily used for a scaling service by spinning up another identical service instance. The services can be elastically scaled or shrunk based on the traffic flow. Further to this, microservices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

Disposability with minimal overhead

This principle advocates building applications with minimal startup and shutdown times with graceful shutdown support. In an automated deployment environment, we should be able bring up or bring down instances as quick as possible. If the application's startup or shutdown takes considerable time, it will have an adverse effect on automation. The startup time is proportionally related to the size of the application. In a cloud environment targeting auto-scaling, we should be able to spin up new instance quickly. This is also applicable when promoting new versions of services.

In the microservices context, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown time. Microservices also should consider a lazy loading of objects and data.

Development and production parity

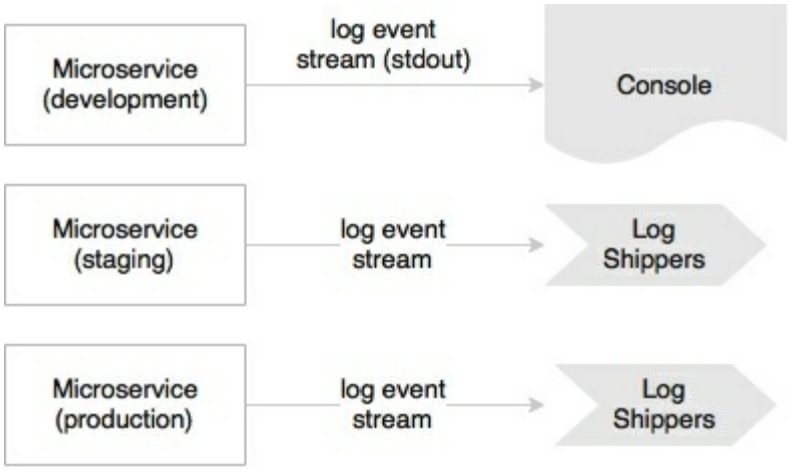
This principle states the importance of keeping development and production environments as identical as possible. For example, let's consider an application with multiple services or processes, such as a job scheduler service, cache services, and one or more application services. In a development environment, we tend to run all of them on a single machine, whereas in production, we will facilitate independent machines to run each of these processes. This is primarily to manage the cost of infrastructure. The downside is that if production fails, there is no identical environment to re-produce and fix the issues.

Not only is this principle valid for microservices, but it is also applicable to any application development.

Externalizing logs

A Twelve-Factor application never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os. If the I/Os are not fast enough in a given infrastructure, it could create a bottleneck. The solution to this is to use a centralized logging framework. Splunk, Greylog, Logstash, Logplex, and Loggly are some examples of log shipping and analysis tools. The recommended approach is to ship logs to a central repository by tapping the logback appenders and write to one of the shippers' endpoints.

In a microservices ecosystem, this is very important as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services.



In development, the microservice may direct the log stream to `stdout`, whereas in production, these streams will be captured by the log shippers and sent to a central log service for storage and analysis.

Package admin processes

Apart from application services, most applications provide admin tasks as well. This principle advises to use the same release bundle as well as an identical environment for both application services and admin tasks. Admin code should also be packaged along with the application code.

Not only is this principle valid for microservices, but also it is applicable to any application development.

Microservice use cases

A microservice is not a silver bullet and will not solve all the architectural challenges of today's world. There is no hard-and-fast rule or rigid guideline on when to use microservices.

Microservices may not fit in each and every use case. The success of microservices largely depends on the selection of use cases. The first and the foremost activity is to do a litmus test of the use case against the microservices' benefits. The litmus test must cover all the microservices' benefits we discussed earlier in this chapter. For a given use case, if there are no quantifiable benefits or the cost outweighs the benefits, then the use case may not be the right choice for microservices.

Let's discuss some commonly used scenarios that are suitable candidates for a microservices architecture:

- Migrating a monolithic application due to improvements required in scalability, manageability, agility, or speed of delivery. Another similar scenario is rewriting an end-of-life heavily used legacy application. In both cases, microservices present an opportunity. Using a microservices architecture, it is possible to replatform a legacy application by slowly transforming functions to microservices. There are benefits in this approach. There is no humongous upfront investment required, no major disruption to business, and no severe business risks. As the service dependencies are known, the microservices dependencies can be well managed.
- Utility computing scenarios such as integrating an optimization service, forecasting service, price calculation service, prediction service, offer service, recommendation service, and so on are good candidates for microservices. These are independent stateless computing units that accept certain data, apply algorithms, and return the results. Independent technical services such as the communication service, the encryption service, authentication services, and so on are also good candidates for microservices.
- In many cases, we can build headless business applications or services that are autonomous in nature—for instance, the payment service, login service, flight search service, customer profile service, notification service, and so on. These are normally reused across multiple channels and, hence, are good candidates for building them as microservices.
- There could be micro or macro applications that serve a single purpose and performing a single responsibility. A simple time tracking application is an example of this category. All it does is capture the time, duration, and task performed. Common-use enterprise applications are also candidates for microservices.
- Backend services of a well-architected, responsive client-side MVC web application (the **Backend as a Service** (**BaaS**) scenario) load data on demand in response to the user navigation. In most of these scenarios, data could be coming from multiple logically different data sources as described in the *Fly By Points* example mentioned earlier.
- Highly agile applications, applications demanding speed of delivery or time to market, innovation pilots, applications selected for DevOps, applications of the System of Innovation type, and so on could also be considered as potential candidates for the microservices architecture.

- Applications that we could anticipate getting benefits from microservices such as polyglot requirements, applications that require **Command Query Responsibility segregations** (**CQRS**), and so on are also potential candidates of the microservices architecture.

If the use case falls into any of these categories, it is a potential candidate for the microservices architecture.

There are few scenarios in which we should consider avoiding microservices:

- If the organization's policies are forced to use centrally managed heavyweight components such as ESB to host a business logic or if the organization has any other policies that hinder the fundamental principles of microservices, then microservices are not the right solution unless the organizational process is relaxed.
- If the organization's culture, processes, and so on are based on the traditional waterfall delivery model, lengthy release cycles, matrix teams, manual deployments and cumbersome release processes, no infrastructure provisioning, and so on, then microservices may not be the right fit. This is underpinned by Conway's Law. This states that there is a strong link between the organizational structure and software it creates.

Tip

Read more about the Conway's Law at:

http://www.melconway.com/Home/Conways_Law.html

Microservices early adopters

Many organizations have already successfully embarked on their journey to the microservices world. In this section, we will examine some of the frontrunners on the microservices space to analyze why they did what they did and how they did it. We will conduct some analysis at the end to draw some conclusions:

- **Netflix** (www.netflix.com): Netflix, an international on-demand media streaming company, is a pioneer in the microservices space. Netflix transformed their large pool of developers developing traditional monolithic code to smaller development teams producing microservices. These microservices work together to stream digital media to millions of Netflix customers. At Netflix, engineers started with monolithic, went through the pain, and then broke the application into smaller units that are loosely coupled and aligned to the business capability.
- **Uber** (www.uber.com): Uber, an international transportation network company, began in 2008 with a monolithic architecture with a single code base. All services were embedded into the monolithic application. When Uber expanded their business from one city to multiple cities, the challenges started. Uber then moved to SOA-based architecture by breaking the system into smaller independent units. Each module was given to different teams and empowered them to choose their language, framework, and database. Uber has many microservices deployed in their ecosystem using RPC and REST.
- **Airbnb** (www.airbnb.com): Airbnb, a world leader providing a trusted marketplace for accommodation, started with a monolithic application that performed all the required functions of the business. Airbnb faced scalability issues with increased traffic. A single code base became too complicated to manage, resulted in a poor separation of concerns, and ran into performance issues. Airbnb broke their monolithic application into smaller pieces with separate code bases running on separate machines with separate deployment cycles. Airbnb developed their own microservices or SOA ecosystem around these services.
- **Orbitz** (www.orbitz.com): Orbitz, an online travel portal, started with a monolithic architecture in the 2000s with a web layer, a business layer, and a database layer. As Orbitz expanded their business, they faced manageability and scalability issues with monolithic-tiered architecture. Orbitz then went through continuous architecture changes. Later, Orbitz broke down their monolithic to many smaller applications.
- **eBay** (www.ebay.com): eBay, one of the largest online retailers, started in the late 1990s with a monolithic Perl application and FreeBSD as the database. eBay went through scaling issues as the business grew. It was consistently investing in improving its architecture. In the mid 2000s, eBay moved to smaller decomposed systems based on Java and web services. They employed database partitions and functional segregation to meet the required scalability.
- **Amazon** (www.amazon.com): Amazon, one of the largest online retailer websites, was run on a big monolithic application written on C++ in 2001. The well-architected monolithic application was based on a tiered architecture with many modular components. However, all these components were tightly coupled. As a result, Amazon was not able to speed up their development cycle by splitting teams into smaller groups. Amazon then separated out the code as independent functional services, wrapped with web services, and eventually advanced to

microservices.

- **Gilt** (www.gilt.com): Gilt, an online shopping website, began in 2007 with a tiered monolithic Rails application and a Postgres database at the back. Similarly to many other applications, as traffic volumes increased, the web application was not able to provide the required resiliency. Gilt went through an architecture overhaul by introducing Java and polyglot persistence. Later, Gilt moved to many smaller applications using the microservices concept.
- **Twitter** (www.twitter.com): Twitter, one of the largest social websites, began with a three-tiered monolithic rails application in the mid 2000s. Later, when Twitter experienced growth in its user base, they went through an architecture-refactoring cycle. With this refactoring, Twitter moved away from a typical web application to an API-based event driven core. Twitter uses Scala and Java to develop microservices with polyglot persistence.
- **Nike** (www.nike.com): Nike, the world leader in apparel and footwear, transformed their monolithic applications to microservices. Similarly to many other organizations, Nike too was run with age-old legacy applications that were hardly stable. In their journey, Nike moved to heavyweight commercial products with an objective to stabilize legacy applications but ended up in monolithic applications that were expensive to scale, had long release cycles, and needed too much manual work to deploy and manage applications. Later, Nike moved to a microservices-based architecture that brought down the development cycle considerably.

The common theme is monolithic migrations

When we analyze the preceding enterprises, there is one common theme. All these enterprises started with monolithic applications and transitioned to a microservices architecture by applying learning and pain points from their previous editions.

Even today, many start-ups begin with monolith as it is easy to start, conceptualize, and then slowly move to microservices when the demand arises. Monolithic to microservices migration scenarios have an added advantage: they have all the information upfront, readily available for refactoring.

Though, for all these enterprises, it is monolithic transformation, the catalysts were different for different organizations. Some of the common motivations are a lack of scalability, long development cycles, process automation, manageability, and changes in the business models.

While monolithic migrations are no-brainers, there are opportunities to build microservices from the ground up. More than building ground-up systems, look for opportunities to build smaller services that are quick wins for business—for example, adding a trucking service to an airline's end-to-end cargo management system or adding a customer scoring service to a retailer's loyalty system. These could be implemented as independent microservices exchanging messages with their respective monolithic applications.

Another point is that many organizations use microservices only for their business-critical customer engagement applications, leaving the rest of the legacy monolithic applications to take their own trajectory.

Another important observation is that most of the organizations examined previously are at different levels of maturity in their microservices journey. When eBay transitioned from a monolithic application in the early 2000s, they functionally split the application into smaller, independent, and deployable units. These logically divided units are wrapped with web services. While single responsibility and autonomy are their underpinning principles, the architectures are limited to the technologies and tools available at that point in time. Organizations such as Netflix and Airbnb built capabilities of their own to solve the specific challenges they faced. To summarize, all of these are not truly microservices, but are small, business-aligned services following the same characteristics.

There is no state called "definite or ultimate microservices". It is a journey and is evolving and maturing day by day. The mantra for architects and developers is the replaceability principle; build an architecture that maximizes the ability to replace its parts and minimizes the cost of replacing its parts. The bottom line is that enterprises shouldn't attempt to develop microservices by just following the hype.

Summary

In this chapter, you learned about the fundamentals of microservices with the help of a few examples.

We explored the evolution of microservices from traditional monolithic applications. We examined some of the principles and the mind shift required for modern application architectures. We also took a look at the characteristics and benefits of microservices and use cases. In this chapter, we established the microservices' relationship with service-oriented architecture and Twelve-Factor Apps. Lastly, we analyzed examples of a few enterprises from different industries.

We will develop a few sample microservices in the next chapter to bring more clarity to our learnings in this chapter.

Chapter 2. Building Microservices with Spring Boot

Developing microservices is not so tedious anymore thanks to the powerful Spring Boot framework. Spring Boot is a framework to develop production-ready microservices in Java.

This chapter will move from the microservices theory explained in the previous chapter to hands-on practice by reviewing code samples. This chapter will introduce the Spring Boot framework and explain how Spring Boot can help build RESTful microservices in line with the principles and characteristics discussed in the previous chapter. Finally, some of the features offered by Spring Boot to make microservices production-ready will be reviewed.

By the end of this chapter, you will have learned about:

- Setting up the latest Spring development environment
- Developing RESTful services using the Spring framework
- Using Spring Boot to build fully qualified microservices
- Useful Spring Boot features to build production-ready microservices

Setting up a development environment

To crystalize microservices concepts, a couple of microservices will be built. For this, it is assumed that the following components are installed:

- **JDK 1.8**: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- **Spring Tool Suite 3.7.2 (STS)**: https://spring.io/tools/sts/all
- **Maven 3.3.1**: https://maven.apache.org/download.cgi

Alternately, other IDEs such as IntelliJ IDEA, NetBeans, or Eclipse could be used. Similarly, alternate build tools such as Gradle can be used. It is assumed that the Maven repository, class path, and other path variables are set properly to run STS and Maven projects.

This chapter is based on the following versions of Spring libraries:

- Spring Framework `4.2.6.RELEASE`
- Spring Boot `1.3.5.RELEASE`

Tip

Detailed steps to download the code bundle are mentioned in the Preface of this book. Have a look.

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Spring-Microservices. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Developing a RESTful service – the legacy approach

This example will review the traditional RESTful service development before jumping deep into Spring Boot.

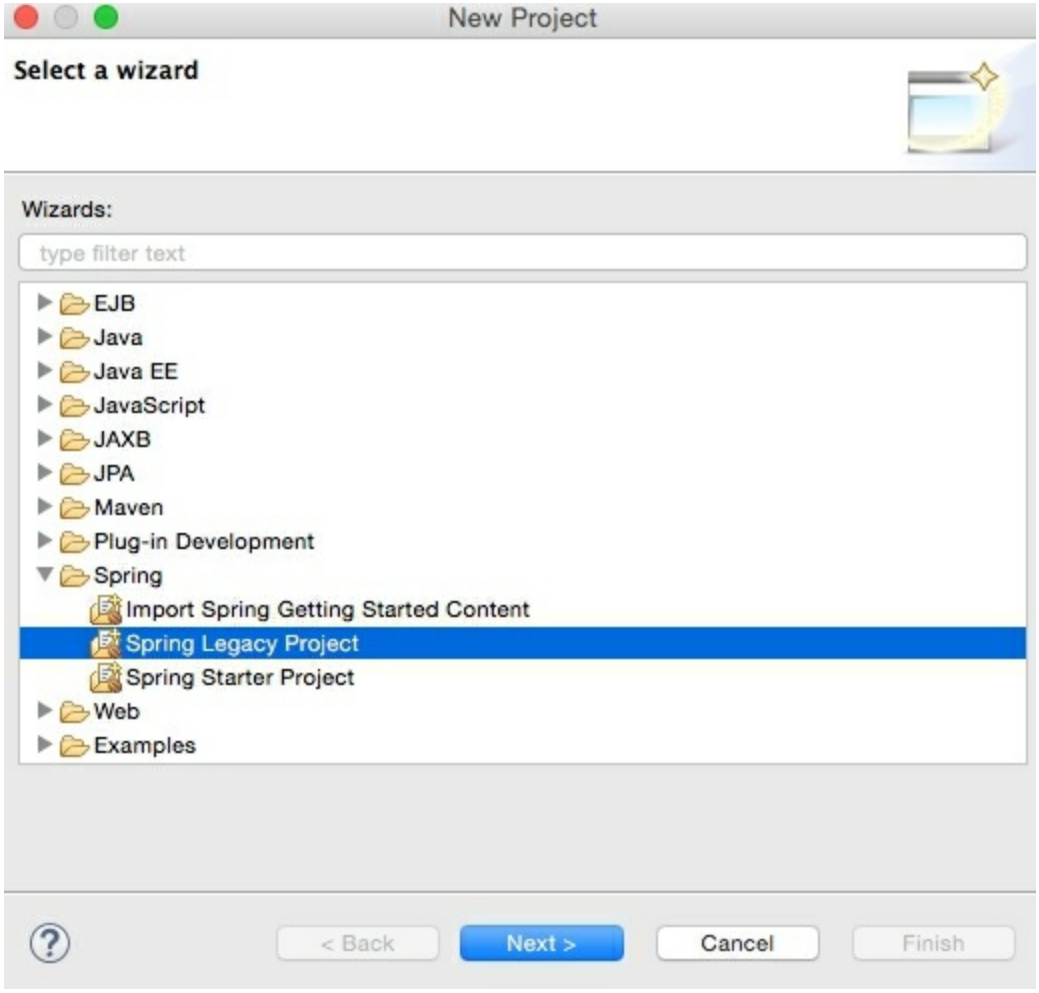
STS will be used to develop this REST/JSON service.

Note

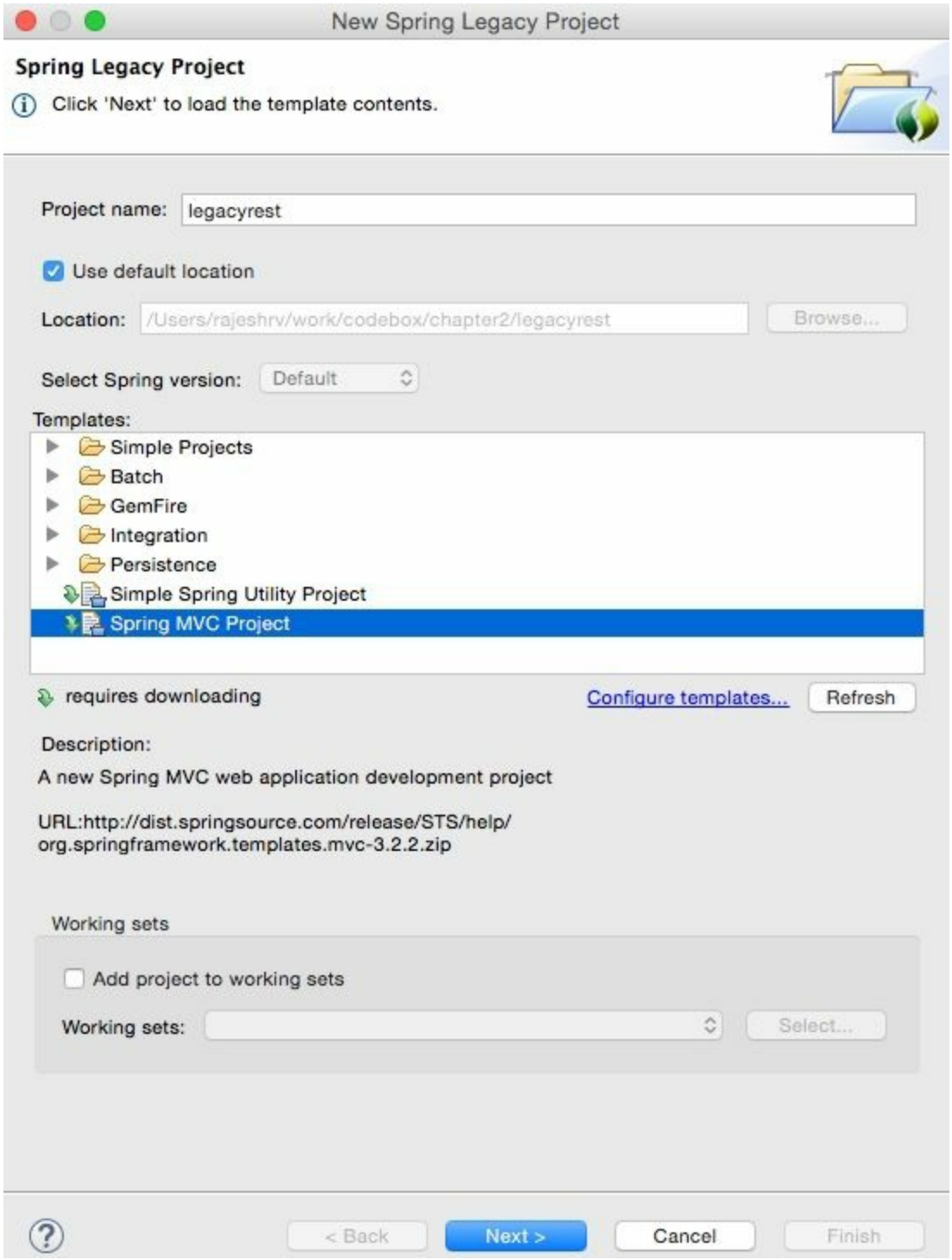
The full source code of this example is available as the `legacyrest` project in the code files of this book.

The following are the steps to develop the first RESTful service:

1. Start STS and set a workspace of choice for this project.
2. Navigate to **File** | **New** | **Project**.
3. Select **Spring Legacy Project** as shown in the following screenshot and click on **Next**:



4. Select **Spring MVC Project** as shown in the following diagram and click on **Next**:



5. Select a top-level package name of choice. This example uses `org.rvslab.chapter2.legacyrest` as the top-level package.
6. Then, click on **Finish**.
7. This will create a project in the STS workspace with the name `legacyrest`.

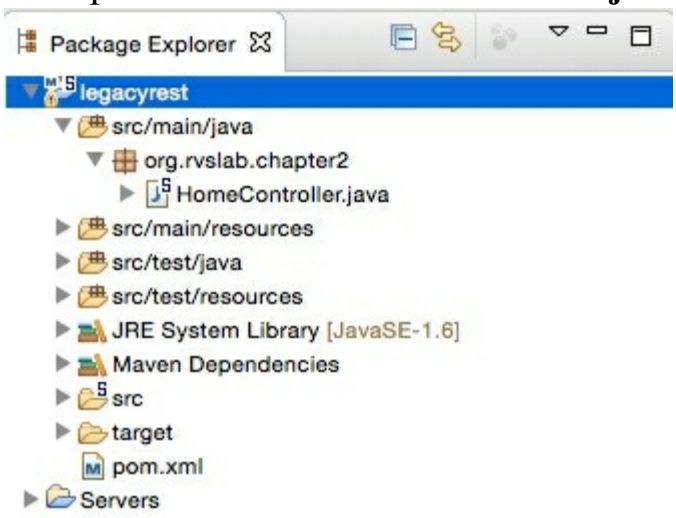
   Before proceeding further, `pom.xml` needs editing.
8. Change the Spring version to `4.2.6.RELEASE`, as follows:

```
<org.springframework-version>4.2.6.RELEASE</org.springframework-version>
```

9. Add **Jackson** dependencies in the `pom.xml` file for JSON-to-POJO and POJO-to-JSON conversions. Note that the `2.*.*` version is used to ensure compatibility with Spring 4.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.4</version>
</dependency>
```

10. Some Java code needs to be added. In **Java Resources**, under **legacyrest**, expand the package and open the default **HomeController.java** file:



11. The default implementation is targeted more towards the MVC project. Rewriting `HomeController.java` to return a JSON value in response to the REST call will do the trick. The resulting `HomeController.java` file will look similar to the following:

```
@RestController
public class HomeController {
  @RequestMapping("/")
  public Greet sayHello(){
    return new Greet("Hello World!");
  }
}
class Greet {
  private String message;
  public Greet(String message) {
    this.message = message;
  }
  //add getter and setter
}
```

Examining the code, there are now two classes:

- `Greet`: This is a simple Java class with getters and setters to represent a data object. There is only one attribute in the `Greet` class, which is `message`.
- `HomeController.java`: This is nothing but a Spring controller REST endpoint to handle HTTP requests.

Note that the annotation used in `HomeController` is `@RestController`, which automatically

injects `@Controller` and `@ResponseBody` and has the same effect as the following code:

```
@Controller
@ResponseBody
public class HomeController { }
```

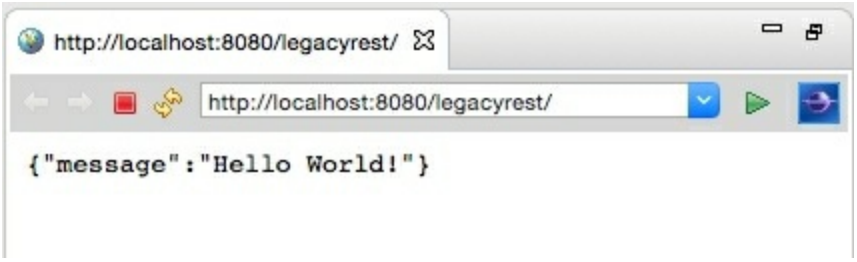
12. The project can now be run by right-clicking on **legacyrest**, navigating to **Run As** | **Run On Server**, and then selecting the default server (**Pivotal tc Server Developer Edition v3.1**) that comes along with STS.

    This should automatically start the server and deploy the web application on the TC server.

    If the server started properly, the following message will appear in the console:

```
INFO : org.springframework.web.servlet.DispatcherServlet - FrameworkServlet
'appServlet': initialization completed in 906 ms
May 08, 2016 8:22:48 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2289 ms
```

13. If everything is fine, STS will open a browser window to `http://localhost:8080/legacyrest/` and display the JSON object as shown in the browser. Right-click on and navigate to **legacyrest** | **Properties** | **Web Project Settings** and review **Context Root** to identify the context root of the web application:



The alternate build option is to use Maven. Right-click on the project and navigate to **Run As** | **Maven install**. This will generate `chapter2-1.0.0-BUILD-SNAPSHOT.war` under the target folder. This war is deployable in any servlet container such as Tomcat, JBoss, and so on.

Moving from traditional web applications to microservices

Carefully examining the preceding RESTful service will reveal whether this really constitutes a microservice. At first glance, the preceding RESTful service is a fully qualified interoperable REST/JSON service. However, it is not fully autonomous in nature. This is primarily because the service relies on an underlying application server or web container. In the preceding example, a war was explicitly created and deployed on a Tomcat server.

This is a traditional approach to developing RESTful services as a web application. However, from the microservices point of view, one needs a mechanism to develop services as executables, self-contained JAR files with an embedded HTTP listener.

Spring Boot is a tool that allows easy development of such kinds of services. Dropwizard and WildFly Swarm are alternate server-less RESTful stacks.

Using Spring Boot to build RESTful microservices

Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and microservices quickly and easily. The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations. Using the 80-20 principle, developers should be able to kickstart a variety of Spring applications with many default values. Spring Boot further presents opportunities for the developers to customize applications by overriding the autoconfigured values.

Spring Boot not only increases the speed of development but also provides a set of production-ready ops features such as health checks and metrics collection. As Spring Boot masks many configuration parameters and abstracts many lower-level implementations, it minimizes the chance of error to a certain extent. Spring Boot recognizes the nature of the application based on the libraries available in the class path and runs the autoconfiguration classes packaged in these libraries.

Often, many developers mistakenly see Spring Boot as a code generator, but in reality, it is not. Spring Boot only autoconfigures build files—for example, POM files in the case of Maven. It also sets properties, such as data source properties, based on certain opinionated defaults. Take a look at the following code:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

For instance, in the preceding case, Spring Boot understands that the project is set to use the Spring Data JPA and HSQL databases. It automatically configures the driver class and other connection parameters.

One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations. Spring Boot also enables microservices' development by packaging all the required runtime dependencies in a fat executable JAR file.

Getting started with Spring Boot

There are different ways that Spring Boot-based application development can be started:

- Using the Spring Boot CLI as a command-line tool
- Using IDEs such as STS to provide Spring Boot, which are supported out of the box
- Using the Spring Initializr project at http://start.spring.io

All these three options will be explored in this chapter, developing a variety of sample services.

Developing the Spring Boot microservice using the CLI

The easiest way to develop and demonstrate Spring Boot's capabilities is using the Spring Boot CLI, a command-line tool. Perform the following steps:

1. Install the Spring Boot command-line tool by downloading the `spring-boot-cli-1.3.5.RELEASE-bin.zip` file from http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.3.5.RELEASE/spring-boot-cli-1.3.5.RELEASE-bin.zip.
2. Unzip the file into a directory of your choice. Open a terminal window and change the terminal prompt to the `bin` folder.

   Ensure that the `bin` folder is added to the system path so that Spring Boot can be run from any location.
3. Verify the installation with the following command. If successful, the Spring CLI version will be printed in the console:

```
$spring --version
Spring CLI v1.3.5.RELEASE
```

4. As the next step, a quick REST service will be developed in Groovy, which is supported out of the box in Spring Boot. To do so, copy and paste the following code using any editor of choice and save it as `myfirstapp.groovy` in any folder:

```
@RestController
class HelloworldController {
    @RequestMapping("/")
    String sayHello() {
        "Hello World!"
    }
}
```

5. In order to run this Groovy application, go to the folder where `myfirstapp.groovy` is saved and execute the following command. The last few lines of the server start-up log will be similar to the following:

```
$spring run myfirstapp.groovy

2016-05-09 18:13:55.351  INFO 35861 --- [nio-8080-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization started
2016-05-09 18:13:55.375  INFO 35861 --- [nio-8080-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization completed in 24 ms
```

6. Open a browser window and go to `http://localhost:8080`; the browser will display the following message:

Hello World!

There is no war file created, and no Tomcat server was run. Spring Boot automatically picked up Tomcat as the webserver and embedded it into the application. This is a very basic, minimal microservice. The `@RestController` annotation, used in the previous code, will be examined in detail in the next example.

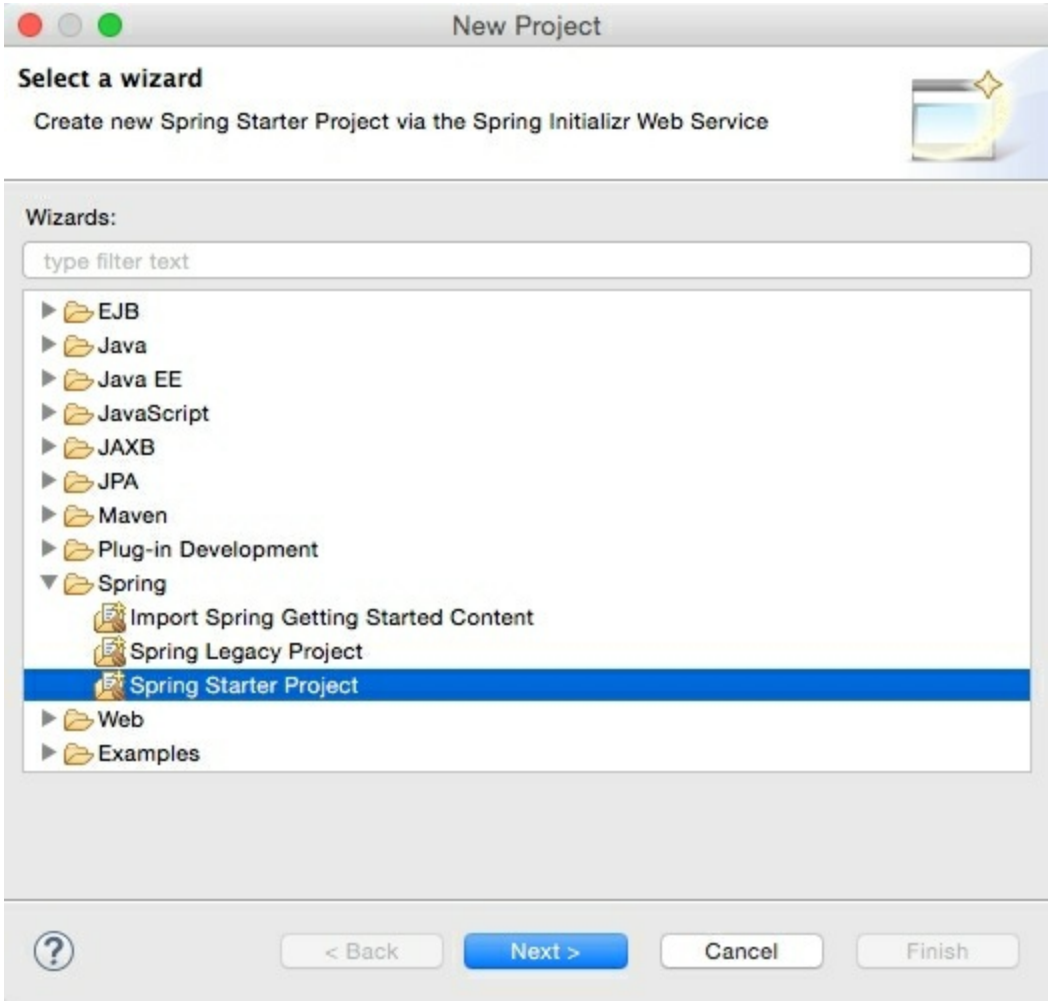
Developing the Spring Boot Java microservice using STS

In this section, developing another Java-based REST/JSON Spring Boot service using STS will be demonstrated.

Note

The full source code of this example is available as the `chapter2.bootrest` project in the code files of this book.

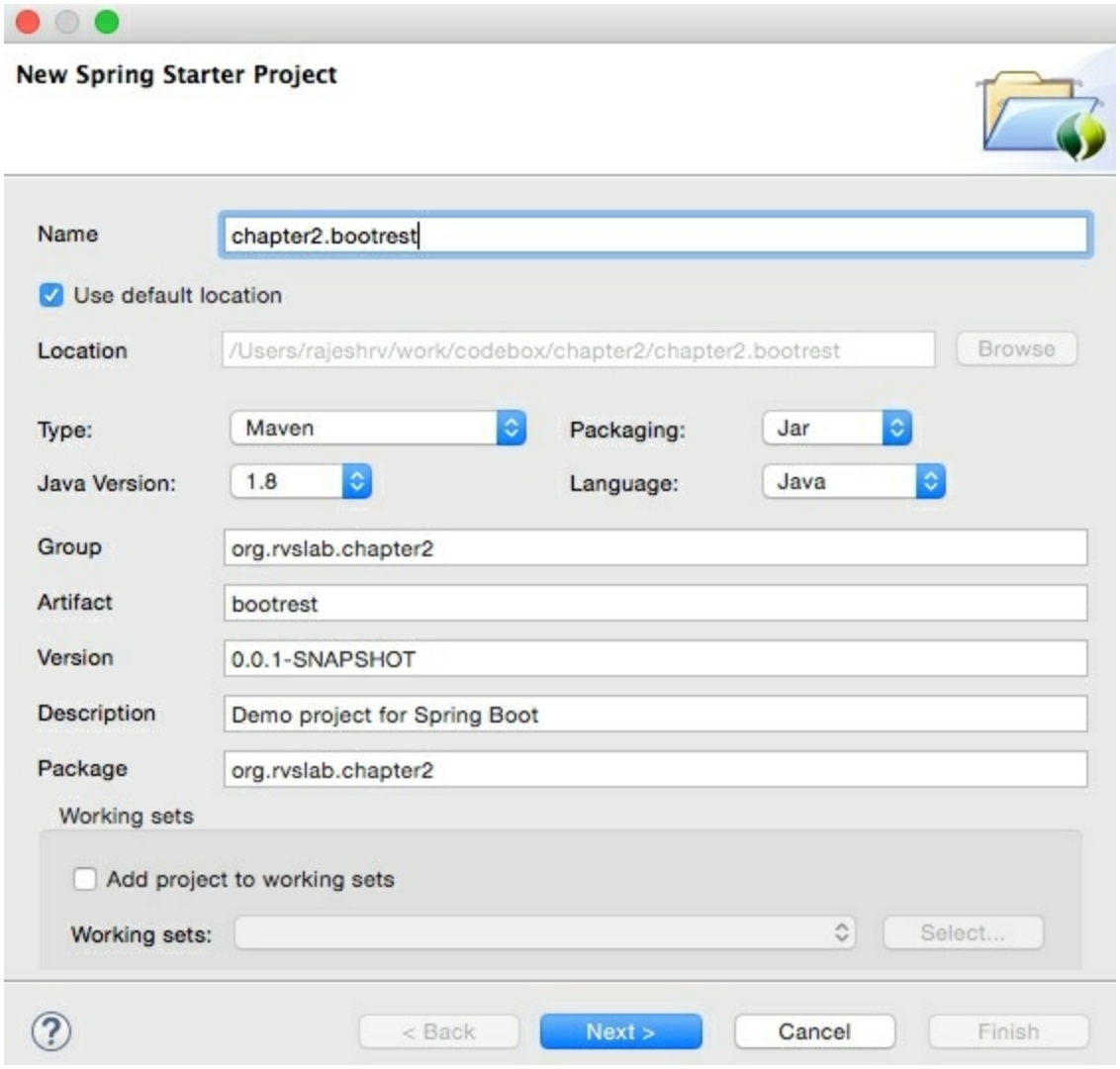
1. Open STS, right-click within the **Project Explorer** window, navigate to **New** | **Project**, and select **Spring Starter Project**, as shown in the following screenshot, and click on **Next**:



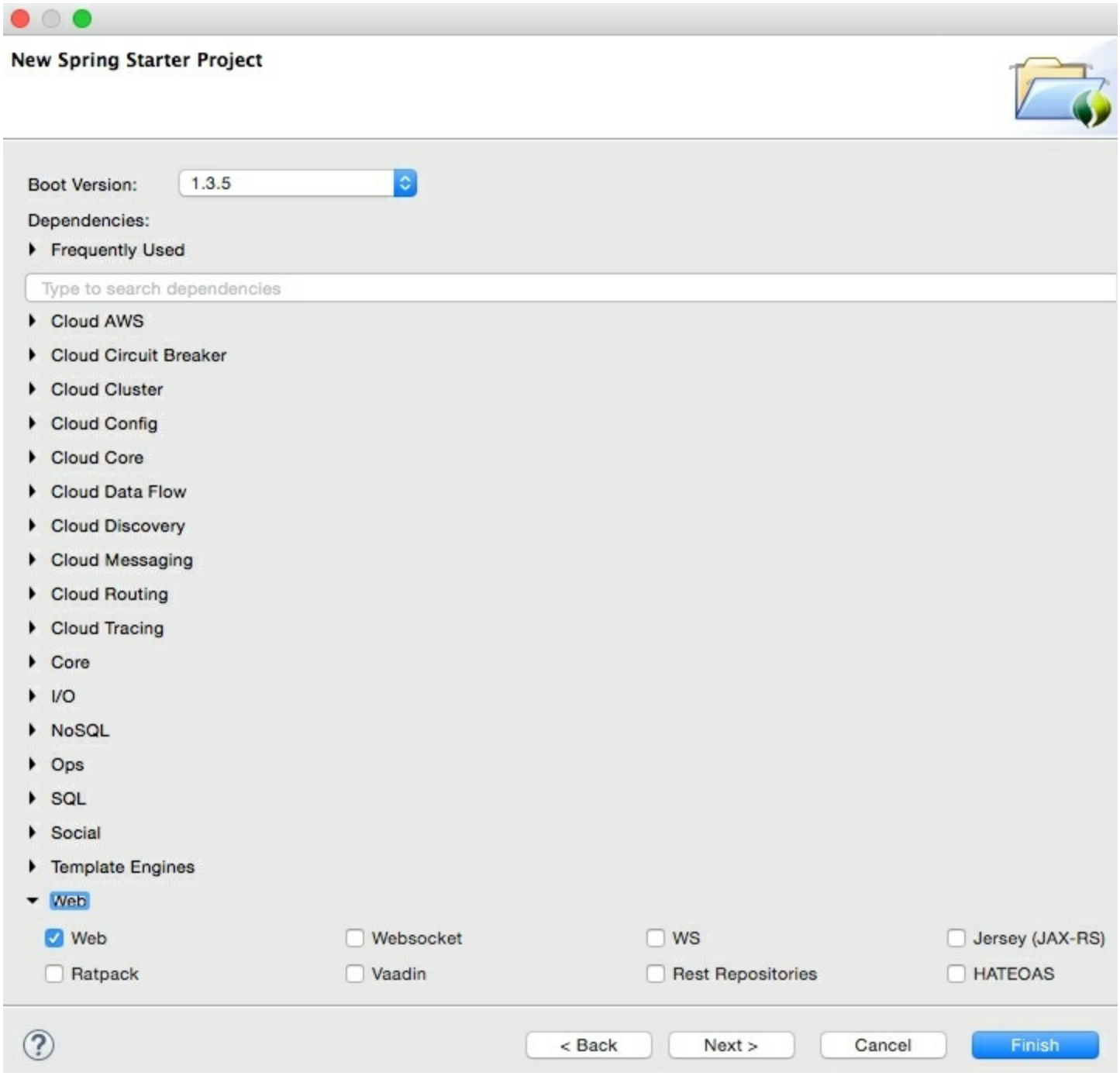
   Spring Starter Project is a basic template wizard that provides a number of other starter libraries to select from.
2. Type the project name as `chapter2.bootrest` or any other name of your choice. It is important to choose the packaging as JAR. In traditional web applications, a war file is created and then deployed to a servlet container, whereas Spring Boot packages all the dependencies to a self-

contained, autonomous JAR file with an embedded HTTP listener.

3. Select 1.8 under **Java Version**. Java 1.8 is recommended for Spring 4 applications. Change the other Maven properties such as **Group**, **Artifact**, and **Package**, as shown in the following screenshot:



4. Once completed, click on **Next**.
5. The wizard will show the library options. In this case, as the REST service is developed, select **Web** under **Web**. This is an interesting step that tells Spring Boot that a Spring MVC web application is being developed so that Spring Boot can include the necessary libraries, including Tomcat as the HTTP listener and other configurations, as required:



6. Click on **Finish**.

   This will generate a project named `chapter2.bootrest` in **Project Explorer** in STS:



7. Take a moment to examine the generated application. Files that are of interest are:
    - `pom.xml`
    - `Application.java`
    - `Application.properties`
    - `ApplicationTests.java`

Examining the POM file

The parent element is one of the interesting aspects in the `pom.xml` file. Take a look at the following:

```
<parent>
  <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.4.RELEASE</version>
</parent>
```

The `spring-boot-starter-parent` pattern is a **bill of materials** (**BOM**), a pattern used by Maven's dependency management. BOM is a special kind of POM file used to manage different library versions required for a project. The advantage of using the `spring-boot-starter-parent` POM file is that developers need not worry about finding the right compatible versions of different libraries such as Spring, Jersey, JUnit, Logback, Hibernate, Jackson, and so on. For instance, in our first legacy example, a specific version of the Jackson library was added to work with Spring 4. In this example, these are taken care of by the `spring-boot-starter-parent` pattern.

The starter POM file has a list of Boot dependencies, sensible resource filtering, and sensible plug-in configurations required for the Maven builds.

Tip

Refer to https://github.com/spring-projects/spring-boot/blob/1.3.x/spring-boot-dependencies/pom.xml to take a look at the different dependencies provided in the starter parent (version 1.3.x). All these dependencies can be overridden if required.

The starter POM file itself does not add JAR dependencies to the project. Instead, it will only add library versions. Subsequently, when dependencies are added to the POM file, they refer to the library versions from this POM file. A snapshot of some of the properties are as shown as follows:

```
<spring-boot.version>1.3.5.BUILD-SNAPSHOT</spring-boot.version>
<hibernate.version>4.3.11.Final</hibernate.version>
<jackson.version>2.6.6</jackson.version>
<jersey.version>2.22.2</jersey.version>
<logback.version>1.1.7</logback.version>
<spring.version>4.2.6.RELEASE</spring.version>
<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>
<tomcat.version>8.0.33</tomcat.version>
```

Reviewing the dependency section, one can see that this is a clean and neat POM file with only two dependencies, as follows:

```
<dependencies>
   <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
```

```
   <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
   </dependency>
</dependencies>
```

As web is selected, `spring-boot-starter-web` adds all dependencies required for a Spring MVC project. It also includes dependencies to Tomcat as an embedded HTTP listener. This provides an effective way to get all the dependencies required as a single bundle. Individual dependencies could be replaced with other libraries, for example replacing Tomcat with Jetty.

Similar to web, Spring Boot comes up with a number of `spring-boot-starter-*` libraries, such as `amqp`, `aop`, `batch`, `data-jpa`, `thymeleaf`, and so on.

The last thing to be reviewed in the `pom.xml` file is the Java 8 property. By default, the parent POM file adds Java 6. It is recommended to override the Java version to 8 for Spring:

```
<java.version>1.8</java.version>
```

Examining Application.java

Spring Boot, by default, generated a `org.rvslab.chapter2.Application.java` class under `src/main/java` to bootstrap, as follows:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

There is only a `main` method in `Application`, which will be invoked at startup as per the Java convention. The `main` method bootstraps the Spring Boot application by calling the `run` method on `SpringApplication`. `Application.class` is passed as a parameter to tell Spring Boot that this is the primary component.

More importantly, the magic is done by the `@SpringBootApplication` annotation. The `@SpringBootApplication` annotation is a top-level annotation that encapsulates three other annotations, as shown in the following code snippet:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
```

The `@Configuration` annotation hints that the contained class declares one or more `@Bean` definitions. The `@Configuration` annotation is meta-annotated with `@Component`; therefore, it is a candidate for component scanning.

The `@EnableAutoConfiguration` annotation tells Spring Boot to automatically configure the Spring application based on the dependencies available in the class path.

Examining application.properties

A default `application.properties` file is placed under `src/main/resources`. It is an important file to configure any required properties for the Spring Boot application. At the moment, this file is kept empty and will be revisited with some test cases later in this chapter.

Examining ApplicationTests.java

The last file to be examined is `ApplicationTests.java` under `src/test/java`. This is a placeholder to write test cases against the Spring Boot application.

To implement the first RESTful service, add a REST endpoint, as follows:

1. One can edit `Application.java` under `src/main/java` and add a RESTful service implementation. The RESTful service is exactly the same as what was done in the previous project. Append the following code at the end of the `Application.java` file:

```
@RestController
class GreetingController{
  @RequestMapping("/")
  Greet greet(){
    return new Greet("Hello World!");
  }
}
class Greet {
  private String message;
public Greet() {}

  public Greet(String message) {
    this.message = message;
  }
//add getter and setter
}
```

2. To run, navigate to **Run As** | **Spring Boot App**. Tomcat will be started on the `8080` port:



   We can notice from the log that:
   - Spring Boot get its own process ID (in this case, it is `41130`)
   - Spring Boot is automatically started with the Tomcat server at the localhost, port `8080`.
3. Next, open a browser and point to `http://localhost:8080`. This will show the JSON response as shown in the following screenshot:



A key difference between the legacy service and this one is that the Spring Boot service is self-contained. To make this clearer, run the Spring Boot application outside STS. Open a terminal window, go to the project folder, and run Maven, as follows:

```
$ maven install
```

This will generate a fat JAR file under the target folder of the project. Running the application from the command line shows:

```
$java -jar target/bootrest-0.0.1-SNAPSHOT.jar
```

As one can see, `bootrest-0.0.1-SNAPSHOT.jar` is self-contained and could be run as a standalone application. At this point, the JAR is as thin as 13 MB. Even though the application is no more than just "Hello World", the Spring Boot service just developed, practically follows the principles of microservices.

Testing the Spring Boot microservice

There are multiple ways to test REST/JSON Spring Boot microservices. The easiest way is to use a web browser or a curl command pointing to the URL, as follows:

```
curl http://localhost:8080
```

There are number of tools available to test RESTful services, such as Postman, Advanced REST client, SOAP UI, Paw, and so on.

In this example, to test the service, the default test class generated by Spring Boot will be used.

Adding a new test case to `ApplicatonTests.java` results in:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@WebIntegrationTest
public class ApplicationTests {
  @Test
  public void testVanillaService() {
    RestTemplate restTemplate = new RestTemplate();
    Greet greet = restTemplate.getForObject("http://localhost:8080",
Greet.class);
    Assert.assertEquals("Hello World!", greet.getMessage());
  }
}
```

Note that `@WebIntegrationTest` is added and `@WebAppConfiguration` removed at the class level. The `@WebIntegrationTest` annotation is a handy annotation that ensures that the tests are fired against a fully up-and-running server. Alternately, a combination of `@WebAppConfiguration` and `@IntegrationTest` will give the same result.

Also note that `RestTemplate` is used to call the RESTful service. `RestTemplate` is a utility class that abstracts the lower-level details of the HTTP client.

To test this, one can open a terminal window, go to the project folder, and run `mvn install`.

Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example

In the next example, Spring Initializr will be used to create a Spring Boot project. Spring Initializr is a drop-in replacement for the STS project wizard and provides a web UI to configure and generate a Spring Boot project. One of the advantages of Spring Initializr is that it can generate a project through the website that then can be imported into any IDE.

In this example, the concept of **HATEOAS** (short for **Hypermedia As The Engine Of Application State**) for REST-based services and the **HAL** (**Hypertext Application Language**) browser will be examined.

HATEOAS is a REST service pattern in which navigation links are provided as part of the payload metadata. The client application determines the state and follows the transition URLs provided as part of the state. This methodology is particularly useful in responsive mobile and web applications in which the client downloads additional data based on user navigation patterns.

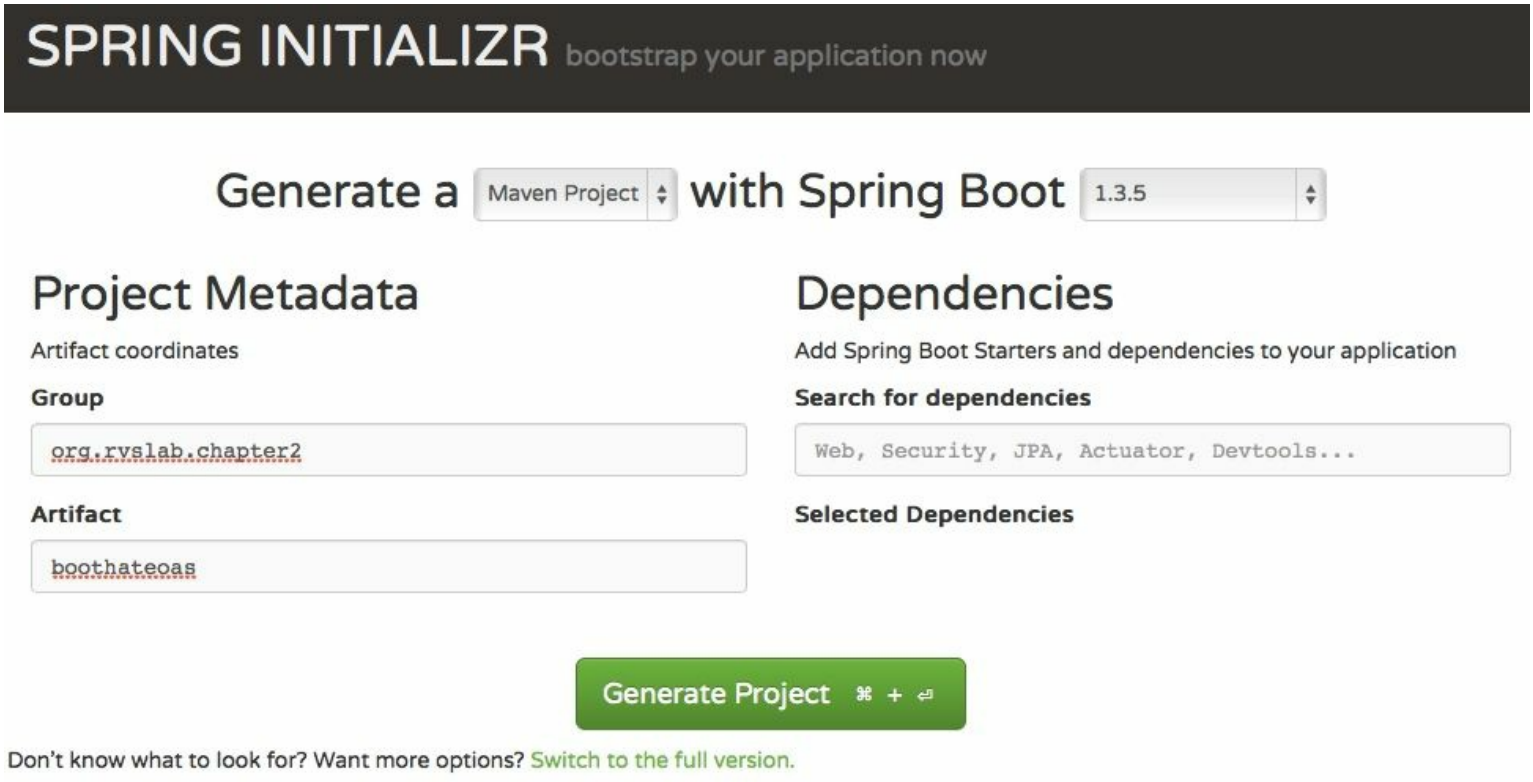
The HAL browser is a handy API browser for `hal+json` data. HAL is a format based on JSON that establishes conventions to represent hyperlinks between resources. HAL helps APIs be more explorable and discoverable.

Note

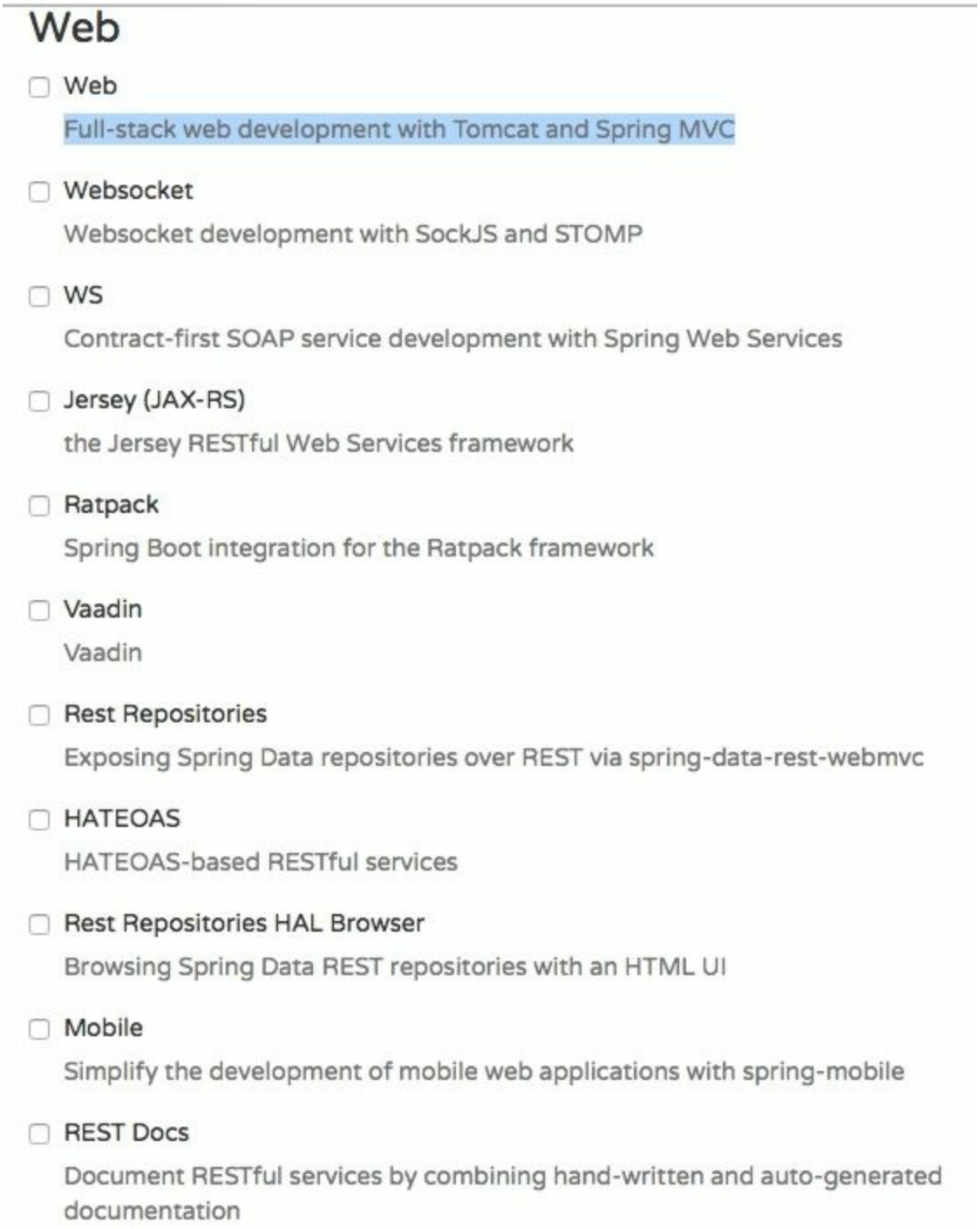
The full source code of this example is available as the `chapter2.boothateoas` project in the code files of this book.

Here are the concrete steps to develop a HATEOAS sample using Spring Initilizr:

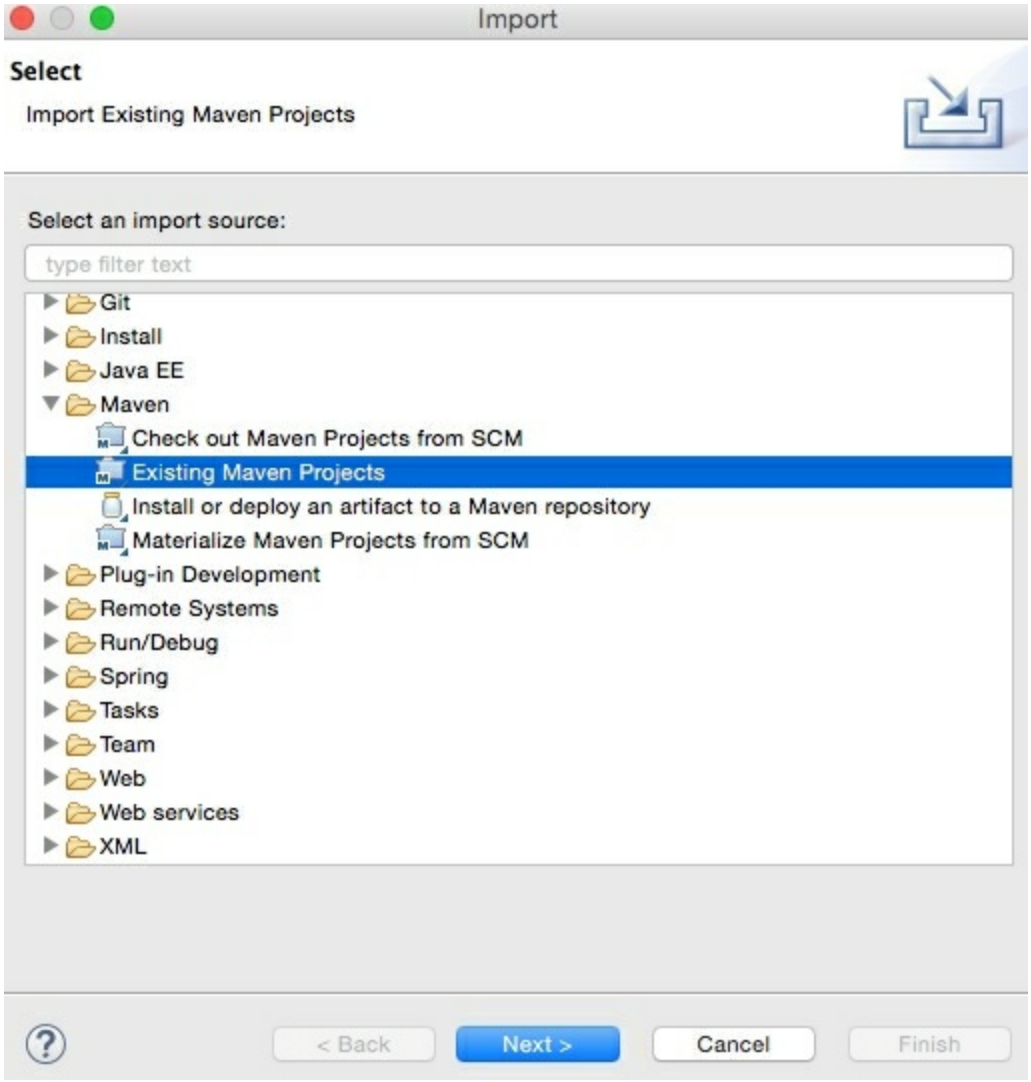
1. In order to use Spring Initilizr, go to https://start.spring.io:



2. Fill the details, such as whether it is a Maven project, Spring Boot version, group, and artifact ID, as shown earlier, and click on **Switch to the full version** link under the **Generate Project** button. Select **Web**, **HATEOAS**, and **Rest Repositories HAL Browser**. Make sure that the Java version is 8 and the package type is selected as **JAR**:



3. Once selected, hit the **Generate Project** button. This will generate a Maven project and download the project as a ZIP file into the download directory of the browser.
4. Unzip the file and save it to a directory of your choice.
5. Open STS, go to the **File** menu and click on **Import**:



6. Navigate to **Maven** | **Existing Maven Projects** and click on **Next**.
7. Click on **Browse** next to **Root Directory** and select the unzipped folder. Click on **Finish**. This will load the generated Maven project into STS' **Project Explorer**.
8. Edit the `Application.java` file to add a new REST endpoint, as follows:

```
@RequestMapping("/greeting")
@ResponseBody
public HttpEntity<Greet> greeting(@RequestParam(value = "name", required =
false, defaultValue = "HATEOAS") String name) {
        Greet greet = new Greet("Hello " + name);

greet.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelf
Rel());

        return new ResponseEntity<Greet>(greet, HttpStatus.OK);
}
```

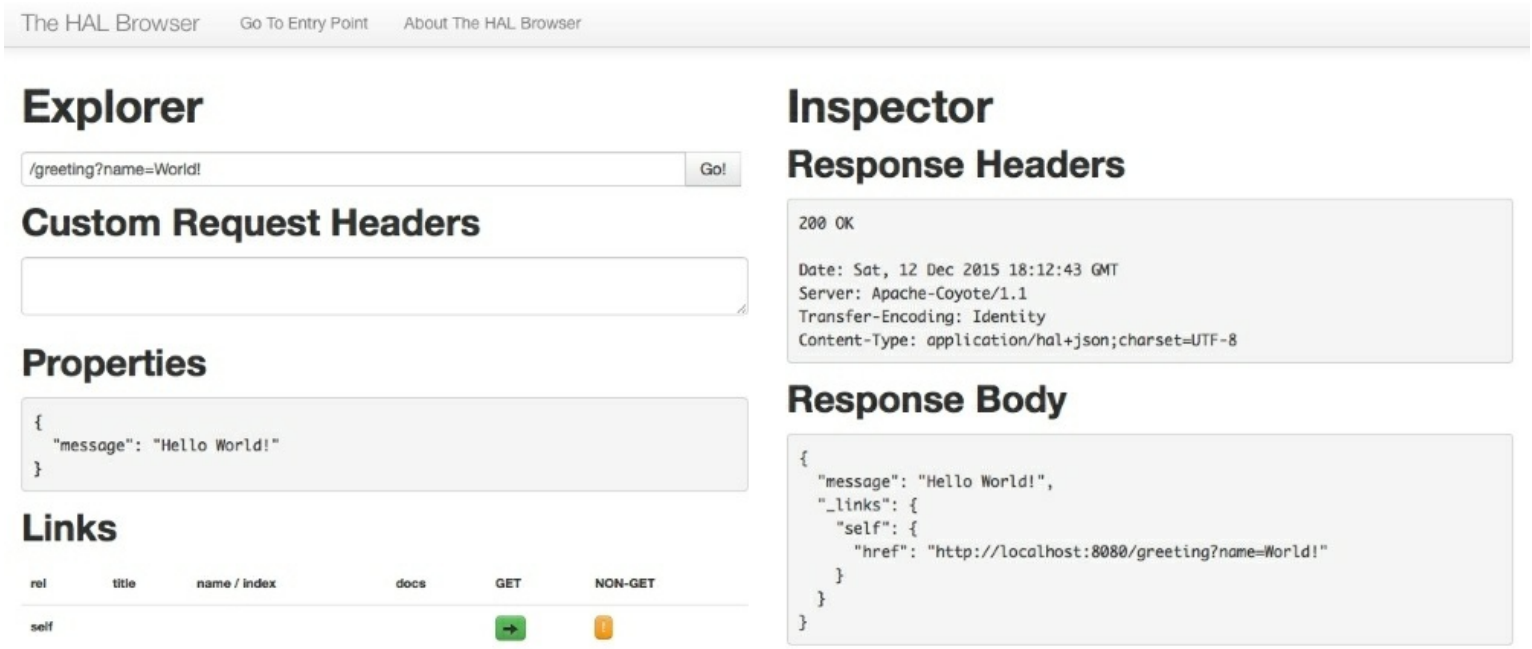
9. Note that this is the same `GreetingController` class as in the previous example. However, a method was added this time named `greeting`. In this new method, an additional optional request parameter is defined and defaulted to `HATEOAS`. The following code adds a link to the resulting JSON code. In this case, it adds the link to the same API:

```
greet.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelf
Rel());
```

In order to do this, we need to extend the `Greet` class from `ResourceSupport`, as shown here. The rest of the code remains the same:

```
class Greet extends ResourceSupport{
```

10. The `add` method is a method in `ResourceSupport`. The `linkTo` and `methodOn` methods are static methods of `ControllerLinkBuilder`, a utility class for creating links on controller classes. The `methodOn` method will do a dummy method invocation, and `linkTo` will create a link to the controller class. In this case, we will use `withSelfRel` to point it to itself.
11. This will essentially produce a link, `/greeting?name=HATEOAS`, by default. A client can read the link and initiate another call.
12. Run this as a Spring Boot app. Once the server startup is complete, point the browser to `http://localhost:8080`.
13. This will open the HAL browser window. In the **Explorer** field, type `/greeting?name=World!` and click on the **Go** button. If everything is fine, the HAL browser will show the response details as shown in the following screenshot:



As shown in the screenshot, the **Response Body** section has the result with a link with `href` pointing back to the same service. This is because we pointed the reference to itself. Also, review the **Links** section. The little green box against **self** is the navigable link.

It does not make much sense in this simple example, but this could be handy in larger applications with many related entities. Using the links provided, the client can easily navigate back and forth between these entities with ease.

What's next?

A number of basic Spring Boot examples have been reviewed so far. The rest of this chapter will examine some of the Spring Boot features that are important from a microservices development perspective. In the upcoming sections, we will take a look at how to work with dynamically configurable properties, change the default embedded web server, add security to the microservices, and implement cross-origin behavior when dealing with microservices.

Note

The full source code of this example is available as the `chapter2.boot-advanced` project in the code files of this book.

The Spring Boot configuration

In this section, the focus will be on the configuration aspects of Spring Boot. The `chapter2.bootrest` project, already developed, will be modified in this section to showcase configuration capabilities. Copy and paste `chapter2.bootrest` and rename the project as `chapter2.boot-advanced`.

Understanding the Spring Boot autoconfiguration

Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path. For each `spring-boot-starter-*` dependency in the POM file, Spring Boot executes a default `AutoConfiguration` class. `AutoConfiguration` classes use the `*AutoConfiguration` lexical pattern, where `*` represents the library. For example, the autoconfiguration of JPA repositories is done through `JpaRepositoriesAutoConfiguration`.

Run the application with `--debug` to see the autoconfiguration report. The following command shows the autoconfiguration report for the `chapter2.boot-advanced` project:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --debug
```

Here are some examples of the autoconfiguration classes:

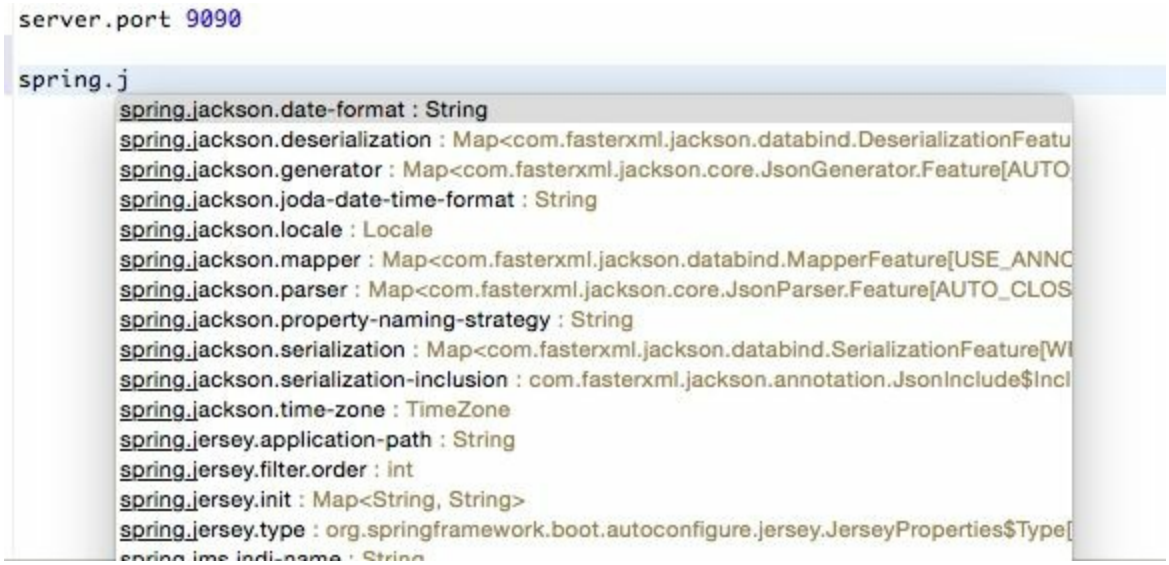
- `ServerPropertiesAutoConfiguration`
- `RepositoryRestMvcAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `JmsAutoConfiguration`

It is possible to exclude the autoconfiguration of certain libraries if the application has special requirements and you want to get full control of the configurations. The following is an example of excluding `DataSourceAutoConfiguration`:

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

Overriding default configuration values

It is also possible to override default configuration values using the `application.properties` file. STS provides an easy-to-autocomplete, contextual help on `application.properties`, as shown in the following screenshot:



In the preceding screenshot, `server.port` is edited to be set as `9090`. Running this application again will start the server on port `9090`.

Changing the location of the configuration file

In order to align with the Twelve-Factor app, configuration parameters need to be externalized from the code. Spring Boot externalizes all configurations into `application.properties`. However, it is still part of the application's build. Furthermore, properties can be read from outside the package by setting the following properties:

```
spring.config.name= # config file name
spring.config.location= # location of config file
```

Here, `spring.config.location` could be a local file location.

The following command starts the Spring Boot application with an externally provided configuration file:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --
spring.config.name=bootrest.properties
```

Reading custom properties

At startup, `SpringApplication` loads all the properties and adds them to the Spring `Environment` class. Add a custom property to the `application.properties` file. In this case, the custom property is named `bootrest.customproperty`. Autowire the Spring `Environment` class into the `GreetingController` class. Edit the `GreetingController` class to read the custom property from `Environment` and add a log statement to print the custom property to the console.

Perform the following steps to do this:

1. Add the following property to the `application.properties` file:

```
bootrest.customproperty=hello
```

2. Then, edit the `GreetingController` class as follows:

```
@Autowired
Environment env;

Greet greet(){
    logger.info("bootrest.customproperty "+
env.getProperty("bootrest.customproperty"));
    return new Greet("Hello World!");
}
```

3. Rerun the application. The log statement prints the custom variable in the console, as follows:

```
org.rvslab.chapter2.GreetingController   : bootrest.customproperty hello
```

Using a .yaml file for configuration

As an alternate to `application.properties`, one may use a `.yaml` file. YAML provides a JSON-like structured configuration compared to the flat properties file.

To see this in action, simply replace `application.properties` with `application.yaml` and add the following property:

```
server
port: 9080
```

Rerun the application to see the port printed in the console.

Using multiple configuration profiles

Furthermore, it is possible to have different profiles such as development, testing, staging, production, and so on. These are logical names. Using these, one can configure different values for the same properties for different environments. This is quite handy when running the Spring Boot application against different environments. In such cases, there is no rebuild required when moving from one environment to another.

Update the `.yaml` file as follows. The Spring Boot group profiles properties based on the dotted separator:

```
spring:
    profiles: development
server:
      port: 9090
---

spring:
    profiles: production
server:
      port: 8080
```

Run the Spring Boot application as follows to see the use of profiles:

```
mvn -Dspring.profiles.active=production install
mvn -Dspring.profiles.active=development install
```

Active profiles can be specified programmatically using the `@ActiveProfiles` annotation, which is especially useful when running test cases, as follows:

```
@ActiveProfiles("test")
```

Other options to read properties

The properties can be loaded in a number of ways, such as the following:

- Command-line parameters (`-Dhost.port =9090`)
- Operating system environment variables
- JNDI (`java:comp/env`)

Changing the default embedded web server

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. In the following example, Tomcat is replaced with Undertow:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Implementing Spring Boot security

It is important to secure microservices. In this section, some basic measures to secure Spring Boot microservices will be reviewed using `chapter2.bootrest` to demonstrate the security features.

Securing microservices with basic security

Adding basic authentication to Spring Boot is pretty simple. Add the following dependency to `pom.xml`. This will include the necessary Spring security library files:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Open `Application.java` and add `@EnableGlobalMethodSecurity` to the `Application` class. This annotation will enable method-level security:

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

The default basic authentication assumes the user as being `user`. The default password will be printed in the console at startup. Alternately, the username and password can be added in `application.properties`, as shown here:

```
security.user.name=guest
security.user.password=guest123
```

Add a new test case in `ApplicationTests` to test the secure service results, as in the following:

```
  @Test
  public void testSecureService() {
    String plainCreds = "guest:guest123";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + new
String(Base64.encode(plainCreds.getBytes())));
    HttpEntity<String> request = new HttpEntity<String>(headers);
    RestTemplate restTemplate = new RestTemplate();

    ResponseEntity<Greet> response =
restTemplate.exchange("http://localhost:8080", HttpMethod.GET, request,
Greet.class);
    Assert.assertEquals("Hello World!", response.getBody().getMessage());
  }
```

As shown in the code, a new `Authorization` request header with Base64 encoding the username-password string is created.

Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server

rejected the request with the following message:

```
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized
```

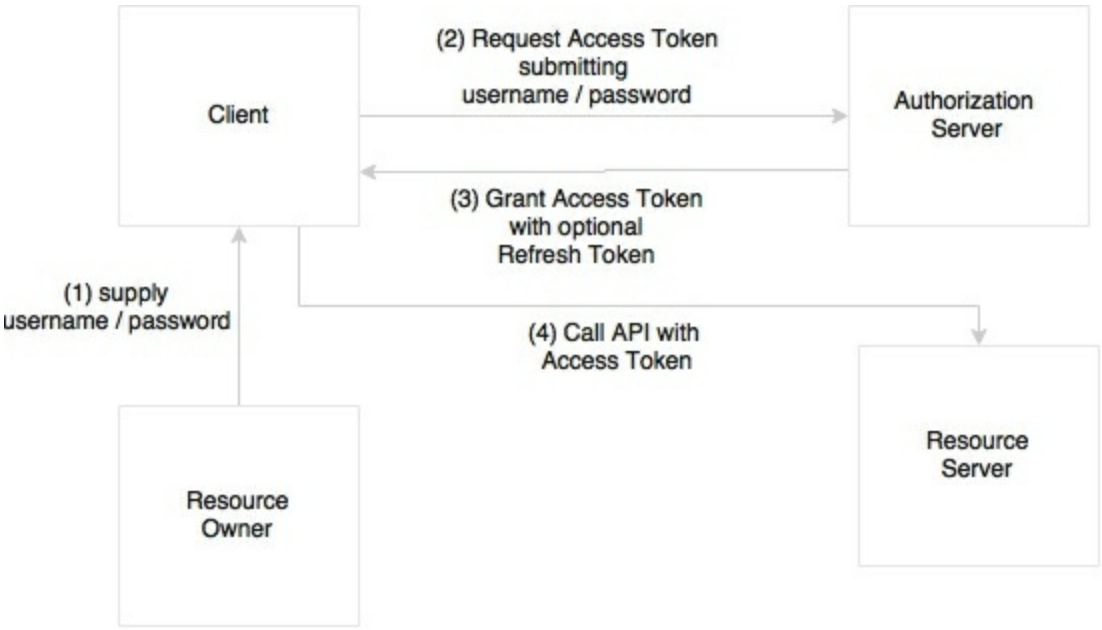
Securing a microservice with OAuth2

In this section, we will take a look at the basic Spring Boot configuration for OAuth2. When a client application requires access to a protected resource, the client sends a request to an authorization server. The authorization server validates the request and provides an access token. This access token is validated for every client-to-server request. The request and response sent back and forth depends on the grant type.

Tip

Read more about OAuth and grant types at http://oauth.net.

The resource owner password credentials grant approach will be used in this example:



In this case, as shown in the preceding diagram, the resource owner provides the client with a username and password. The client then sends a token request to the authorization server by providing the credential information. The authorization server authorizes the client and returns with an access token. On every subsequent request, the server validates the client token.

To implement OAuth2 in our example, perform the following steps:

1. As a first step, update `pom.xml` with the OAuth2 dependency, as follows:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```

2. Next, add two new annotations, `@EnableAuthorizationServer` and

`@EnableResourceServer`, to the `Application.java` file. The `@EnableAuthorizationServer` annotation creates an authorization server with an in-memory repository to store client tokens and provide clients with a username, password, client ID, and secret. The `@EnableResourceServer` annotation is used to access the tokens. This enables a spring security filter that is authenticated via an incoming OAuth2 token.

In our example, both the authorization server and resource server are the same. However, in practice, these two will run separately. Take a look at the following code:

```
@EnableResourceServer
@EnableAuthorizationServer
@SpringBootApplication
public class Application {
```

3. Add the following properties to the `application.properties` file:

```
security.user.name=guest
security.user.password=guest123
security.oauth2.client.clientId: trustedclient
security.oauth2.client.clientSecret: trustedclient123
security.oauth2.client.authorized-grant-types:
authorization_code,refresh_token,password
security.oauth2.client.scope: openid
```

4. Then, add another test case to test OAuth2, as follows:

```
  @Test
  public void testOAuthService() {
        ResourceOwnerPasswordResourceDetails resource = new
ResourceOwnerPasswordResourceDetails();
        resource.setUsername("guest");
        resource.setPassword("guest123");
          resource.setAccessTokenUri("http://localhost:8080/oauth/token");
        resource.setClientId("trustedclient");
        resource.setClientSecret("trustedclient123");
        resource.setGrantType("password");

        DefaultOAuth2ClientContext clientContext = new
DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource,
clientContext);

        Greet greet = restTemplate.getForObject("http://localhost:8080",
Greet.class);

        Assert.assertEquals("Hello World!", greet.getMessage());
  }
```

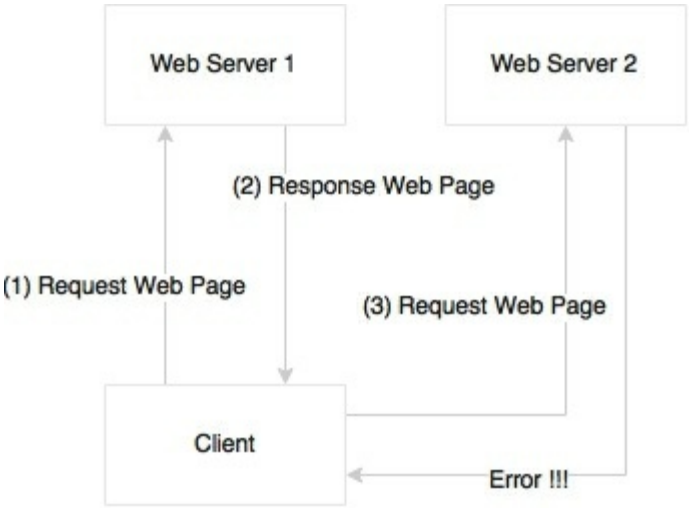
As shown in the preceding code, a special REST template, `OAuth2RestTemplate`, is created by passing the resource details encapsulated in a resource details object. This REST template handles the OAuth2 processes underneath. The access token URI is the endpoint for the token access.

5. Rerun the application using `mvn install`. The first two test cases will fail, and the new one will succeed. This is because the server only accepts OAuth2-enabled requests.

These are quick configurations provided by Spring Boot out of the box but are not good enough to be production grade. We may need to customize `ResourceServerConfigurer` and `AuthorizationServerConfigurer` to make them production-ready. This notwithstanding, the approach remains the same.

Enabling cross-origin access for microservices

Browsers are generally restricted when client-side web applications running from one origin request data from another origin. Enabling cross-origin access is generally termed as **CORS** (**Cross-Origin Resource Sharing**).



This example shows how to enable cross-origin requests. With microservices, as each service runs with its own origin, it will easily get into the issue of a client-side web application consuming data from multiple origins. For instance, a scenario where a browser client accessing Customer from the Customer microservice and Order History from the Order microservices is very common in the microservices world.

Spring Boot provides a simple declarative approach to enabling cross-origin requests. The following example shows how to enable a microservice to enable cross-origin requests:

```
@RestController
class GreetingController{
  @CrossOrigin
  @RequestMapping("/")
  Greet greet(){
    return new Greet("Hello World!");
  }
}
```

By default, all the origins and headers are accepted. We can further customize the cross-origin annotations by giving access to specific origins, as follows. The `@CrossOrigin` annotation enables a method or class to accept cross-origin requests:

```
@CrossOrigin("http://mytrustedorigin.com")
```

Global CORS can be enabled using the `WebMvcConfigurer` bean and customizing the `addCorsMappings(CorsRegistry registry)` method.

Implementing Spring Boot messaging

In an ideal case, all microservice interactions are expected to happen asynchronously using publish-subscribe semantics. Spring Boot provides a hassle-free mechanism to configure messaging solutions:

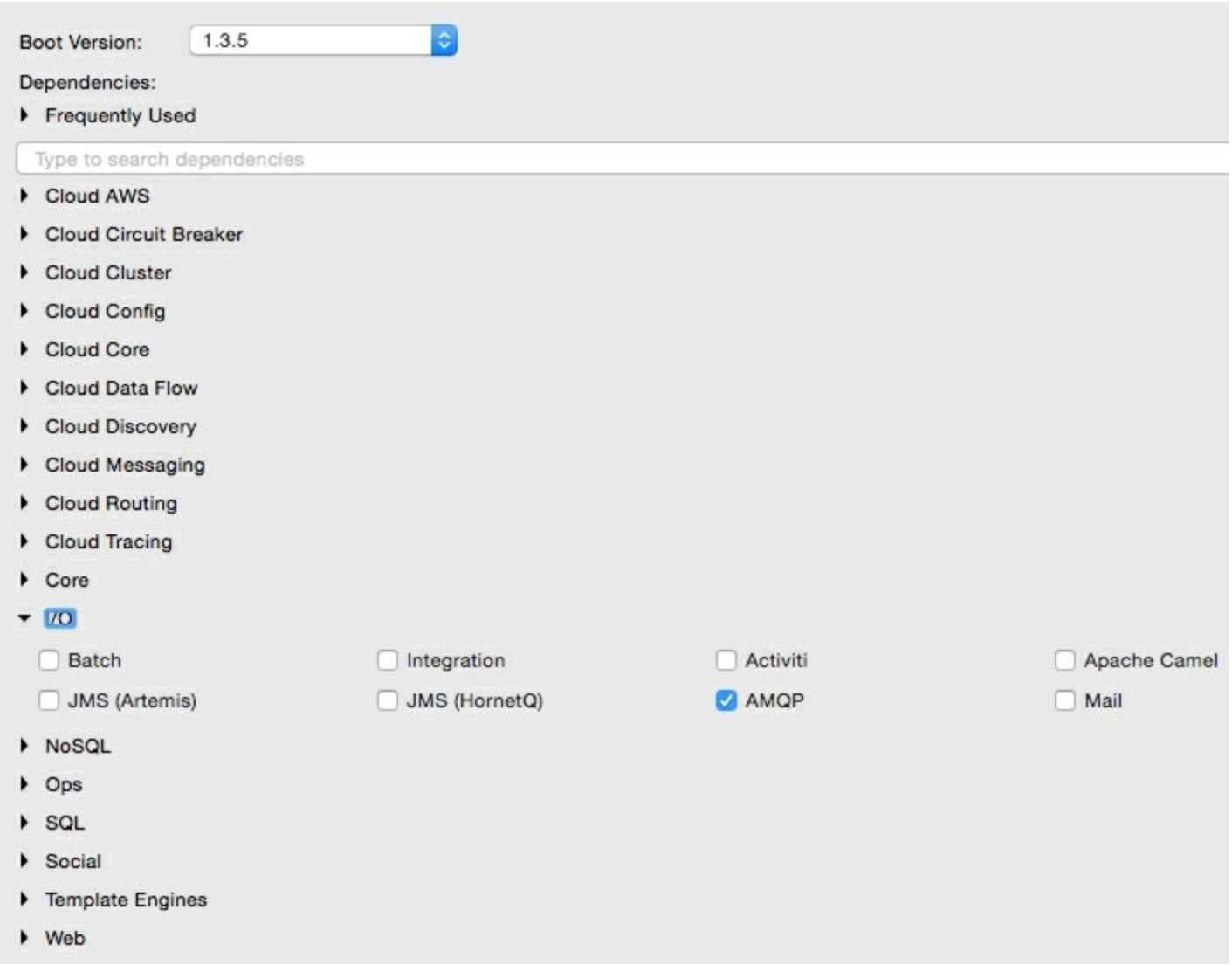


In this example, we will create a Spring Boot application with a sender and receiver, both connected though an external queue. Perform the following steps:

Note

The full source code of this example is available as the `chapter2.bootmessaging` project in the code files of this book.

1. Create a new project using STS to demonstrate this capability. In this example, instead of selecting **Web**, select **AMQP** under **I/O**:



2. Rabbit MQ will also be needed for this example. Download and install the latest version of Rabbit MQ from https://www.rabbitmq.com/download.html.

   Rabbit MQ 3.5.6 is used in this book.
3. Follow the installation steps documented on the site. Once ready, start the RabbitMQ server via the following command:

   ```
   $./rabbitmq-server
   ```

4. Make the configuration changes to the `application.properties` file to reflect the RabbitMQ configuration. The following configuration uses the default port, username, and password of RabbitMQ:

   ```
   spring.rabbitmq.host=localhost
   spring.rabbitmq.port=5672
   spring.rabbitmq.username=guest
   spring.rabbitmq.password=guest
   ```

5. Add a message sender component and a queue named `TestQ` of the `org.springframework.amqp.core.Queue` type to the `Application.java` file under `src/main/java`. `RabbitMessagingTemplate` is a convenient way to send messages, which

will abstract all the messaging semantics. Spring Boot provides all boilerplate configurations to send messages:

```
@Component
class Sender {
  @Autowired
  RabbitMessagingTemplate template;
  @Bean
  Queue queue() {
    return new Queue("TestQ", false);
  }
  public void send(String message){
    template.convertAndSend("TestQ", message);
  }
}
```

6. To receive the message, all that needs to be used is a `@RabbitListener` annotation. Spring Boot autoconfigures all the required boilerplate configurations:

```
@Component
class Receiver {
    @RabbitListener(queues = "TestQ")
    public void processMessage(String content) {
       System.out.println(content);
    }
}
```

7. The last piece of this exercise is to wire the sender to our main application and implement the `run` method of `CommandLineRunner` to initiate the message sending. When the application is initialized, it invokes the `run` method of `CommandLineRunner`, as follows:

```
@SpringBootApplication
public class Application implements CommandLineRunner{

  @Autowired
  Sender sender;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
      sender.send("Hello Messaging..!!!");
    }
}
```

8. Run the application as a Spring Boot application and verify the output. The following message will be printed in the console:

```
Hello Messaging..!!!
```

Developing a comprehensive microservice example

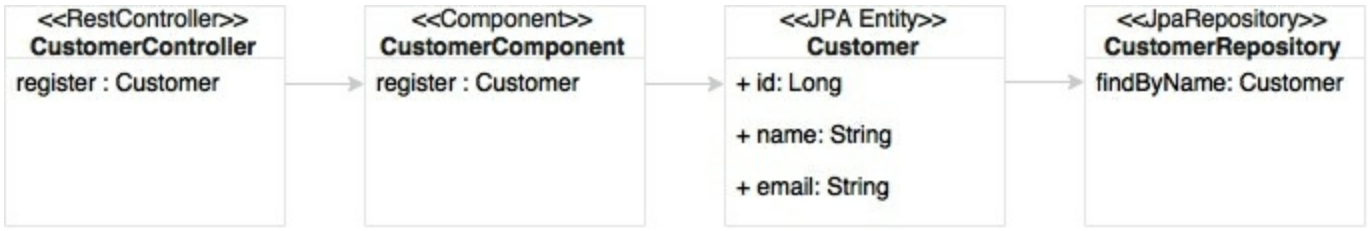
So far, the examples we have considered are no more than just a simple "Hello world." Putting together what we have learned, this section demonstrates an end-to-end Customer Profile microservice implementation. The Customer Profile microservices will demonstrate interaction between different microservices. It also demonstrates microservices with business logic and primitive data stores.

In this example, two microservices, the Customer Profile and Customer Notification services, will be developed:



As shown in the diagram, the Customer Profile microservice exposes methods to **create, read, update, and delete** (**CRUD**) a customer and a registration service to register a customer. The registration process applies certain business logic, saves the customer profile, and sends a message to the Customer Notification microservice. The Customer Notification microservice accepts the message sent by the registration service and sends an e-mail message to the customer using an SMTP server. Asynchronous messaging is used to integrate Customer Profile with the Customer Notification service.

The Customer microservices class domain model diagram is as shown here:



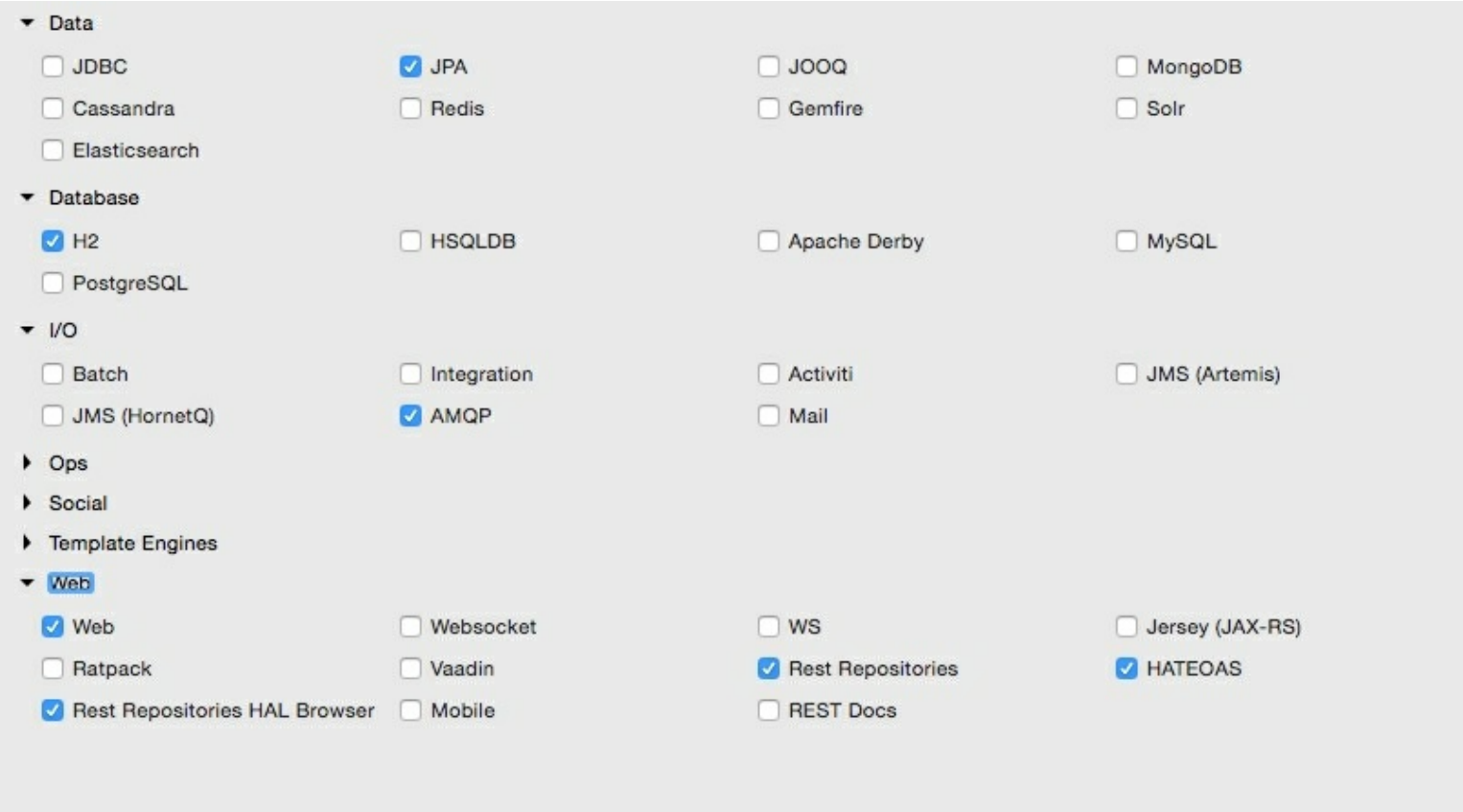
`CustomerController` in the diagram is the REST endpoint, which invokes a component class,

`CustomerComponent`. The component class/bean handles all the business logic. `CustomerRepository` is a Spring data JPA repository defined to handle the persistence of the `Customer` entity.

Note

The full source code of this example is available as the `chapter2.bootcustomer` and `chapter2.bootcustomernotification` projects in the code files of this book.

1. Create a new Spring Boot project and call it `chapter2.bootcustomer`, the same way as earlier. Select the options as in the following screenshot in the starter module selection screen:



This will create a web project with JPA, the REST repository, and H2 as a database. H2 is a tiny in-memory embedded database with which it is easy to demonstrate database features. In the real world, it is recommended to use an appropriate enterprise-grade database. This example uses JPA to define persistence entities and the REST repository to expose REST-based repository services.

The project structure will be similar to the following screenshot:



2. Start building the application by adding an Entity class named `Customer`. For simplicity, there are only three fields added to the `Customer` Entity class: the autogenerated `id` field, `name`, and `email`. Take a look at the following code:

```
@Entity
class Customer {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String name;
  private String email;
```

3. Add a repository class to handle the persistence handling of Customer. `CustomerRepository` extends the standard JPA repository. This means that all CRUD methods and default finder methods are automatically implemented by the Spring Data JPA repository, as follows:

```
@RepositoryRestResource
interface CustomerRespository extends JpaRepository <Customer,Long>{
  Optional<Customer> findByName(@Param("name") String name);
}
```

   In this example, we added a new method to the repository class, `findByName`, which essentially searches the customer based on the customer name and returns a `Customer` object if there is a matching name.
4. The `@RepositoryRestResource` annotation enables the repository access through RESTful services. This will also enable HATEOAS and HAL by default. As for CRUD methods there is no additional business logic required, we will leave it as it is without controller or component classes. Using HATEOAS will help us navigate through Customer Repository methods effortlessly.

   Note that there is no configuration added anywhere to point to any database. As H2 libraries are in the class path, all the configuration is done by default by Spring Boot based on the H2 autoconfiguration.
5. Update the `Application.java` file by adding `CommandLineRunner` to initialize the repository

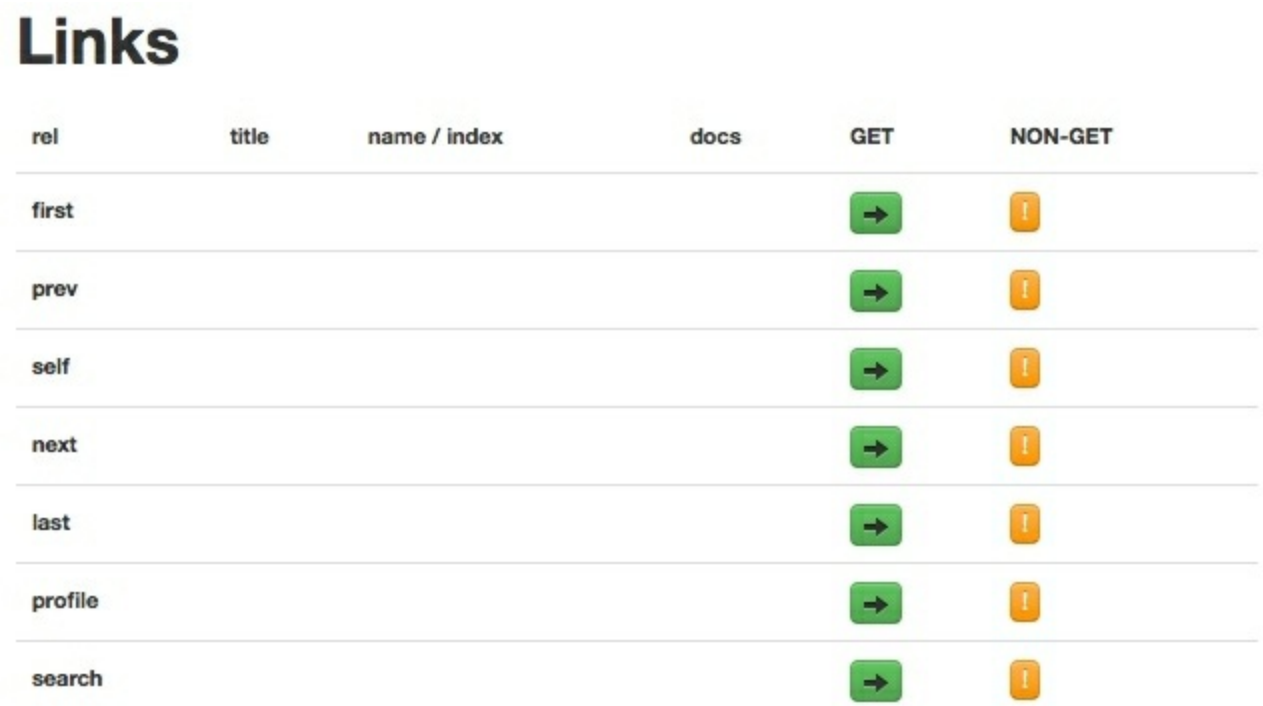
with some customer records, as follows:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
  CommandLineRunner init(CustomerRespository repo) {
  return (evt) ->  {
    repo.save(new Customer("Adam","adam@boot.com"));
    repo.save(new Customer("John","john@boot.com"));
  repo.save(new Customer("Smith","smith@boot.com"));
    repo.save(new Customer("Edgar","edgar@boot.com"));
    repo.save(new Customer("Martin","martin@boot.com"));
    repo.save(new Customer("Tom","tom@boot.com"));
    repo.save(new Customer("Sean","sean@boot.com"));
  };
  }
}
```

6. `CommandLineRunner`, defined as a bean, indicates that it should run when it is contained in `SpringApplication`. This will insert six sample customer records into the database at startup.
7. At this point, run the application as Spring Boot App. Open the HAL browser and point the browser to `http://localhost:8080`.
8. In the **Explorer** section, point to `http://localhost:8080/customers` and click on **Go**. This will list all the customers in the **Response Body** section of the HAL browser.
9. In the **Explorer** section, enter `http://localhost:8080/customers?size=2&page=1&sort=name` and click on **Go**. This will automatically execute paging and sorting on the repository and return the result.

   As the page size is set to `2` and the first page is requested, it will come back with two records in a sorted order.
10. Review the **Links** section. As shown in the following screenshot, it will facilitate navigating **first**, **next**, **prev**, and **last**. These are done using the HATEOAS links automatically generated by the repository browser:



11. Also, one can explore the details of a customer by selecting the appropriate link, such as `http://localhost:8080/customers/2`.
12. As the next step, add a controller class, `CustomerController`, to handle service endpoints. There is only one endpoint in this class, `/register`, which is used to register a customer. If successful, it returns the `Customer` object as the response, as follows:

```
@RestController
class CustomerController{

  @Autowired
  CustomerRegistrar customerRegistrar;

  @RequestMapping( path="/register", method = RequestMethod.POST)
  Customer register(@RequestBody Customer customer){
    return customerRegistrar.register(customer);
  }
}
```

13. A `CustomerRegistrar` component is added to handle the business logic. In this case, there is only minimal business logic added to the component. In this component class, while registering a customer, we will just check whether the customer name already exists in the database or not. If it does not exist, then we will insert a new record, and otherwise, we will send an error message back, as follows:

```
@Component
class CustomerRegistrar {

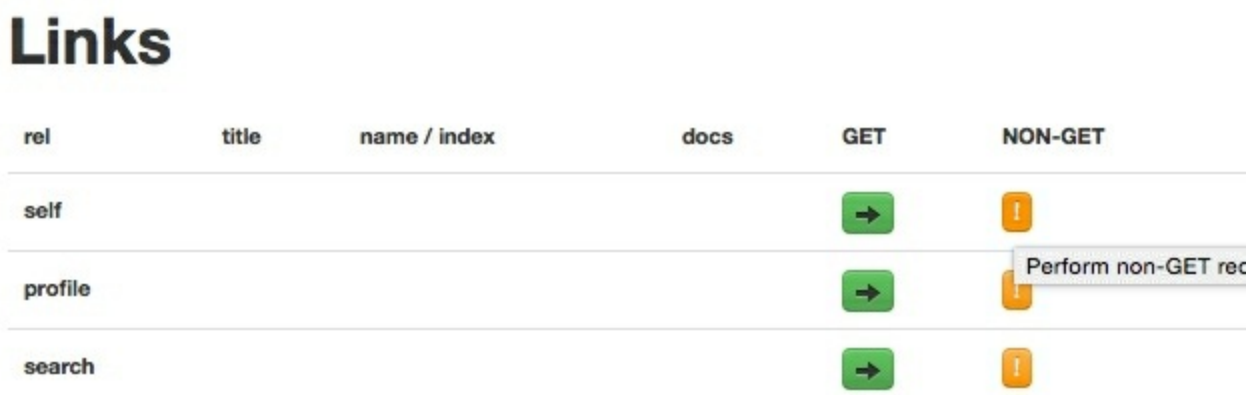
  CustomerRespository customerRespository;

  @Autowired
```

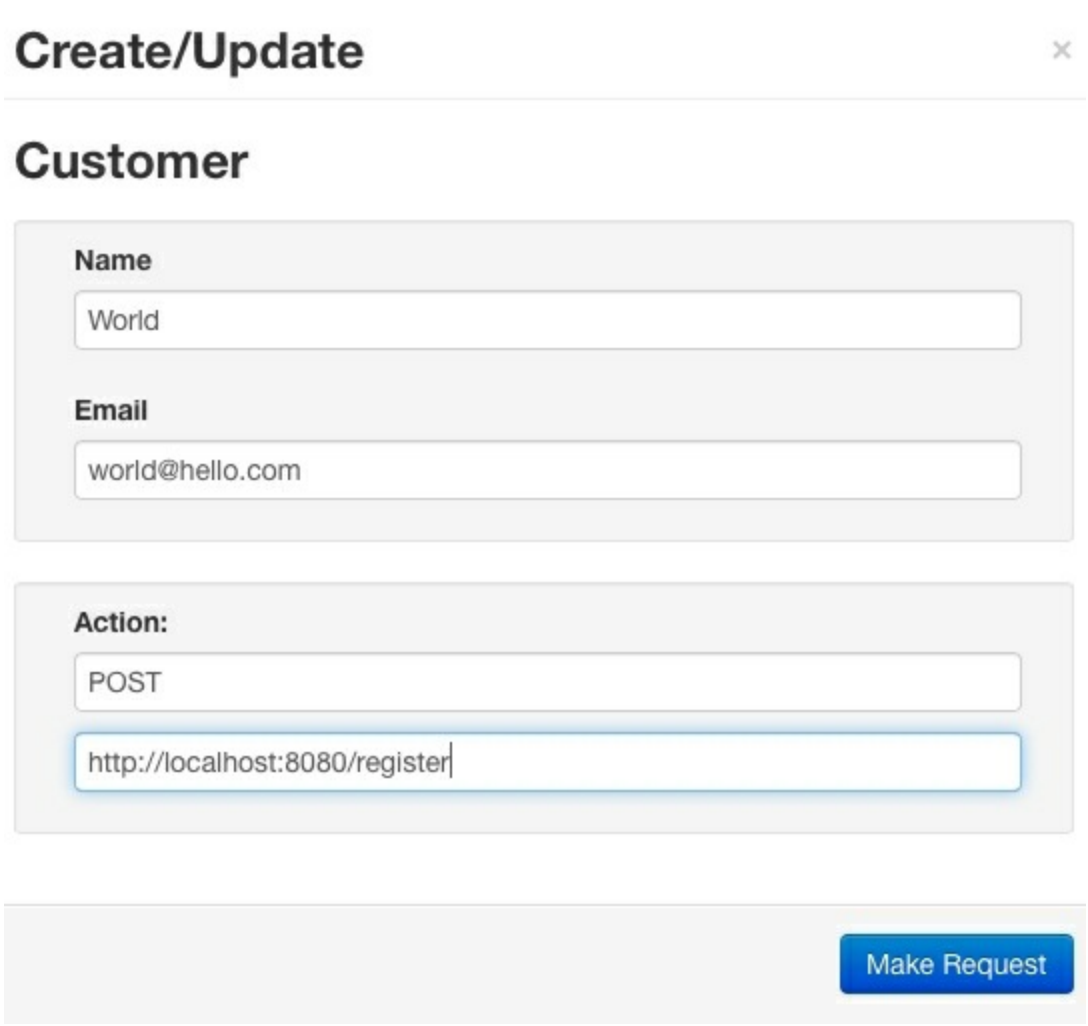
```
  CustomerRegistrar(CustomerRespository customerRespository){
    this.customerRespository = customerRespository;
  }

  Customer register(Customer customer){
    Optional<Customer> existingCustomer =
customerRespository.findByName(customer.getName());
    if (existingCustomer.isPresent()){
      throw new RuntimeException("is already exists");
    } else {
      customerRespository.save(customer);
    }
    return customer;
  }
}
```

14. Restart the Boot application and test using the HAL browser via the URL `http://localhost:8080`.
15. Point the **Explorer** field to `http://localhost:8080/customers`. Review the results in the **Links** section:



16. Click on the **NON-GET** option against **self**. This will open a form to create a new customer:



17. Fill the form and change the **Action** as shown in the diagram. Click on the **Make Request** button. This will call the register service and register the customer. Try giving a duplicate name to test the negative case.
18. Let's complete the last part in the example by integrating the Customer Notification service to notify the customer. When registration is successful, send an e-mail to the customer by asynchronously calling the Customer Notification microservice.
19. First update `CustomerRegistrar` to call the second service. This is done through messaging. In this case, we injected a `Sender` component to send a notification to the customer by passing the customer's e-mail address to the sender, as follows:

```
@Component
@Lazy
class CustomerRegistrar {

  CustomerRespository customerRespository;
  Sender sender;

  @Autowired
  CustomerRegistrar(CustomerRespository customerRespository, Sender sender){
    this.customerRespository = customerRespository;
    this.sender = sender;
  }
```

```
  Customer register(Customer customer){
    Optional<Customer> existingCustomer =
customerRespository.findByName(customer.getName());
    if (existingCustomer.isPresent()){
      throw new RuntimeException("is already exists");
    } else {
      customerRespository.save(customer);
      sender.send(customer.getEmail());
    }
    return customer;
  }
}
```

20. The sender component will be based on RabbitMQ and AMQP. In this example, `RabbitMessagingTemplate` is used as explored in the last messaging example; take a look at the following:

```
@Component
@Lazy
class Sender {

  @Autowired
  RabbitMessagingTemplate template;

  @Bean
  Queue queue() {
    return new Queue("CustomerQ", false);
  }

  public void send(String message){
    template.convertAndSend("CustomerQ", message);
  }
}
```

    The `@Lazy` annotation is a useful one and it helps to increase the boot startup time. These beans will be initialized only when the need arises.
21. We will also update the `application.property` file to include Rabbit MQ-related properties, as follows:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

22. We are ready to send the message. To consume the message and send e-mails, we will create a notification service. For this, let's create another Spring Boot service, `chapter2.bootcustomernotification`. Make sure that the **AMQP** and **Mail** starter libraries are selected when creating the Spring Boot service. Both **AMQP** and **Mail** are under **I/O**.
23. The package structure of the `chapter2.bootcustomernotification` project is as shown here:



24. Add a `Receiver` class. The `Receiver` class waits for a message on customer. This will receive a message sent by the Customer Profile service. On the arrival of a message, it sends an e-mail, as follows:

```
@Component
class Receiver {
  @Autowired
  Mailer mailer;

  @Bean
  Queue queue() {
    return new Queue("CustomerQ", false);
  }

  @RabbitListener(queues = "CustomerQ")
    public void processMessage(String email) {
       System.out.println(email);
       mailer.sendMail(email);
    }
}
```

25. Add another component to send an e-mail to the customer. We will use `JavaMailSender` to send an e-mail via the following code:

```
@Component
class Mailer {
  @Autowired
  private  JavaMailSender  javaMailService;
    public void sendMail(String email){
      SimpleMailMessage mailMessage=new SimpleMailMessage();
      mailMessage.setTo(email);
      mailMessage.setSubject("Registration");
      mailMessage.setText("Successfully Registered");
      javaMailService.send(mailMessage);
    }
}
```

Behind the scenes, Spring Boot automatically configures all the parameters required by

`JavaMailSender`.

26. To test SMTP, a test setup for SMTP is required to ensure that the mails are going out. In this example, FakeSMTP will be used. You can download FakeSMTP from http://nilhcem.github.io/FakeSMTP.
27. Once you download `fakeSMTP-2.0.jar`, run the SMTP server by executing the following command:

```
$ java -jar fakeSMTP-2.0.jar
```

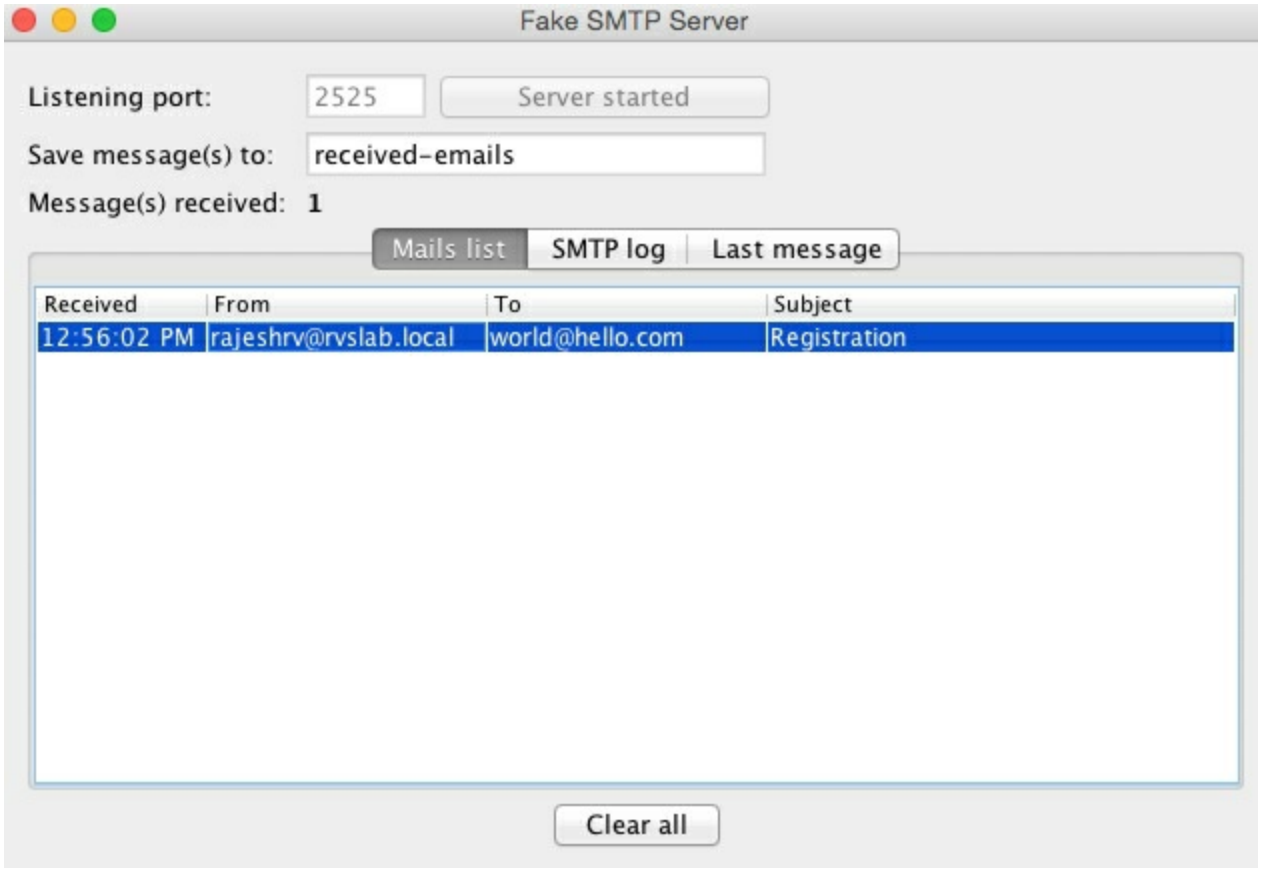
    This will open a GUI to monitor e-mail messages. Click on the **Start Server** button next to the listening port textbox.
28. Update `application.properties` with the following configuration parameters to connect to RabbitMQ as well as to the mail server:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

spring.mail.host=localhost
spring.mail.port=2525
```

29. We are ready to test our microservices end to end. Start both the Spring Boot apps. Open the browser and repeat the customer creation steps through the HAL browser. In this case, immediately after submitting the request, we will be able to see the e-mail in the SMTP GUI.

    Internally, the Customer Profile service asynchronously calls the Customer Notification service, which, in turn, sends the e-mail message to the SMTP server:


Fake SMTP Server
Listening port:
2525
Server started
Save message(s) to:
received-emails
Message(s) received:
1
Mails list
SMTP log
Last message
Received
From
To
Subject
12:56:02 PM
rajeshrv@rvslab.local
world@hello.com
Registration
Clear all

Spring Boot actuators

The previous sections explored most of the Spring Boot features required to develop a microservice. In this section, some of the production-ready operational aspects of Spring Boot will be explored.

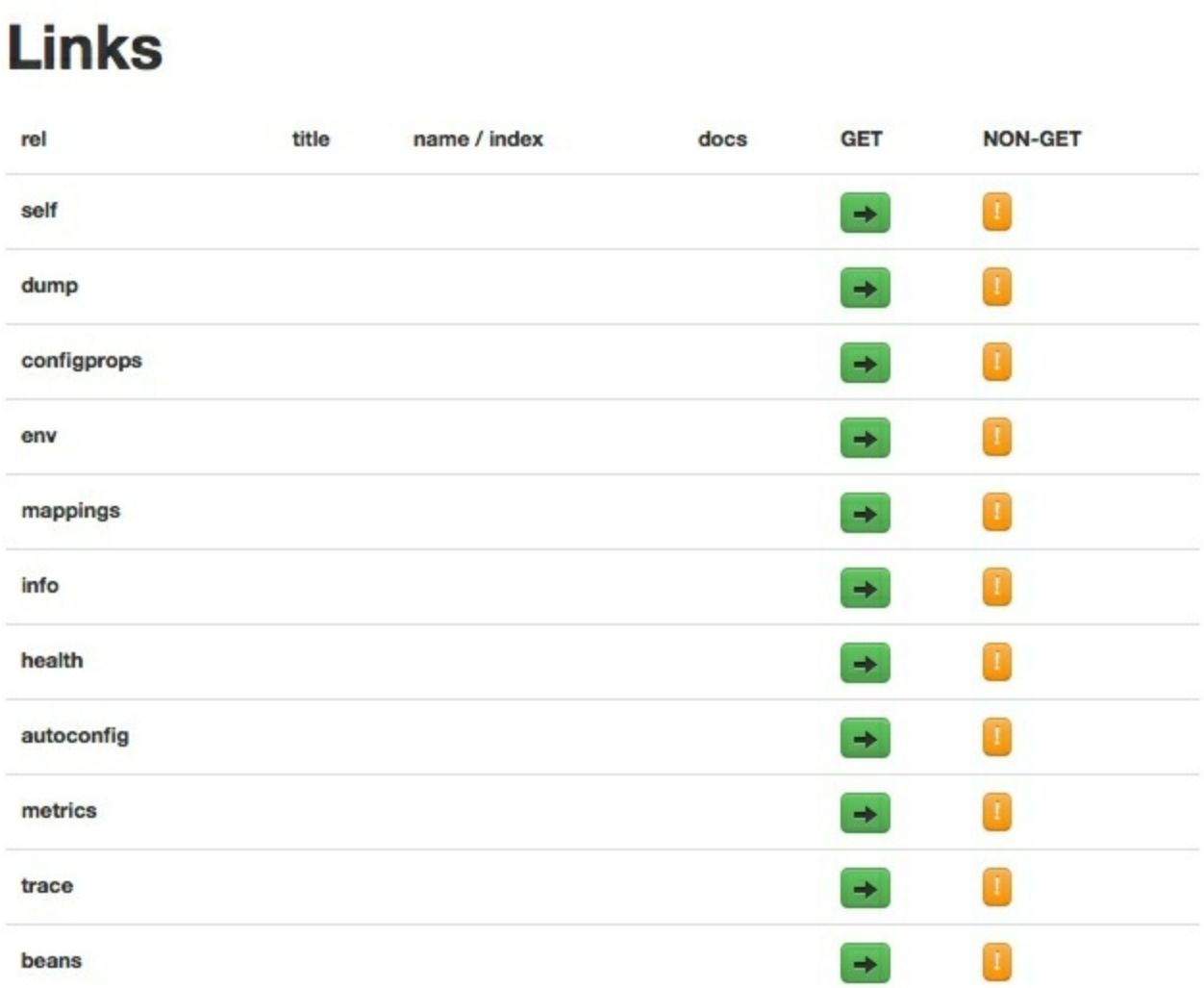
Spring Boot actuators provide an excellent out-of-the-box mechanism to monitor and manage Spring Boot applications in production:

Note

The full source code of this example is available as the `chapter2.bootactuator` project in the code files of this book.

1. Create another **Spring Starter Project** and name it `chapter2.bootactuator`. This time, select **Web** and **Actuators** under **Ops**. Similar to the `chapter2.bootrest` project, add a `GreeterController` endpoint with the `greet` method.
2. Start the application as Spring Boot app.
3. Point the browser to `localhost:8080/actuator`. This will open the HAL browser. Then, review the **Links** section.

   A number of links are available under the **Links** section. These are automatically exposed by the Spring Boot actuator:

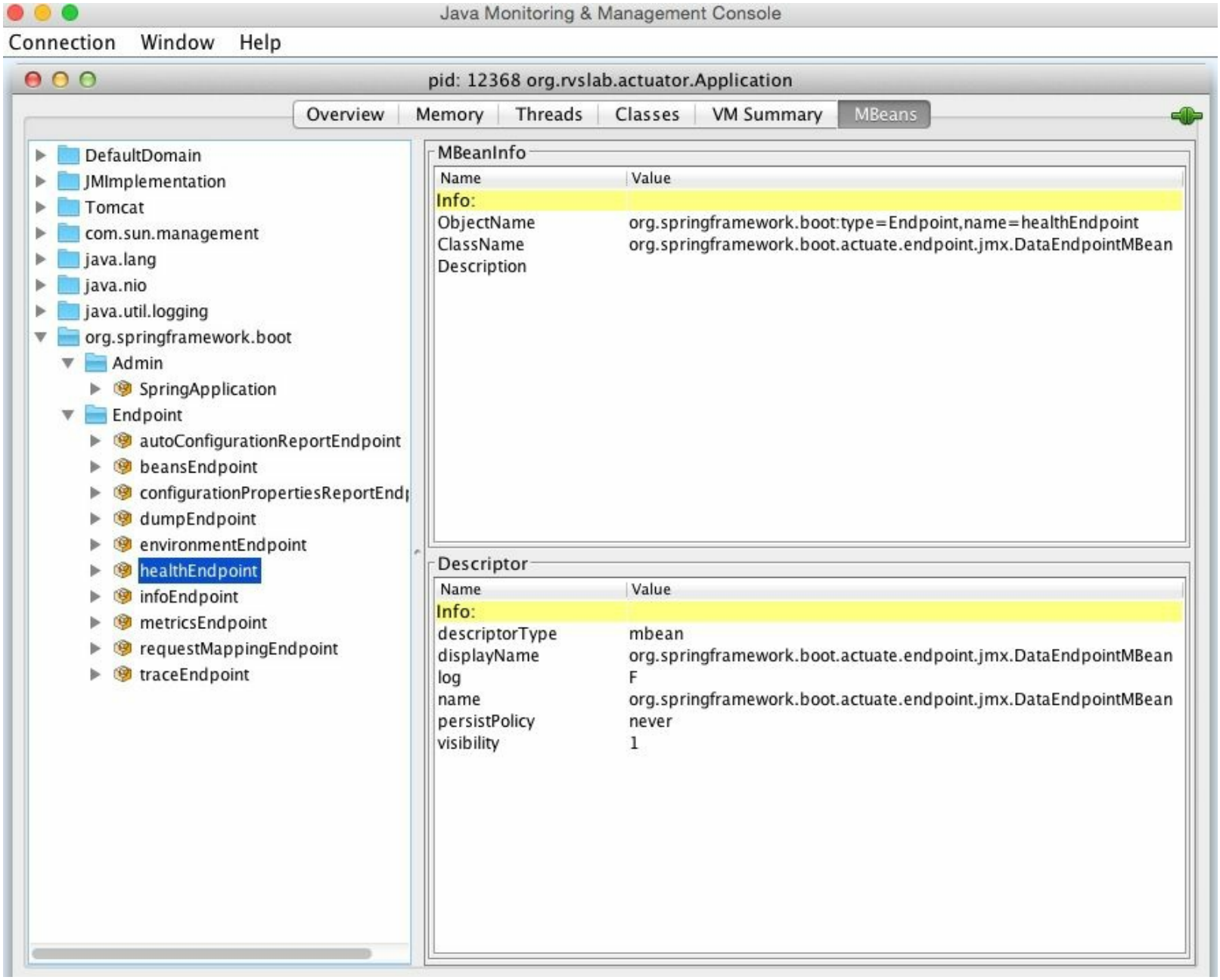


Some of the important links are listed as follows:

- `dump`: This performs a thread dump and displays the result
- `mappings`: This lists all the HTTP request mappings
- `info`: This displays information about the application
- `health`: This displays the application's health conditions
- `autoconfig`: This displays the autoconfiguration report
- `metrics`: This shows different metrics collected from the application

Monitoring using JConsole

Alternately, we can use the JMX console to see the Spring Boot information. Connect to the remote Spring Boot instance from JConsole. The Boot information will be shown as follows:



Monitoring using SSH

Spring Boot provides remote access to the Boot application using SSH. The following command connects to the Spring Boot application from a terminal window:

```
$ ssh -p 2000 user@localhost
```

The password can be customized by adding the `shell.auth.simple.user.password` property in the `application.properties` file. The updated `application.properties` file will look similar to the following:

```
shell.auth.simple.user.password=admin
```

When connected with the preceding command, similar actuator information can be accessed. Here is an example of the metrics information accessed through the CLI:

- `help`: This lists out all the options available
- `dashboard`: This is one interesting feature that shows a lot of system-level information

Configuring application information

The following properties can be set in `application.properties` to customize application-related information. After adding, restart the server and visit the `/info` endpoint of the actuator to take a look at the updated information, as follows:

```
info.app.name=Boot actuator
info.app.description= My Greetings Service
info.app.version=1.0.0
```

Adding a custom health module

Adding a new custom module to the Spring Boot application is not so complex. To demonstrate this feature, assume that if a service gets more than two transactions in a minute, then the server status will be set as Out of Service.

In order to customize this, we have to implement the `HealthIndicator` interface and override the `health` method. The following is a quick and dirty implementation to do the job:

```
class TPSCounter {
  LongAdder count;
  int threshold = 2;
  Calendar expiry = null;

  TPSCounter(){
    this.count = new LongAdder();
    this.expiry = Calendar.getInstance();
    this.expiry.add(Calendar.MINUTE, 1);
  }

  boolean isExpired(){
    return Calendar.getInstance().after(expiry);
  }

  boolean isWeak(){
    return (count.intValue() > threshold);
  }

  void increment(){
     count.increment();
  }
}
```

The preceding class is a simple POJO class that maintains the transaction counts in the window. The `isWeak` method checks whether the transaction in a particular window reached its threshold. The `isExpired` method checks whether the current window is expired or not. The `increment` method simply increases the counter value.

For the next step, implement our custom health indicator class, `TPSHealth`. This is done by extending `HealthIndicator`, as follows:

```
@Component
class TPSHealth implements HealthIndicator {
  TPSCounter counter;

@Override
    public Health health() {
        boolean health = counter.isWeak(); // perform some specific health check
        if (health) {
            return Health.outOfService().withDetail("Too many requests",
```

```
"OutofService").build();
        }
        return Health.up().build();
    }

    void updateTx(){
    if(counter == null || counter.isExpired()){
      counter = new TPSCounter();

    }
    counter.increment();
    }
}
```

The `health` method checks whether the counter is weak or not. A weak counter means the service is handling more transactions than it can handle. If it is weak, it marks the instance as Out of Service.

Finally, we will autowire `TPSHealth` into the `GreetingController` class and then call `health.updateTx()` in the `greet` method, as follows:

```
  Greet greet(){
    logger.info("Serving Request....!!!");
    health.updateTx();
    return new Greet("Hello World!");
  }
```

Go to the `/health` end point in the HAL browser and take a look at the status of the server.

Now, open another browser, point to `http://localhost:8080`, and fire the service twice or thrice. Go back to the `/health` endpoint and refresh to see the status. It should be changed to Out of Service.

In this example, as there is no action taken other than collecting the health status, even though the status is Out of Service, new service calls will still go through. However, in the real world, a program should read the `/health` endpoint and block further requests from going to this instance.

Building custom metrics

Similar to health, customization of the metrics is also possible. The following example shows how to add counter service and gauge service, just for demonstration purposes:

```
@Autowired
CounterService counterService;

@Autowired
GaugeService gaugeService;
```

Add the following methods in the greet method:

```
this.counterService.increment("greet.txnCount");
this.gaugeService.submit("greet.customgauge", 1.0);
```

Restart the server and go to `/metrics` to see the new gauge and counter added already reflected there.

Documenting microservices

The traditional approach of API documentation is either by writing service specification documents or using static service registries. With a large number of microservices, it would be hard to keep the documentation of APIs in sync.

Microservices can be documented in many ways. This section will explore how microservices can be documented using the popular Swagger framework. The following example will use Springfox libraries to generate REST API documentation. Springfox is a set of Java- and Spring-friendly libraries.

Create a new **Spring Starter Project** and select **Web** in the library selection window. Name the project `chapter2.swagger`.

Note

The full source code of this example is available as the `chapter2.swagger` project in the code files of this book.

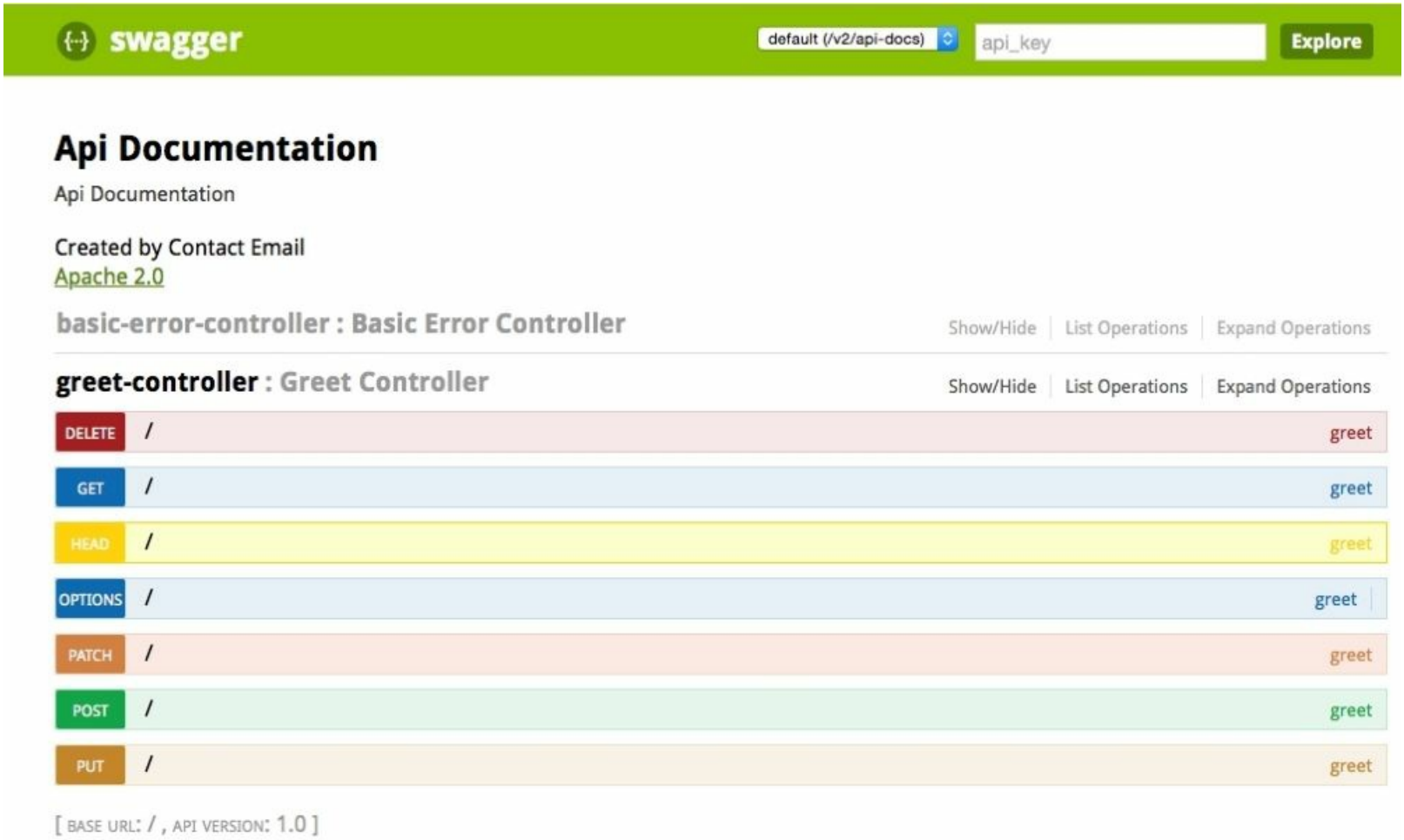
As Springfox libraries are not part of the Spring suite, edit `pom.xml` and add Springfox Swagger library dependencies. Add the following dependencies to the project:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.3.1</version>
</dependency>
```

Create a REST service similar to the services created earlier, but also add the `@EnableSwagger2` annotation, as follows:

```
@SpringBootApplication
@EnableSwagger2
public class Application {
```

This is all that's required for a basic Swagger documentation. Start the application and point the browser to `http://localhost:8080/swagger-ui.html`. This will open the Swagger API documentation page:



As shown in the diagram, the Swagger lists out the possible operations on **Greet Controller**. Click on the **GET** operation. This expands the **GET** row, which provides an option to try out the operation.

Summary

In this chapter, you learned about Spring Boot and its key features to build production-ready applications.

We explored the previous-generation web applications and then how Spring Boot makes developers' lives easier to develop fully qualified microservices. We also discussed the asynchronous message-based interaction between services. Further, we explored how to achieve some of the key capabilities required for microservices, such as security, HATEOAS, cross-origin, configurations, and so on with practical examples. We also took a look at how Spring Boot actuators help the operations teams and also how we can customize it to our needs. Finally, documenting microservices APIs was also explored.

In the next chapter, we will take a deeper look at some of the practical issues that may arise when implementing microservices. We will also discuss a capability model that essentially helps organizations when dealing with large microservices implementations.

Chapter 3. Applying Microservices Concepts

Microservices are good, but can also be an evil if they are not properly conceived. Wrong microservice interpretations could lead to irrecoverable failures.

This chapter will examine the technical challenges around practical implementations of microservices. It will also provide guidelines around critical design decisions for successful microservices development. The solutions and patterns for a number of commonly raised concerns around microservices will also be examined. This chapter will also review the challenges in enterprise scale microservices development, and how to overcome those challenges. More importantly, a capability model for a microservices ecosystem will be established at the end.

In this chapter you will learn about the following:

- Trade-offs between different design choices and patterns to be considered when developing microservices
- Challenges and anti-patterns in developing enterprise grade microservices
- A capability model for a microservices ecosystem

Patterns and common design decisions

Microservices have gained enormous popularity in recent years. They have evolved as the preferred choice of architects, putting SOA into the backyards. While acknowledging the fact that microservices are a vehicle for developing scalable cloud native systems, successful microservices need to be carefully designed to avoid catastrophes. Microservices are not the one-size-fits-all, universal solution for all architecture problems.

Generally speaking, microservices are a great choice for building a lightweight, modular, scalable, and distributed system of systems. Over-engineering, wrong use cases, and misinterpretations could easily turn the system into a disaster. While selecting the right use cases is paramount in developing a successful microservice, it is equally important to take the right design decisions by carrying out an appropriate trade-off analysis. A number of factors are to be considered when designing microservices, as detailed in the following sections.

Establishing appropriate microservice boundaries

One of the most common questions relating to microservices is regarding the size of the service. How big (mini-monolithic) or how small (nano service) can a microservice be, or is there anything like right-sized services? Does size really matter?

A quick answer could be "one REST endpoint per microservice", or "less than 300 lines of code", or "a component that performs a single responsibility". But before we pick up any of these answers, there is lot more analysis to be done to understand the boundaries for our services.

Domain-driven design (**DDD**) defines the concept of a **bounded context**. A bounded context is a subdomain or a subsystem of a larger domain or system that is responsible for performing a particular function.

Tip

Read more about DDD at http://domainlanguage.com/ddd/.

The following diagram is an example of the domain model:



In a finance back office, system invoices, accounting, billing, and the like represent different bounded contexts. These bounded contexts are strongly isolated domains that are closely aligned with business capabilities. In the financial domain, the invoices, accounting, and billing are different business capabilities often handled by different subunits under the finance department.

A bounded context is a good way to determine the boundaries of microservices. Each bounded context could be mapped to a single microservice. In the real world, communication between bounded contexts are typically less coupled, and often, disconnected.

Even though real world organizational boundaries are the simplest mechanisms for establishing a bounded context, these may prove wrong in some cases due to inherent problems within the organization's structures. For example, a business capability may be delivered through different channels such as front offices, online, roaming agents, and so on. In many organizations, the business units may be organized based on delivery channels rather than the actual underlying business capabilities. In such cases, organization boundaries may not provide accurate service boundaries.

A top-down domain decomposition could be another way to establish the right bounded contexts.

There is no silver bullet to establish microservices boundaries, and often, this is quite challenging. Establishing boundaries is much easier in the scenario of monolithic application to microservices migration, as the service boundaries and dependencies are known from the existing system. On the other hand, in a green field microservices development, the dependencies are hard to establish upfront.

The most pragmatic way to design microservices boundaries is to run the scenario at hand through a number of possible options, just like a service litmus test. Keep in mind that there may be multiple conditions matching a given scenario that will lead to a trade-off analysis.

The following scenarios could help in defining the microservice boundaries.

Autonomous functions

If the function under review is autonomous by nature, then it can be taken as a microservices boundary. Autonomous services typically would have fewer dependencies on external functions. They accept input, use its internal logic and data for computation, and return a result. All utility functions such as an encryption engine or a notification engine are straightforward candidates.

A delivery service that accepts an order, processes it, and then informs the trucking service is another example of an autonomous service. An online flight search based on cached seat availability information is yet another example of an autonomous function.

Size of a deployable unit

Most of the microservices ecosystems will take advantage of automation, such as automatic integration, delivery, deployment, and scaling. Microservices covering broader functions result in larger deployment units. Large deployment units pose challenges in automatic file copy, file download, deployment, and start up times. For instance, the size of a service increases with the density of the functions that it implements.

A good microservice ensures that the size of its deployable units remains manageable.

Most appropriate function or subdomain

It is important to analyze what would be the most useful component to detach from the monolithic application. This is particularly applicable when breaking monolithic applications into microservices. This could be based on parameters such as resource-intensiveness, cost of ownership, business benefits, or flexibility.

In a typical hotel booking system, approximately 50-60% of the requests are search-based. In this case, moving out the search function could immediately bring in flexibility, business benefits, cost reduction, resource free up, and so on.

Polyglot architecture

One of the key characteristics of microservices is its support for polyglot architecture. In order to meet different non-functional and functional requirements, components may require different treatments. It could be different architectures, different technologies, different deployment topologies, and so on. When components are identified, review them against the requirement for polyglot architectures.

In the hotel booking scenario mentioned earlier, a Booking microservice may need transactional integrity, whereas a Search microservice may not. In this case, the Booking microservice may use an ACID compliance database such as MySQL, whereas the Search microservice may use an eventual consistent database such as Cassandra.

Selective scaling

Selective scaling is related to the previously discussed polyglot architecture. In this context, all functional modules may not require the same level of scalability. Sometimes, it may be appropriate to determine boundaries based on scalability requirements.

For example, in the hotel booking scenario, the Search microservice has to scale considerably more than many of the other services such as the Booking microservice or the Notification microservice due to the higher velocity of search requests. In this case, a separate Search microservice could run on top of an Elasticsearch or an in-memory data grid for better response.

Small, agile teams

Microservices enable Agile development with small, focused teams developing different parts of the pie. There could be scenarios where parts of the systems are built by different organizations, or even across different geographies, or by teams with varying skill sets. This approach is a common practice, for example, in manufacturing industries.

In the microservices world, each of these teams builds different microservices, and then assembles them together. Though this is not the desired way to break down the system, organizations may end up in such situations. Hence, this approach cannot be completely ruled out.

In an online product search scenario, a service could provide personalized options based on what the customer is looking for. This may require complex machine learning algorithms, and hence need a specialist team. In this scenario, this function could be built as a microservice by a separate specialist team.

Single responsibility

In theory, the single responsibility principle could be applied at a method, at a class, or at a service. However, in the context of microservices, it does not necessarily map to a single service or endpoint.

A more practical approach could be to translate single responsibility into single business capability

or a single technical capability. As per the *single responsibility* principle, one responsibility cannot be shared by multiple microservices. Similarly, one microservice should not perform multiple responsibilities.

There could, however, be special cases where a single business capability is divided across multiple services. One of such cases is managing the customer profile, where there could be situations where you may use two different microservices for managing reads and writes using a **Command Query Responsibility Segregation (CQRS)** approach to achieve some of the quality attributes.

Replicability or changeability

Innovation and speed are of the utmost importance in IT delivery. Microservices boundaries should be identified in such a way that each microservice is easily detachable from the overall system, with minimal cost of re-writing. If part of the system is just an experiment, it should ideally be isolated as a microservice.

An organization may develop a recommendation engine or a customer ranking engine as an experiment. If the business value is not realized, then throw away that service, or replace it with another one.

Many organizations follow the startup model, where importance is given to meeting functions and quick delivery. These organizations may not worry too much about the architecture and technologies. Instead, the focus will be on what tools or technologies can deliver solutions faster. Organizations increasingly choose the approach of developing **Minimum Viable Products (MVPs)** by putting together a few services, and allowing the system to evolve. Microservices play a vital role in such cases where the system evolves, and services gradually get rewritten or replaced.

Coupling and cohesion

Coupling and cohesion are two of the most important parameters for deciding service boundaries. Dependencies between microservices have to be evaluated carefully to avoid highly coupled interfaces. A functional decomposition, together with a modeled dependency tree, could help in establishing a microservices boundary. Avoiding too chatty services, too many synchronous request-response calls, and cyclic synchronous dependencies are three key points, as these could easily break the system. A successful equation is to keep high cohesion within a microservice, and loose coupling between microservices. In addition to this, ensure that transaction boundaries are not stretched across microservices. A first class microservice will react upon receiving an event as an input, execute a number of internal functions, and finally send out another event. As part of the compute function, it may read and write data to its own local store.

Think microservice as a product

DDD also recommends mapping a bounded context to a product. As per DDD, each bounded context is an ideal candidate for a product. Think about a microservice as a product by itself. When microservice boundaries are established, assess them from a product's point of view to see whether

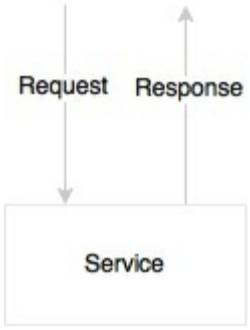
they really stack up as product. It is much easier for business users to think boundaries from a product point of view. A product boundary may have many parameters, such as a targeted community, flexibility in deployment, sell-ability, reusability, and so on.

Designing communication styles

Communication between microservices can be designed either in synchronous (request-response) or asynchronous (fire and forget) styles.

Synchronous style communication

The following diagram shows an example request/response style service:



In synchronous communication, there is no shared state or object. When a caller requests a service, it passes the required information and waits for a response. This approach has a number of advantages.

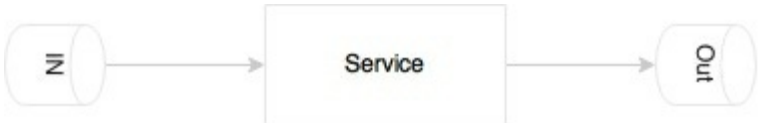
An application is stateless, and from a high availability standpoint, many active instances can be up and running, accepting traffic. Since there are no other infrastructure dependencies such as a shared messaging server, there are management fewer overheads. In case of an error at any stage, the error will be propagated back to the caller immediately, leaving the system in a consistent state, without compromising data integrity.

The downside in a synchronous request-response communication is that the user or the caller has to wait until the requested process gets completed. As a result, the calling thread will wait for a response, and hence, this style could limit the scalability of the system.

A synchronous style adds hard dependencies between microservices. If one service in the service chain fails, then the entire service chain will fail. In order for a service to succeed, all dependent services have to be up and running. Many of the failure scenarios have to be handled using timeouts and loops.

Asynchronous style communication

The following diagram is a service designed to accept an asynchronous message as input, and send the response asynchronously for others to consume:



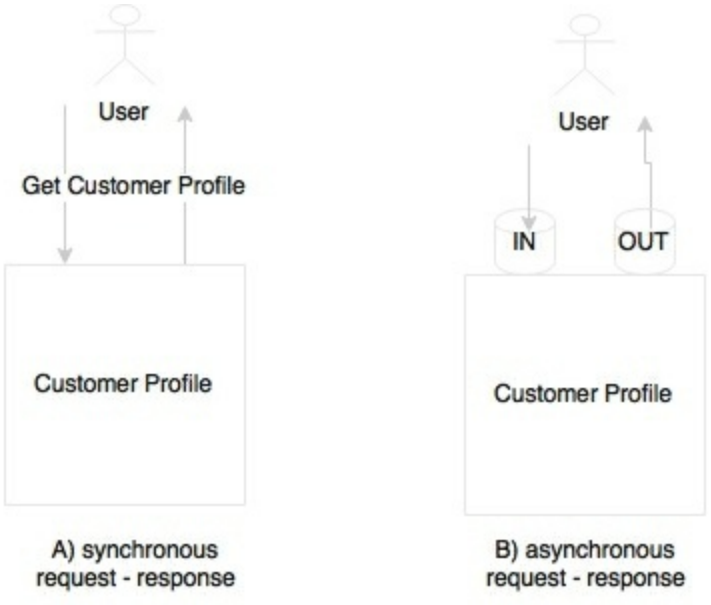
The asynchronous style is based on reactive event loop semantics which decouple microservices. This approach provides higher levels of scalability, because services are independent, and can internally spawn threads to handle an increase in load. When overloaded, messages will be queued in a messaging server for later processing. That means that if there is a slowdown in one of the services, it will not impact the entire chain. This provides higher levels of decoupling between services, and therefore maintenance and testing will be simpler.

The downside is that it has a dependency to an external messaging server. It is complex to handle the fault tolerance of a messaging server. Messaging typically works with an active/passive semantics. Hence, handling continuous availability of messaging systems is harder to achieve. Since messaging typically uses persistence, a higher level of I/O handling and tuning is required.

How to decide which style to choose?

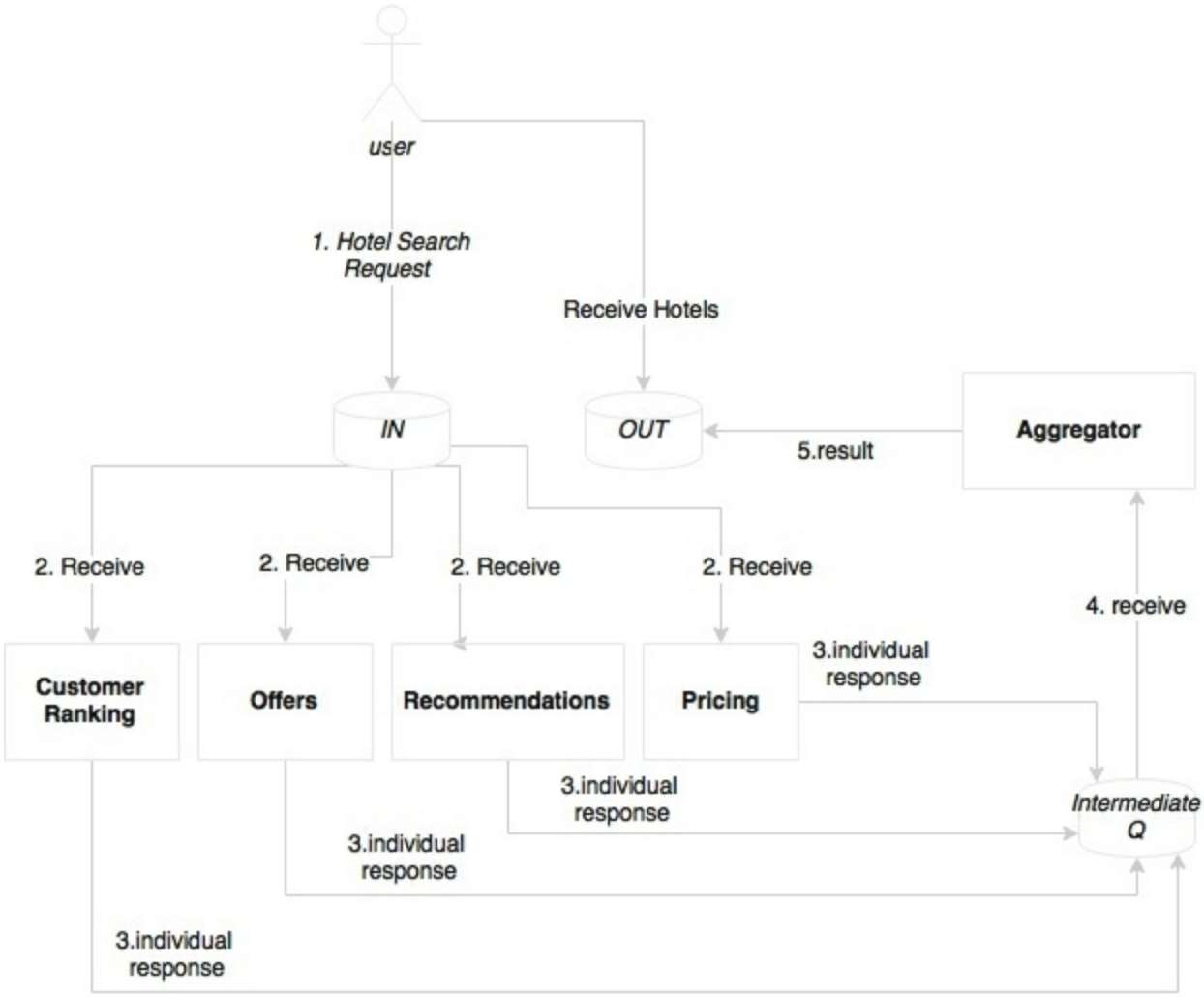
Both approaches have their own merits and constraints. It is not possible to develop a system with just one approach. A combination of both approaches is required based on the use cases. In principle, the asynchronous approach is great for building true, scalable microservice systems. However, attempting to model everything as asynchronous leads to complex system designs.

How does the following example look in the context where an end user clicks on a UI to get profile details?



This is perhaps a simple query to the backend system to get a result in a request-response model. This can also be modeled in an asynchronous style by pushing a message to an input queue, and waiting for a response in an output queue till a response is received for the given correlation ID. However, though we use asynchronous messaging, the user is still blocked for the entire duration of the query.

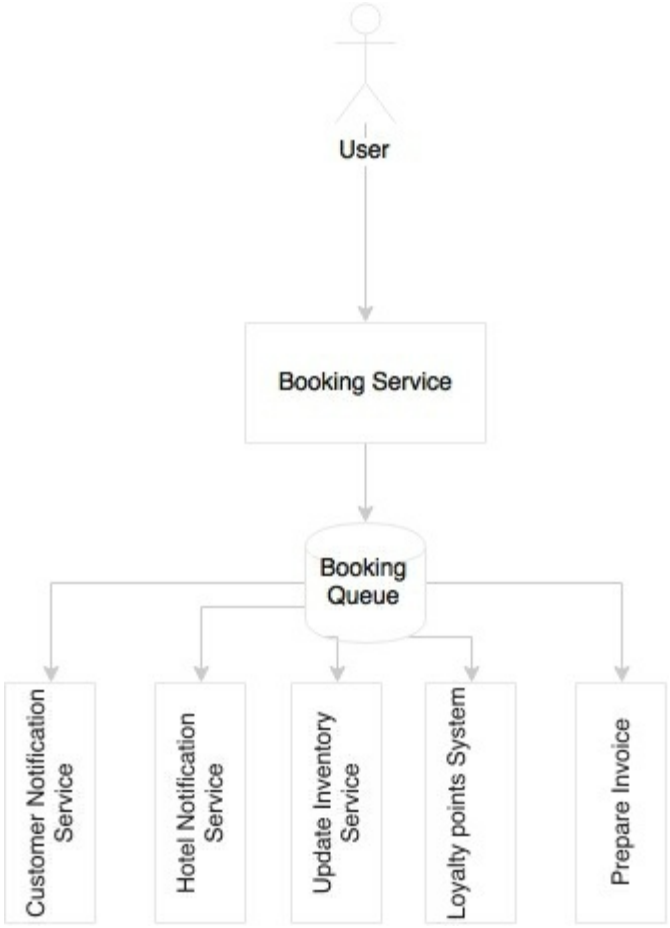
Another use case is that of a user clicking on a UI to search hotels, which is depicted in the following diagram:



This is very similar to the previous scenario. However, in this case, we assume that this business function triggers a number of activities internally before returning the list of hotels back to the user. For example, when the system receives this request, it calculates the customer ranking, gets offers based on the destination, gets recommendations based on customer preferences, optimizes the prices based on customer values and revenue factors, and so on. In this case, we have an opportunity to do many of these activities in parallel so that we can aggregate all these results before presenting them to the customer. As shown in the preceding diagram, virtually any computational logic could be plugged in to the search pipeline listening to the **IN** queue.

An effective approach in this case is to start with a synchronous request response, and refactor later to introduce an asynchronous style when there is value in doing that.

The following example shows a fully asynchronous style of service interactions:



The service is triggered when the user clicks on the booking function. It is again, by nature, a synchronous style communication. When booking is successful, it sends a message to the customer's e-mail address, sends a message to the hotel's booking system, updates the cached inventory, updates the loyalty points system, prepares an invoice, and perhaps more. Instead of pushing the user into a long wait state, a better approach is to break the service into pieces. Let the user wait till a booking record is created by the Booking Service. On successful completion, a booking event will be published, and return a confirmation message back to the user. Subsequently, all other activities will happen in parallel, asynchronously.

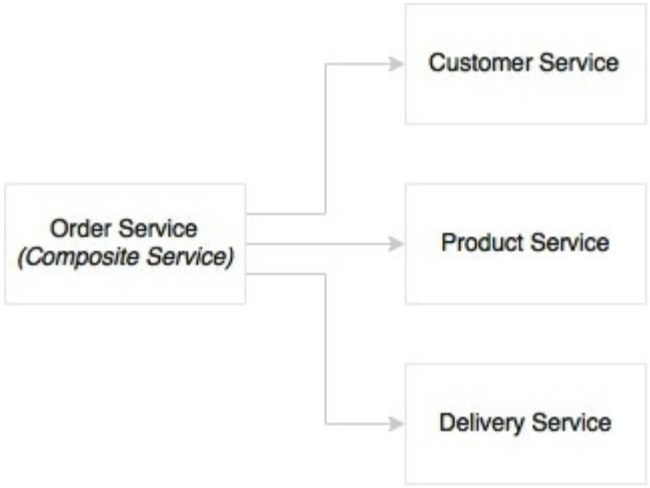
In all three examples, the user has to wait for a response. With the new web application frameworks, it is possible to send requests asynchronously, and define the callback method, or set an observer for getting a response. Therefore, the users won't be fully blocked from executing other activities.

In general, an asynchronous style is always better in the microservices world, but identifying the right pattern should be purely based on merits. If there are no merits in modeling a transaction in an asynchronous style, then use the synchronous style till you find an appealing case. Use reactive programming frameworks to avoid complexity when modeling user-driven requests, modeled in an asynchronous style.

Orchestration of microservices

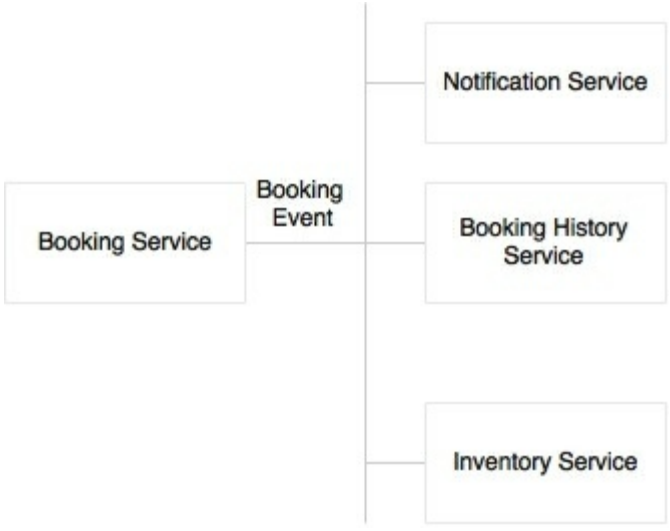
Composability is one of the service design principles. This leads to confusion around who is responsible for the composing services. In the SOA world, ESBs are responsible for composing a set of finely-grained services. In some organizations, ESBs play the role of a proxy, and service providers themselves compose and expose coarse-grained services. In the SOA world, there are two approaches for handling such situations.

The first approach is orchestration, which is depicted in the following diagram:



In the orchestration approach, multiple services are stitched together to get a complete function. A central brain acts as the orchestrator. As shown in the diagram, the order service is a composite service that will orchestrate other services. There could be sequential as well as parallel branches from the master process. Each task will be fulfilled by an atomic task service, typically a web service. In the SOA world, ESBs play the role of orchestration. The orchestrated service will be exposed by ESBs as a composite service.

The second approach is choreography, which is shown in the following diagram:

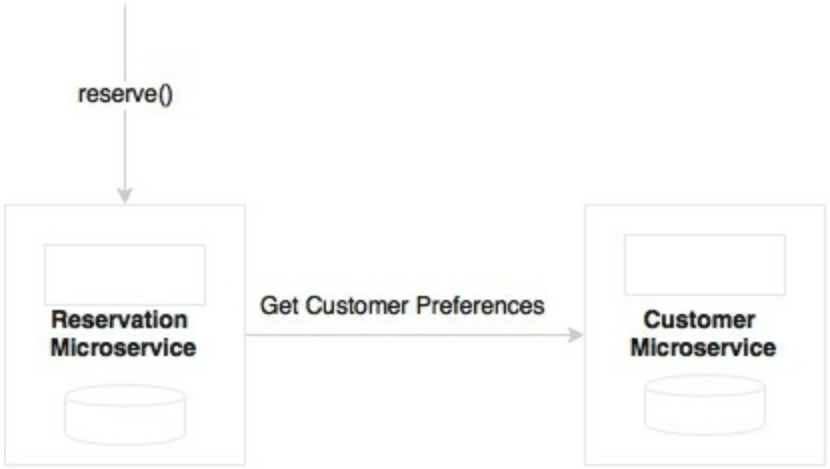


In the choreography approach, there is no central brain. An event, a booking event in this case, is published by a producer, a number of consumers wait for the event, and independently apply different logics on the incoming event. Sometimes, events could even be nested where the consumers can send another event which will be consumed by another service. In the SOA world, the caller pushes a message to the ESB, and the downstream flow will be automatically determined by the consuming services.

Microservices are autonomous. This essentially means that in an ideal situation, all required components to complete their function should be within the service. This includes the database, orchestration of its internal services, intrinsic state management, and so on. The service endpoints provide coarse-grained APIs. This is perfectly fine as long as there are no external touch points required. But in reality, microservices may need to talk to other microservices to fulfil their function.

In such cases, choreography is the preferred approach for connecting multiple microservices together. Following the autonomy principle, a component sitting outside a microservice and controlling the flow is not the desired option. If the use case can be modeled in choreographic style, that would be the best possible way to handle the situation.

But it may not be possible to model choreography in all cases. This is depicted in the following diagram:

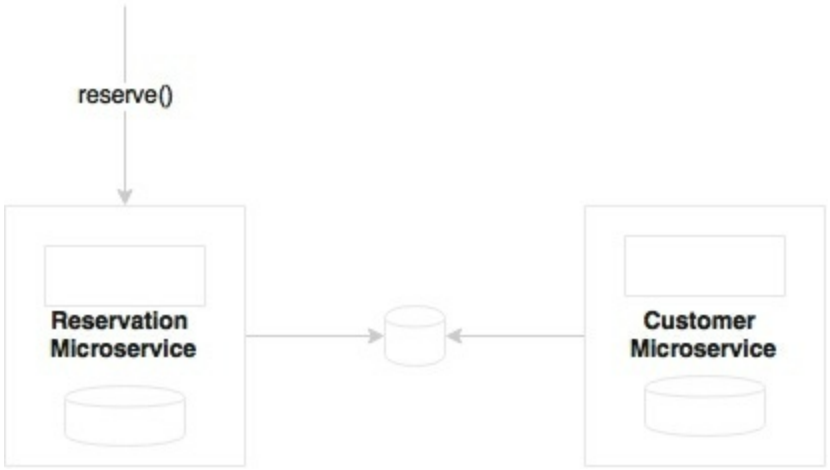


In the preceding example, Reservation and Customer are two microservices, with clearly segregated functional responsibilities. A case could arise when Reservation would want to get Customer preferences while creating a reservation. These are quite normal scenarios when developing complex systems.

Can we move Customer to Reservation so that Reservation will be complete by itself? If Customer and Reservation are identified as two microservices based on various factors, it may not be a good idea to move Customer to Reservation. In such a case, we will meet another monolithic application

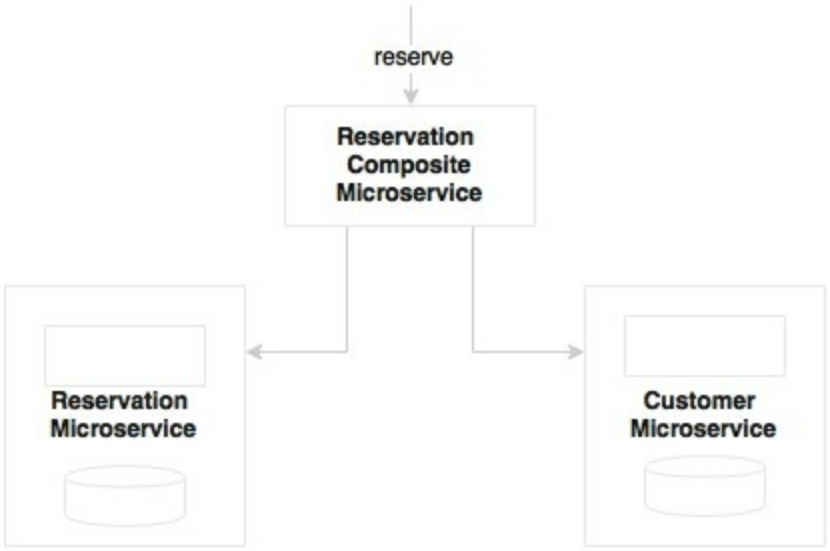
sooner or later.

Can we make the Reservation to Customer call asynchronous? This example is shown in the following diagram:



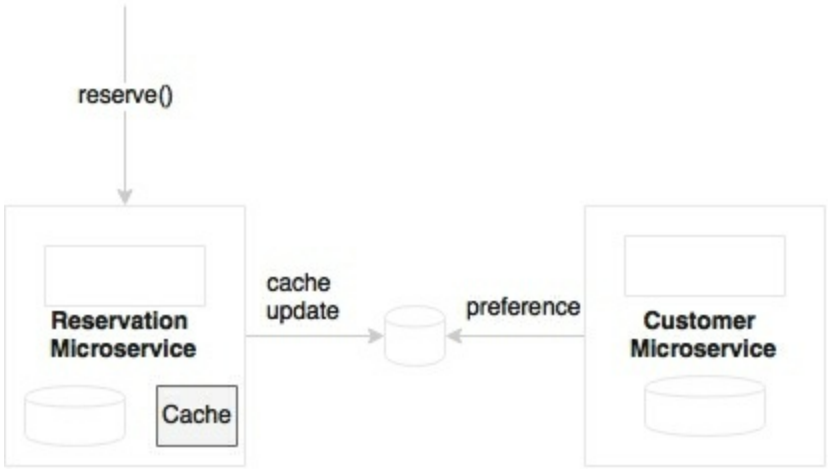
Customer preference is required for Reservation to progress, and hence, it may require a synchronous blocking call to Customer. Retrofitting this by modeling asynchronously does not really make sense.

Can we take out just the orchestration bit, and create another composite microservice, which then composes Reservation and Customer?



This is acceptable in the approach for composing multiple components within a microservice. But creating a composite microservice may not be a good idea. We will end up creating many microservices with no business alignment, which would not be autonomous, and could result in many fine-grained microservices.

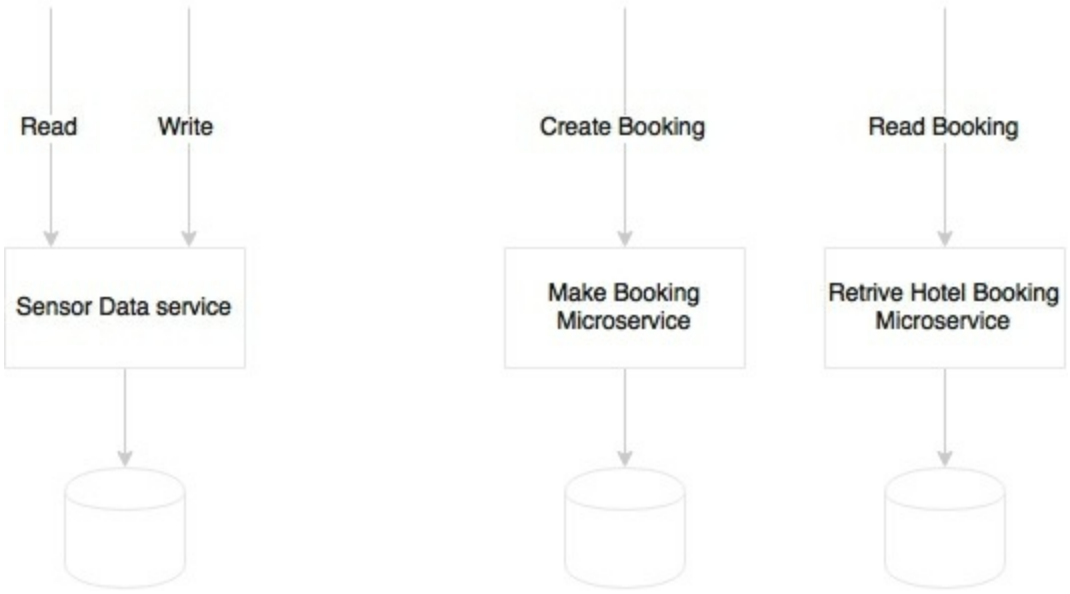
Can we duplicate customer preference by keeping a slave copy of the preference data into Reservation?



Changes will be propagated whenever there is a change in the master. In this case, Reservation can use customer preference without fanning out a call. It is a valid thought, but we need to carefully analyze this. Today we replicate customer preference, but in another scenario, we may want to reach out to customer service to see whether the customer is black-listed from reserving. We have to be extremely careful in deciding what data to duplicate. This could add to the complexity.

How many endpoints in a microservice?

In many situations, developers are confused with the number of endpoints per microservice. The question really is whether to limit each microservice with one endpoint or multiple endpoints:



The number of endpoints is not really a decision point. In some cases, there may be only one endpoint, whereas in some other cases, there could be more than one endpoint in a microservice. For instance, consider a sensor data service which collects sensor information, and has two logical endpoints: create and read. But in order to handle CQRS, we may create two separate physical microservices as shown in the case of **Booking** in the preceding diagram. Polyglot architecture could be another scenario where we may split endpoints into different microservices.

Considering a notification engine, notifications will be send out in response to an event. The process of notification such as preparation of data, identification of a person, and delivery mechanisms, are different for different events. Moreover, we may want to scale each of these processes differently at different time windows. In such situations, we may decide to break each notification endpoint in to a separate microservice.

In yet another example, a Loyalty Points microservice may have multiple services such as accrue, redeem, transfer, and balance. We may not want to treat each of these services differently. All of these services are connected and use the points table for data. If we go with one endpoint per service, we will end up in a situation where many fine-grained services access data from the same data store or replicated copies of the same data store.

In short, the number of endpoints is not a design decision. One microservice may host one or more endpoints. Designing appropriate bounded context for a microservice is more important.

One microservice per VM or multiple?

One microservice could be deployed in multiple **Virtual Machines** (**VMs**) by replicating the deployment for scalability and availability. This is a no brainer. The question is whether multiple microservices could be deployed in one virtual machine? There are pros and cons for this approach. This question typically arises when the services are simple, and the traffic volume is less.

Consider an example where we have a couple of microservices, and the overall transaction per minute is less than 10. Also assume that the smallest possible VM size available is 2-core 8 GB RAM. A further assumption is that in such cases, a 2-core 8 GB VM can handle 10-15 transactions per minute without any performance concerns. If we use different VMs for each microservice, it may not be cost effective, and we will end up paying more for infrastructure and license, since many vendors charge based on the number of cores.

The simplest way to approach this problem is to ask a few questions:

- Does the VM have enough capacity to run both services under peak usage?
- Do we want to treat these services differently to achieve SLAs (selective scaling)? For example, for scalability, if we have an all-in-one VM, we will have to replicate VMs which replicate all services.
- Are there any conflicting resource requirements? For example, different OS versions, JDK versions, and others.

If all your answers are *No*, then perhaps we can start with collocated deployment, until we encounter a scenario to change the deployment topology. However, we will have to ensure that these services are not sharing anything, and are running as independent OS processes.

Having said that, in an organization with matured virtualized infrastructure or cloud infrastructure, this may not be a huge concern. In such environments, the developers need not worry about where the services are running. Developers may not even think about capacity planning. Services will be deployed in a compute cloud. Based on the infrastructure availability, SLAs and the nature of the service, the infrastructure self-manages deployments. AWS Lambda is a good example of such a service.

Rules engine – shared or embedded?

Rules are an essential part of any system. For example, an offer eligibility service may execute a number of rules before making a yes or no decision. Either we hand code rules, or we may use a rules engine. Many enterprises manage rules centrally in a rules repository as well as execute them centrally. These enterprise rule engines are primarily used for providing the business an opportunity to author and manage rules as well as reuse rules from the central repository. **Drools** is one of the popular open source rules engines. IBM, FICO, and Bosch are some of the pioneers in the commercial space. These rule engines improve productivity, enable reuse of rules, facts, vocabularies, and provide faster rule execution using the rete algorithm.

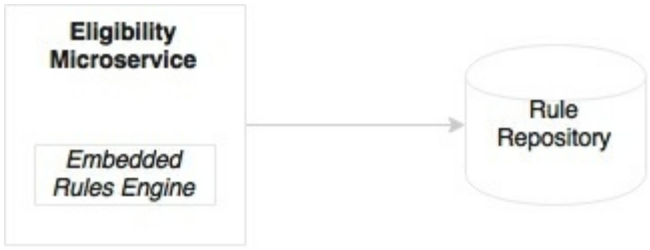
In the context of microservices, a central rules engine means fanning out calls from microservices to the central rules engine. This also means that the service logic is now in two places, some within the service, and some external to the service. Nevertheless, the objective in the context of microservices is to reduce external dependencies:



If the rules are simple enough, few in numbers, only used within the boundaries of a service, and not exposed to business users for authoring, then it may be better to hand-code business rules than rely on an enterprise rule engine:



If the rules are complex, limited to a service context, and not given to business users, then it is better to use an embedded rules engine within the service:



If the rules are managed and authored by business, or if the rules are complex, or if we are reusing rules from other service domains, then a central authoring repository with a locally embedded execution engine could be a better choice.

Note that this has to be carefully evaluated since all vendors may not support the local rule execution approach, and there could be technology dependencies such as running rules only within a specific application server, and so on.

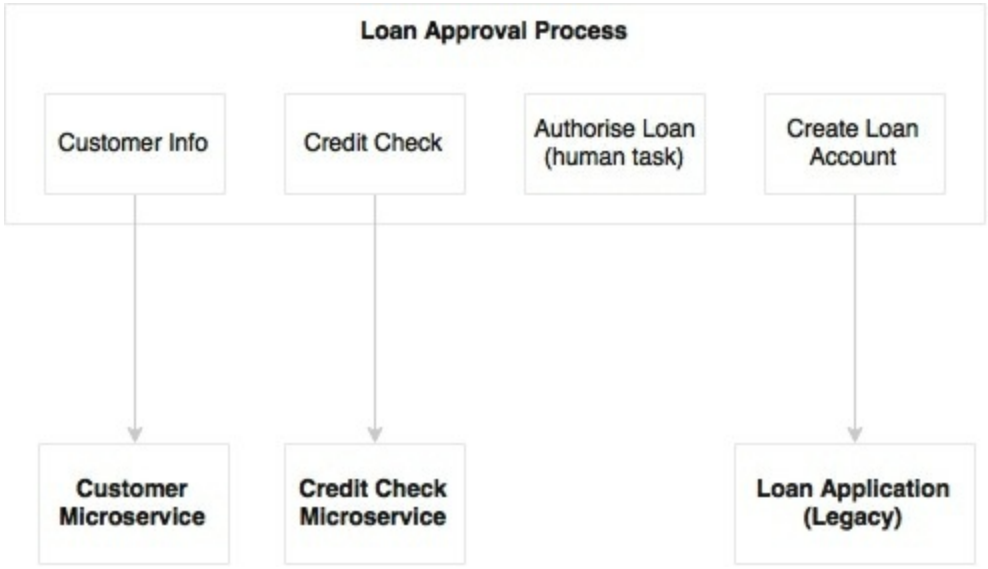
Role of BPM and workflows

Business Process Management (**BPM**) and **Intelligent Business Process Management** (**iBPM**) are tool suites for designing, executing, and monitoring business processes.

Typical use cases for BPM are:

- Coordinating a long-running business process, where some processes are realized by existing assets, whereas some other areas may be niche, and there is no concrete implementation of the processes being in place. BPM allows composing both types, and provides an end-to-end automated process. This often involves systems and human interactions.
- Process-centric organizations, such as those that have implemented Six Sigma, want to monitor their processes for continuous improvement on efficiency.
- Process re-engineering with a top-down approach by redefining the business process of an organization.

There could be two scenarios where BPM fits in the microservices world:



The first scenario is business process re-engineering, or threading an end-to-end long running business process, as stated earlier. In this case, BPM operates at a higher level, where it may automate a cross-functional, long-running business process by stitching a number of coarse-grained microservices, existing legacy connectors, and human interactions. As shown in the preceding diagram, the loan approval BPM invokes microservices as well as legacy application services. It also integrates human tasks.

In this case, microservices are headless services that implement a subprocess. From the microservices' perspective, BPM is just another consumer. Care needs to be taken in this approach to avoid accepting a shared state from a BPM process as well as moving business logic to BPM:

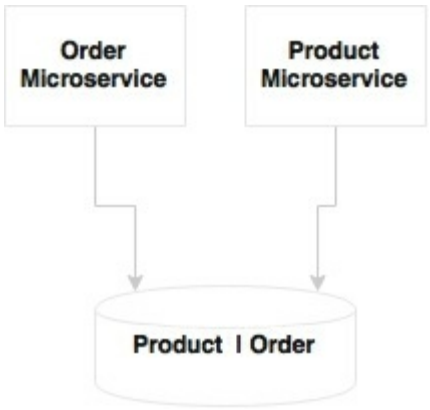


The second scenario is monitoring processes, and optimizing them for efficiency. This goes hand in hand with a completely automated, asynchronously choreographed microservices ecosystem. In this case, microservices and BPM work as independent ecosystems. Microservices send events at various timeframes such as the start of a process, state changes, end of a process, and so on. These events are used by the BPM engine to plot and monitor process states. We may not require a full-fledged BPM solution for this, as we are only mocking a business process to monitor its efficiency. In this case, the order delivery process is not a BPM implementation, but it is more of a monitoring dashboard that captures and displays the progress of the process.

To summarize, BPM could still be used at a higher level for composing multiple microservices in situations where end-to-end cross-functional business processes are modeled by automating systems and human interactions. A better and simpler approach is to have a business process dashboard to which microservices feed state change events as mentioned in the second scenario.

Can microservices share data stores?

In principle, microservices should abstract presentation, business logic, and data stores. If the services are broken as per the guidelines, each microservice logically could use an independent database:

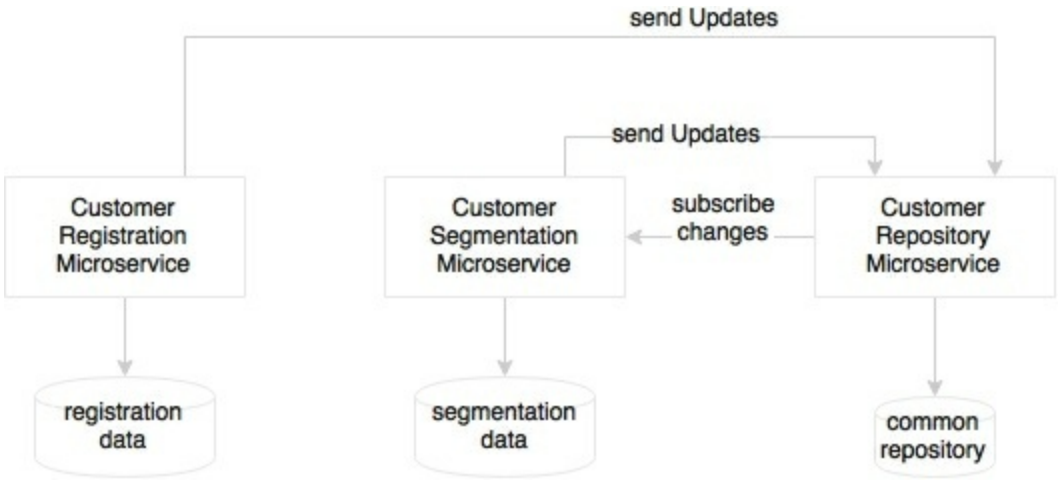


In the preceding diagram, both **Product** and **Order** microservices share one database and one data model. Shared data models, shared schema, and shared tables are recipes for disasters when developing microservices. This may be good at the beginning, but when developing complex microservices, we tend to add relationships between data models, add join queries, and so on. This can result in tightly coupled physical data models.

If the services have only a few tables, it may not be worth investing a full instance of a database like an Oracle database instance. In such cases, a schema level segregation is good enough to start with:



There could be scenarios where we tend to think of using a shared database for multiple services. Taking an example of a customer data repository or master data managed at the enterprise level, the customer registration and customer segmentation microservices logically share the same customer data repository:



As shown in the preceding diagram, an alternate approach in this scenario is to separate the transactional data store for microservices from the enterprise data repository by adding a local transactional data store for these services. This will help the services to have flexibility in remodeling the local data store optimized for its purpose. The enterprise customer repository sends change events when there is any change in the customer data repository. Similarly, if there is any change in any of the transactional data stores, the changes have to be sent to the central customer repository.

Setting up transaction boundaries

Transactions in operational systems are used to maintain the consistency of data stored in an RDBMS by grouping a number of operations together into one atomic block. They either commit or rollback the entire operation. Distributed systems follow the concept of distributed transactions with a two-phase commit. This is particularly required if heterogeneous components such as an RPC service, JMS, and so on participate in a transaction.

Is there a place for transactions in microservices? Transactions are not bad, but one should use transactions carefully, by analyzing what we are trying do.

For a given microservice, an RDBMS like MySQL may be selected as a backing store to ensure 100% data integrity, for example, a stock or inventory management service where data integrity is key. It is appropriate to define transaction boundaries within the microsystem using local transactions. However, distributed global transactions should be avoided in the microservices context. Proper dependency analysis is required to ensure that transaction boundaries do not span across two different microservices as much as possible.

Altering use cases to simplify transactional requirements

Eventual consistency is a better option than distributed transactions that span across multiple microservices. Eventual consistency reduces a lot of overheads, but application developers may need to re-think the way they write application code. This could include remodeling functions, sequencing operations to minimize failures, batching insert and modify operations, remodeling data structure, and finally, compensating operations that negate the effect.

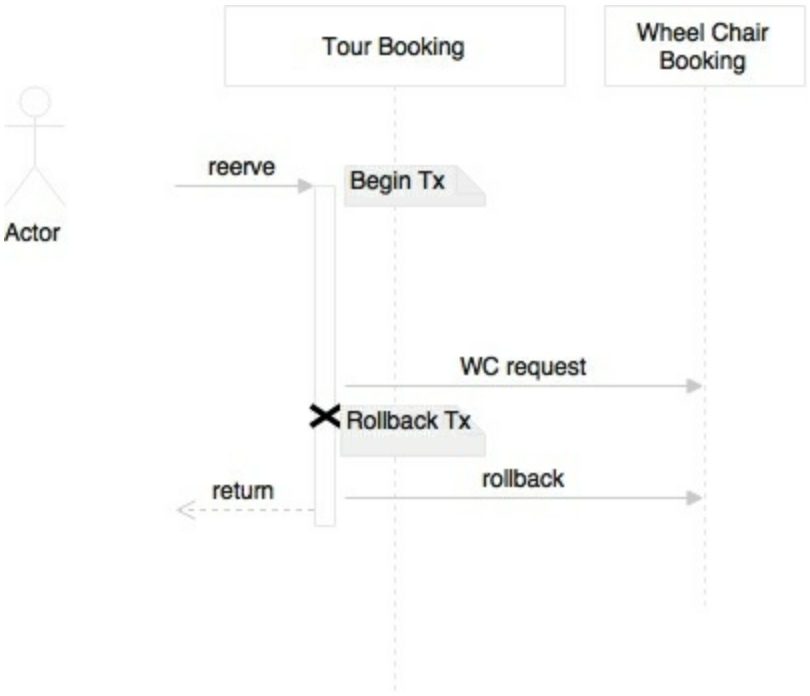
A classical problem is that of the last room selling scenario in a hotel booking use case. What if there is only one room left, and there are multiple customers booking this singe available room? A business model change sometimes makes this scenario less impactful. We could set an "under booking profile", where the actual number of bookable rooms can go below the actual number of rooms (*bookable = available - 3*) in anticipation of some cancellations. Anything in this range will be accepted as "subject to confirmation", and customers will be charged only if payment is confirmed. Bookings will be confirmed in a set time window.

Now consider the scenario where we are creating customer profiles in a NoSQL database like CouchDB. In more traditional approaches with RDBMS, we insert a customer first, and then insert the customer's address, profile details, then preferences, all in one transaction. When using NoSQL, we may not do the same steps. Instead, we may prepare a JSON object with all the details, and insert this into CouchDB in one go. In this second case, no explicit transaction boundaries are required.

Distributed transaction scenarios

The ideal scenario is to use local transactions within a microservice if required, and completely avoid distributed transactions. There could be scenarios where at the end of the execution of one

service, we may want to send a message to another microservice. For example, say a tour reservation has a wheelchair request. Once the reservation is successful, we will have to send a message for the wheelchair booking to another microservice that handles ancillary bookings. The reservation request itself will run on a local transaction. If sending this message fails, we are still in the transaction boundary, and we can roll back the entire transaction. What if we create a reservation and send the message, but after sending the message, we encounter an error in the reservation, the reservation transaction fails, and subsequently, the reservation record is rolled back? Now we end up in a situation where we've unnecessarily created an orphan wheelchair booking:



There are a couple of ways we can address this scenario. The first approach is to delay sending the message till the end. This ensures that there are less chances for any failure after sending the message. Still, if failure occurs after sending the message, then the exception handling routine is run, that is, we send a compensating message to reverse the wheelchair booking.

Service endpoint design consideration

One of the important aspects of microservices is service design. Service design has two key elements: contract design and protocol selection.

Contract design

The first and foremost principle of service design is simplicity. The services should be designed for consumers to consume. A complex service contract reduces the usability of the service. The **KISS** (**Keep It Simple Stupid**) principle helps us to build better quality services faster, and reduces the cost of maintenance and replacement. The **YAGNI** (**You Ain't Gonna Need It**) is another principle supporting this idea. Predicting future requirements and building systems are, in reality, not future-proofed. This results in large upfront investment as well as higher cost of maintenance.

Evolutionary design is a great concept. Do just enough design to satisfy today's wants, and keep changing and refactoring the design to accommodate new features as and when they are required. Having said that, this may not be simple unless there is a strong governance in place.

Consumer Driven Contracts (**CDC**) is a great idea that supports evolutionary design. In many cases, when the service contract gets changed, all consuming applications have to undergo testing. This makes change difficult. CDC helps in building confidence in consumer applications. CDC advocates each consumer to provide their expectation to the provider in the form of test cases so that the provider uses them as integration tests whenever the service contract is changed.

Postel's law is also relevant in this scenario. Postel's law primarily addresses TCP communications; however, this is also equally applicable to service design. When it comes to service design, service providers should be as flexible as possible when accepting consumer requests, whereas service consumers should stick to the contract as agreed with the provider.

Protocol selection

In the SOA world, HTTP/SOAP, and messaging were kinds of default service protocols for service interactions. Microservices follow the same design principles for service interaction. Loose coupling is one of the core principles in the microservices world too.

Microservices fragment applications into many physically independent deployable services. This not only increases the communication cost, it is also susceptible to network failures. This could also result in poor performance of services.

Message-oriented services

If we choose an asynchronous style of communication, the user is disconnected, and therefore, response times are not directly impacted. In such cases, we may use standard JMS or AMQP protocols for communication with JSON as payload. Messaging over HTTP is also popular, as it reduces complexity. Many new entrants in messaging services support HTTP-based communication.

Asynchronous REST is also possible, and is handy when calling long-running services.

HTTP and REST endpoints

Communication over HTTP is always better for interoperability, protocol handling, traffic routing, load balancing, security systems, and the like. Since HTTP is stateless, it is more compatible for handling stateless services with no affinity. Most of the development frameworks, testing tools, runtime containers, security systems, and so on are friendlier towards HTTP.

With the popularity and acceptance of REST and JSON, it is the default choice for microservice developers. The HTTP/REST/JSON protocol stack makes building interoperable systems very easy and friendly. HATEOAS is one of the design patterns emerging for designing progressive rendering and self-service navigations. As discussed in the previous chapter, HATEOAS provides a mechanism to link resources together so that the consumer can navigate between resources. RFC 5988 – Web Linking is another upcoming standard.

Optimized communication protocols

If the service response times are stringent, then we need to pay special attention to the communication aspects. In such cases, we may choose alternate protocols such as Avro, Protocol Buffers, or Thrift for communicating between services. But this limits the interoperability of services. The trade-off is between performance and interoperability requirements. Custom binary protocols need careful evaluation as they bind native objects on both sides—consumer and producer. This could run into release management issues such as class version mismatch in Java-based RPC style communications.

API documentations

Last thing: a good API is not only simple, but should also have enough documentation for the consumers. There are many tools available today for documenting REST-based services like Swagger, RAML, and API Blueprint.

Handling shared libraries

The principle behind microservices is that they should be autonomous and self-contained. In order to adhere to this principle, there may be situations where we will have to duplicate code and libraries. These could be either technical libraries or functional components.



For example, the eligibility for a flight upgrade will be checked at the time of check-in as well as when boarding. If check-in and boarding are two different microservices, we may have to duplicate the eligibility rules in both the services. This was the trade-off between adding a dependency versus code duplication.

It may be easy to embed code as compared to adding an additional dependency, as it enables better release management and performance. But this is against the DRY principle.

Note

DRY principle

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

The downside of this approach is that in case of a bug or an enhancement on the shared library, it has to be upgraded in more than one place. This may not be a severe setback as each service can contain a different version of the shared library:



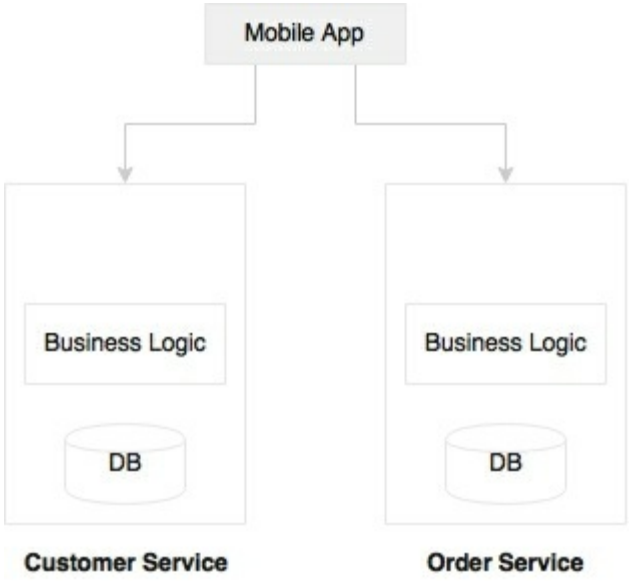
An alternative option of developing the shared library as another microservice itself needs careful analysis. If it is not qualified as a microservice from the business capability point of view, then it may add more complexity than its usefulness. The trade-off analysis is between overheads in communication versus duplicating the libraries in multiple services.

User interfaces in microservices

The microservices principle advocates a microservice as a vertical slice from the database to presentation:



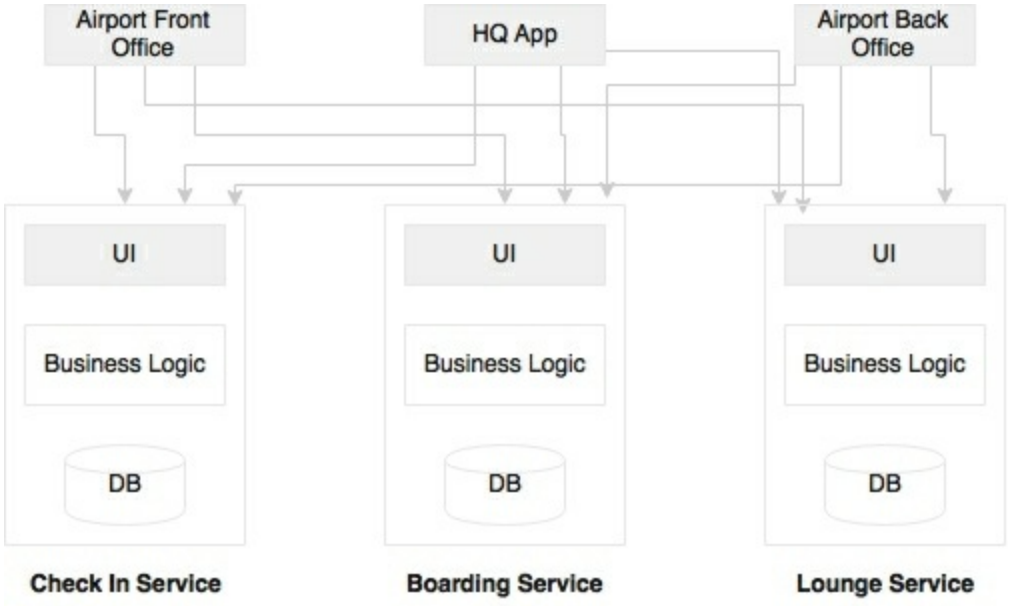
In reality, we get requirements to build quick UI and mobile applications mashing up the existing APIs. This is not uncommon in the modern scenario, where a business wants quick turnaround time from IT:



Penetration of mobile applications is one of the causes of this approach. In many organizations, there will be mobile development teams sitting close to the business team, developing rapid mobile applications by combining and mashing up APIs from multiple sources, both internal and external. In such situations, we may just expose services, and leave it for the mobile teams to realize in the way the business wants. In this case, we will build headless microservices, and leave it to the mobile teams to build a presentation layer.

Another category of problem is that the business may want to build consolidated web applications

targeted to communities:



For example, the business may want to develop a departure control application targeting airport users. A departure control web application may have functions such as check-in, lounge management, boarding, and so on. These may be designed as independent microservices. But from the business standpoint, it all needs to be clubbed into a single web application. In such cases, we will have to build web applications by mashing up services from the backend.

One approach is to build a container web application or a placeholder web application, which links to multiple microservices at the backend. In this case, we develop full stack microservices, but the screens coming out of this could be embedded in to another placeholder web application. One of the advantages of this approach is that you can have multiple placeholder web applications targeting different user communities, as shown in the preceding diagram. We may use an API gateway to avoid those crisscross connections. We will explore the API gateway in the next section.

Use of API gateways in microservices

With the advancement of client-side JavaScript frameworks like AngularJS, the server is expected to expose RESTful services. This could lead to two issues. The first issue is the mismatch in contract expectations. The second issue is multiple calls to the server to render a page.

We start with the contract mismatch case. For example, `GetCustomer` may return a JSON with many fields:

```
Customer {
  Name:
  Address:
  Contact:
}
```

In the preceding case, `Name`, `Address`, and `Contact` are nested JSON objects. But a mobile client may expect only basic information such as first name, and last name. In the SOA world, an ESB or a mobile middleware did this job of transformation of data for the client. The default approach in microservices is to get all the elements of `Customer`, and then the client takes up the responsibility to filter the elements. In this case, the overhead is on the network.

There are several approaches we can think about to solve this case:

```
Customer {
  Id: 1
  Name: /customer/name/1
  Address: /customer/address/1
  Contact: /customer/contact/1
}
```

In the first approach, minimal information is sent with links as explained in the section on HATEOAS. In the preceding case, for customer ID `1`, there are three links, which will help the client to access specific data elements. The example is a simple logical representation, not the actual JSON. The mobile client in this case will get basic customer information. The client further uses the links to get the additional required information.

The second approach is used when the client makes the REST call; it also sends the required fields as part of the query string. In this scenario, the client sends a request with `firstname` and `lastname` as the query string to indicate that the client only requires these two fields. The downside is that it ends up in complex server-side logic as it has to filter based on the fields. The server has to send different elements based on the incoming query.

The third approach is to introduce a level of indirection. In this, a gateway component sits between the client and the server, and transforms data as per the consumer's specification. This is a better approach as we do not compromise on the backend service contract. This leads to what is called UI services. In many cases, the API gateway acts as a proxy to the backend, exposing a set of consumer-specific APIs:



There are two ways we can deploy an API gateway. The first one is one API gateway per microservice as shown in diagram **A**. The second approach (diagram **B**) is to have a common API gateway for multiple services. The choice really depends on what we are looking for. If we are using an API gateway as a reverse proxy, then off-the-shelf gateways such as Apigee, Mashery, and the like could be used as a shared platform. If we need fine-grained control over traffic shaping and complex transformations, then per service custom API gateways may be more useful.

A related problem is that we will have to make many calls from the client to the server. If we refer to our holiday example in Chapter 1, *Demystifying Microservices*, you know that for rendering each widget, we had to make a call to the server. Though we transfer only data, it can still add a significant overhead on the network. This approach is not fully wrong, as in many cases, we use responsive design and progressive design. The data will be loaded on demand, based on user navigations. In order to do this, each widget in the client should make independent calls to the server in a lazy mode. If bandwidth is an issue, then an API gateway is the solution. An API gateway acts as a middleman to compose and transform APIs from multiple microservices.

Use of ESB and iPaaS with microservices

Theoretically, SOA is not all about ESBs, but the reality is that ESBs have always been at the center of many SOA implementations. What would be the role of an ESB in the microservices world?

In general, microservices are fully cloud native systems with smaller footprints. The lightweight characteristics of microservices enable automation of deployments, scaling, and so on. On the contrary, enterprise ESBs are heavyweight in nature, and most of the commercial ESBs are not cloud friendly. The key features of an ESB are protocol mediation, transformation, orchestration, and application adaptors. In a typical microservices ecosystem, we may not need any of these features.

The limited ESB capabilities that are relevant for microservices are already available with more lightweight tools such as an API gateway. Orchestration is moved from the central bus to the microservices themselves. Therefore, there is no centralized orchestration capability expected in the case of microservices. Since the services are set up to accept more universal message exchange styles using REST/JSON calls, no protocol mediation is required. The last piece of capability that we get from ESBs are the adaptors to connect back to the legacy systems. In the case of microservices, the service itself provides a concrete implementation, and hence, there are no legacy connectors required. For these reasons, there is no natural space for ESBs in the microservices world.

Many organizations established ESBs as the backbone for their application integrations (EAI). Enterprise architecture policies in such organizations are built around ESBs. There could be a number of enterprise-level policies such as auditing, logging, security, validation, and so on that would have been in place when integrating using ESB. Microservices, however, advocate a more decentralized governance. ESBs will be an overkill if integrated with microservices.

Not all services are microservices. Enterprises have legacy applications, vendor applications, and so on. Legacy services use ESBs to connect with microservices. ESBs still hold their place for legacy integration and vendor applications to integrate at the enterprise level.

With the advancement of clouds, the capabilities of ESBs are not sufficient to manage integration between clouds, cloud to on-premise, and so on. **Integration Platform as a Service** (**iPaaS**) is evolving as the next generation application integration platform, which further reduces the role of ESBs. In typical deployments, iPaaS invokes API gateways to access microservices.

Service versioning considerations

When we allow services to evolve, one of the important aspect to consider is service versioning. Service versioning should be considered upfront, and not as an afterthought. Versioning helps us to release new services without breaking the existing consumers. Both the old version and the new version will be deployed side by side.

Semantic versions are widely used for service versioning. A semantic version has three components: **major**, **minor**, and **patch**. Major is used when there is a breaking change, minor is used when there is a backward compatible change, and patch is used when there is a backward compatible bug fix.

Versioning could get complicated when there is more than one service in a microservice. It is always simple to version services at the service level compared to the operations level. If there is a change in one of the operations, the service is upgraded and deployed to V2. The version change is applicable to all operations in the service. This is the notion of immutable services.

There are three different ways in which we can version REST services:

- URI versioning
- Media type versioning
- Custom header

In URI versioning, the version number is included in the URL itself. In this case, we just need to be worried about the major versions only. Hence, if there is a minor version change or a patch, the consumers do not need to worry about the changes. It is a good practice to alias the latest version to a non-versioned URI, which is done as follows:

```
/api/v3/customer/1234
/api/customer/1234  - aliased to v3.

@RestController("CustomerControllerV3")
@RequestMapping("api/v3/customer")
public class CustomerController {

}
```

A slightly different approach is to use the version number as part of the URL parameter:

```
api/customer/100?v=1.5
```

In case of media type versioning, the version is set by the client on the HTTP `Accept` header as follows:

```
Accept:  application/vnd.company.customer-v3+json
```

A less effective approach for versioning is to set the version in the custom header:

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET, headers =
```

```
{"version=3"})
public Customer getCustomer(@PathVariable("id") long id) {
     //other code goes here.
}
```

In the URI approach, it is simple for the clients to consume services. But this has some inherent issues such as the fact that versioning-nested URI resources could be complex. Indeed, migrating clients is slightly complex as compared to media type approaches, with caching issues for multiple versions of the services, and others. However, these issues are not significant enough for us to not go with a URI approach. Most of the big Internet players such as Google, Twitter, LinkedIn, and Salesforce are following the URI approach.

Design for cross origin

With microservices, there is no guarantee that the services will run from the same host or same domain. Composite UI web applications may call multiple microservices for accomplishing a task, and these could come from different domains and hosts.

CORS allows browser clients to send requests to services hosted on different domains. This is essential in a microservices-based architecture.

One approach is to enable all microservices to allow cross origin requests from other trusted domains. The second approach is to use an API gateway as a single trusted domain for the clients.

Handling shared reference data

When breaking large applications, one of the common issues which we see is the management of master data or reference data. Reference data is more like shared data required between different microservices. City master, country master, and so on will be used in many services such as flight schedules, reservations, and others.

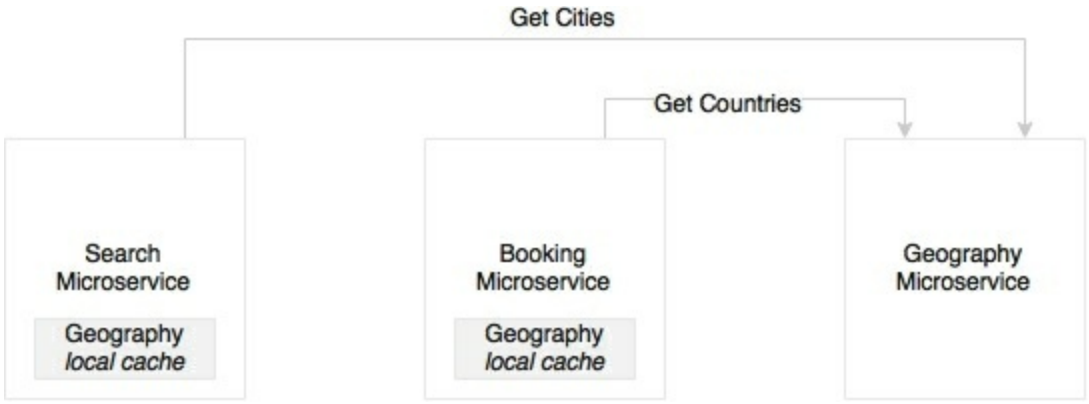
There are a few ways in which we can solve this. For instance, in the case of relatively static, never changing data, then every service can hardcode this data within all the microservices themselves:



Another approach, as shown in the preceding diagram, is to build it as another microservice. This is good, clean, and neat, but the downside is that every service may need to call the master data multiple times. As shown in the diagram for the **Search** and **Booking** example, there are transactional microservices, which use the **Geography** microservice to access shared data:



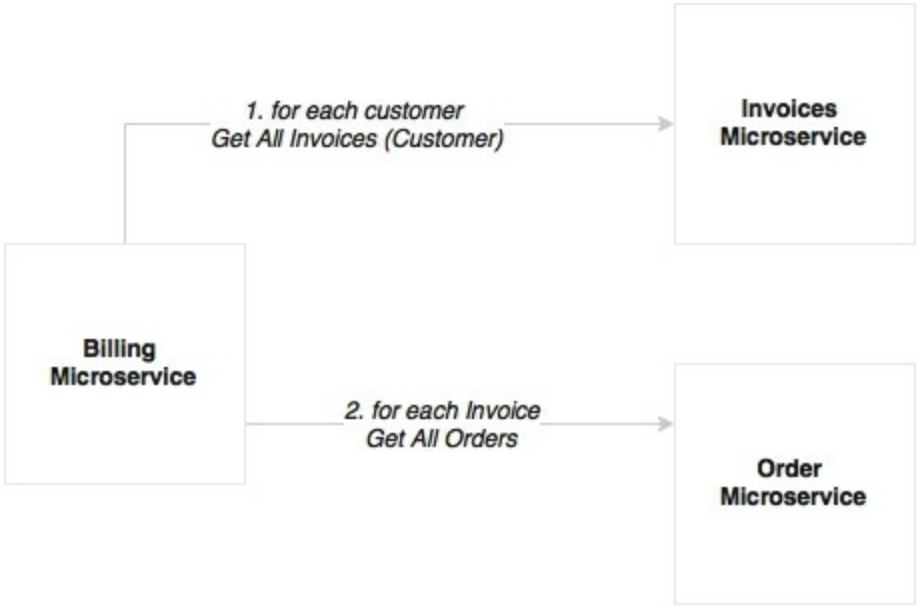
Another option is to replicate the data with every microservice. There is no single owner, but each service has its required master data. When there is an update, all the services are updated. This is extremely performance friendly, but one has to duplicate the code in all the services. It is also complex to keep data in sync across all microservices. This approach makes sense if the code base and data is simple or the data is more static.



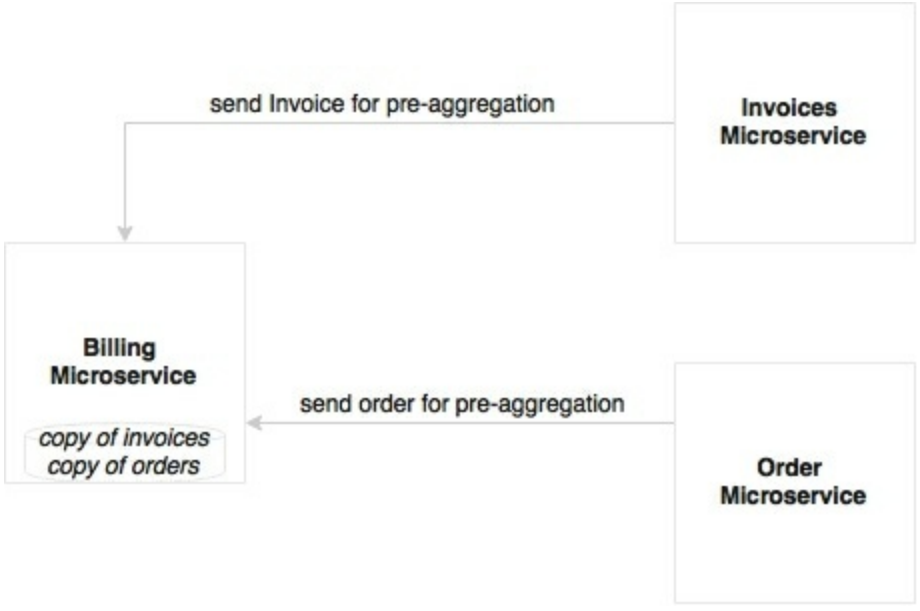
Yet another approach is similar to the first approach, but each service has a local near cache of the required data, which will be loaded incrementally. A local embedded cache such as Ehcache or data grids like Hazelcast or Infinispan could also be used based on the data volumes. This is the most preferred approach for a large number of microservices that have dependency on the master data.

Microservices and bulk operations

Since we have broken monolithic applications into smaller, focused services, it is no longer possible to use join queries across microservice data stores. This could lead to situations where one service may need many records from other services to perform its function.



For example, a monthly billing function needs the invoices of many customers to process the billing. To make it a bit more complicated, invoices may have many orders. When we break billing, invoices, and orders into three different microservices, the challenge that arises is that the **Billing** service has to query the **Invoices** service for each customer to get all the invoices, and then for each invoice, call the **Order** service for getting the orders. This is not a good solution, as the number of calls that goes to other microservices are high:



There are two ways we can think about for solving this. The first approach is to pre-aggregate data as and when it is created. When an order is created, an event is sent out. Upon receiving the event, the **Billing** microservice keeps aggregating data internally for monthly processing. In this case, there is no need for the **Billing** microservice to go out for processing. The downside of this approach is that there is duplication of data.

A second approach, when pre-aggregation is not possible, is to use batch APIs. In such cases, we call `GetAllInvoices`, then we use multiple batches, and each batch further uses parallel threads to get orders. Spring Batch is useful in these situations.

Microservices challenges

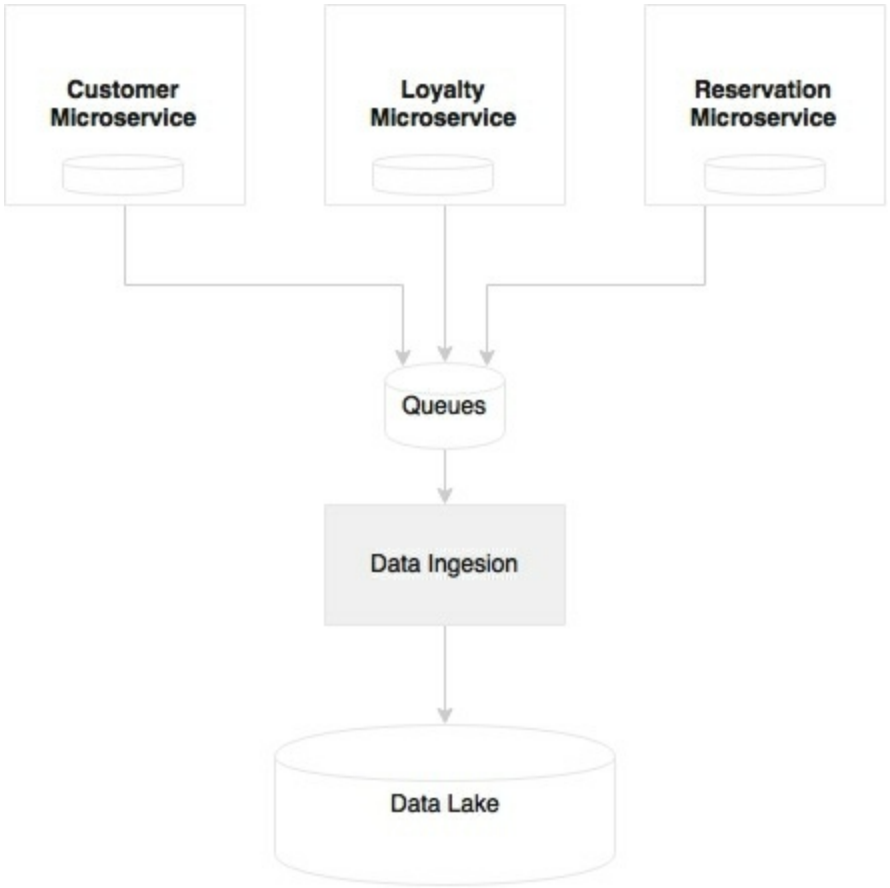
In the previous section, you learned about the right design decisions to be taken, and the trade-offs to be applied. In this section, we will review some of the challenges with microservices, and how to address them for a successful microservice development.

Data islands

Microservices abstract their own local transactional store, which is used for their own transactional purposes. The type of store and the data structure will be optimized for the services offered by the microservice.

For example, if we want to develop a customer relationship graph, we may use a graph database like Neo4j, OrientDB, and the like. A predictive text search to find out a customer based on any related information such as passport number, address, e-mail, phone, and so on could be best realized using an indexed search database like Elasticsearch or Solr.

This will place us into a unique situation of fragmenting data into heterogeneous data islands. For example, Customer, Loyalty Points, Reservations, and others are different microservices, and hence, use different databases. What if we want to do a near real-time analysis of all high value customers by combining data from all three data stores? This was easy with a monolithic application, because all the data was present in a single database:



In order to satisfy this requirement, a data warehouse or a data lake is required. Traditional data warehouses like Oracle, Teradata, and others are used primarily for batch reporting. But with NoSQL databases (like Hadoop) and microbatching techniques, near real-time analytics is possible with the concept of data lakes. Unlike the traditional warehouses that are purpose-built for batch reporting, data lakes store raw data without assuming how the data is going to be used. Now the question really

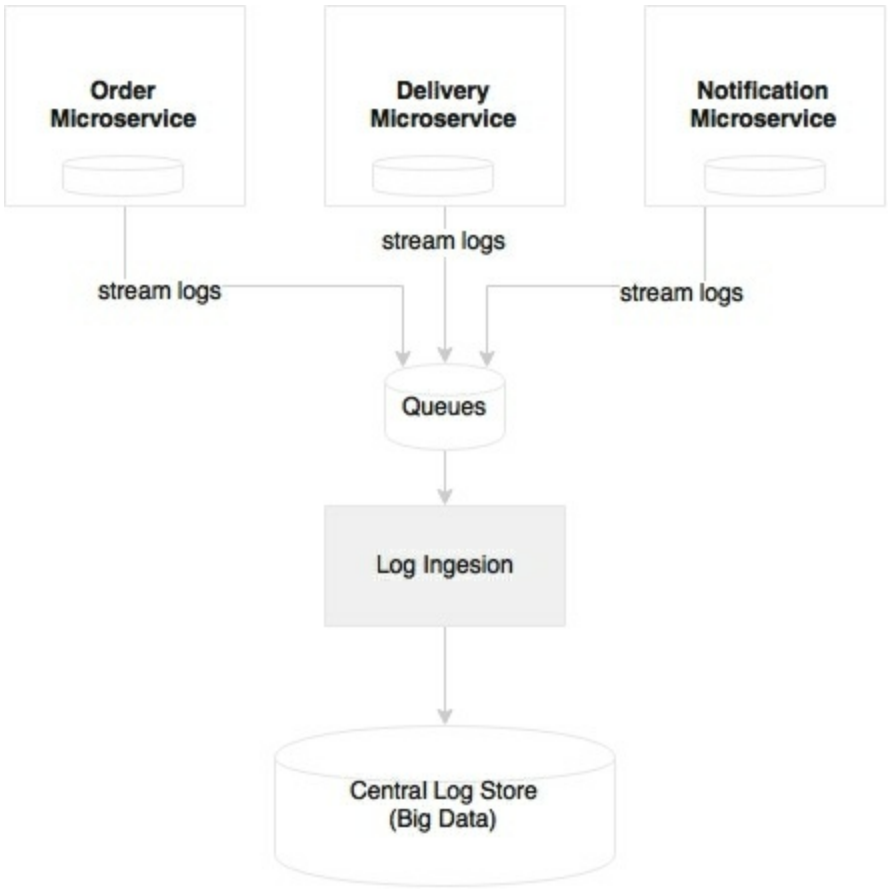
is how to port the data from microservices into data lakes.

Data porting from microservices to a data lake or a data warehouse can be done in many ways. Traditional ETL could be one of the options. Since we allow backdoor entry with ETL, and break the abstraction, this is not considered an effective way for data movement. A better approach is to send events from microservices as and when they occur, for example, customer registration, customer update events, and so on. Data ingestion tools consume these events, and propagate the state change to the data lake appropriately. The data ingestion tools are highly scalable platforms such as Spring Cloud Data Flow, Kafka, Flume, and so on.

Logging and monitoring

Log files are a good piece of information for analysis and debugging. Since each microservice is deployed independently, they emit separate logs, maybe to a local disk. This results in fragmented logs. When we scale services across multiple machines, each service instance could produce separate log files. This makes it extremely difficult to debug and understand the behavior of the services through log mining.

Examining **Order**, **Delivery**, and **Notification** as three different microservices, we find no way to correlate a customer transaction that runs across all three of them:



When implementing microservices, we need a capability to ship logs from each service to a centrally managed log repository. With this approach, services do not have to rely on the local disk or local I/Os. A second advantage is that the log files are centrally managed, and are available for all sorts of analysis such as historical, real time, and trending. By introducing a correlation ID, end-to-end transactions can be easily tracked.

With a large number of microservices, and with multiple versions and service instances, it would be difficult to find out which service is running on which server, what's the health of these services, the service dependencies, and so on. This was much easier with monolithic applications that are tagged against a specific or a fixed set of servers.

Apart from understanding the deployment topology and health, it also poses a challenge in identifying service behaviors, debugging, and identifying hotspots. Strong monitoring capabilities are required to manage such an infrastructure.

We will cover the logging and monitoring aspects in Chapter 7, *Logging and Monitoring Microservices*.

Dependency management

Dependency management is one of the key issues in large microservice deployments. How do we identify and reduce the impact of a change? How do we know whether all the dependent services are up and running? How will the service behave if one of the dependent services is not available?

Too many dependencies could raise challenges in microservices. Four important design aspects are stated as follows:

- Reducing dependencies by properly designing service boundaries.
- Reducing impacts by designing dependencies as loosely coupled as possible. Also, designing service interactions through asynchronous communication styles.
- Tackling dependency issues using patterns such as circuit breakers.
- Monitoring dependencies using visual dependency graphs.

Organization culture

One of the biggest challenges in microservices implementation is the organization culture. To harness the speed of delivery of microservices, the organization should adopt Agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning.

Organizations following a waterfall development or heavyweight release management processes with infrequent release cycles are a challenge for microservices development. Insufficient automation is also a challenge for microservices deployment.

In short, Cloud and DevOps are supporting facets of microservice development. These are essential for successful microservices implementation.

Governance challenges

Microservices impose decentralized governance, and this is quite in contrast with the traditional SOA governance. Organizations may find it hard to come up with this change, and that could negatively impact the microservices development.

There are number of challenges that comes with a decentralized governance model. How do we understand who is consuming a service? How do we ensure service reuse? How do we define which services are available in the organization? How do we ensure enforcement of enterprise polices?

The first thing is to have a set of standards, best practices, and guidelines on how to implement better services. These should be available to the organization in the form of standard libraries, tools, and techniques. This ensures that the services developed are top quality, and that they are developed in a consistent manner.

The second important consideration is to have a place where all stakeholders can not only see all the services, but also their documentations, contracts, and service-level agreements. Swagger and API Blueprint are commonly used for handling these requirements.

Operation overheads

Microservices deployment generally increases the number of deployable units and virtual machines (or containers). This adds significant management overheads and increases the cost of operations.

With a single application, a dedicated number of containers or virtual machines in an on-premise data center may not make much sense unless the business benefit is very high. The cost generally goes down with economies of scale. A large number of microservices that are deployed in a shared infrastructure which is fully automated makes more sense, since these microservices are not tagged against any specific VMs or containers. Capabilities around infrastructure automation, provisioning, containerized deployment, and so on are essential for large scale microservices deployments. Without this automation, it would result in a significant operation overhead and increased cost.

With many microservices, the number of **configurable items** (**CIs**) becomes too high, and the number of servers in which these CIs are deployed might also be unpredictable. This makes it extremely difficult to manage data in a traditional **Configuration Management Database** (**CMDB**). In many cases, it is more useful to dynamically discover the current running topology than a statically configured CMDB-style deployment topology.

Testing microservices

Microservices also pose a challenge for the testability of services. In order to achieve a full-service functionality, one service may rely on another service, and that, in turn, on another service—either synchronously or asynchronously. The issue is how do we test an end-to-end service to evaluate its behavior? The dependent services may or may not be available at the time of testing.

Service virtualization or service mocking is one of the techniques used for testing services without actual dependencies. In testing environments, when the services are not available, mock services can simulate the behavior of the actual service. The microservices ecosystem needs service virtualization capabilities. However, this may not give full confidence, as there may by many corner cases that mock services do not simulate, especially when there are deep dependencies.

Another approach, as discussed earlier, is to use a consumer driven contract. The translated integration test cases can cover more or less all corner cases of the service invocation.

Test automation, appropriate performance testing, and continuous delivery approaches such as A/B testing, future flags, canary testing, blue-green deployments, and red-black deployments, all reduce the risks of production releases.

Infrastructure provisioning

As briefly touched on under operation overheads, manual deployment could severely challenge the microservices rollouts. If a deployment has manual elements, the deployer or operational administrators should know the running topology, manually reroute traffic, and then deploy the application one by one till all services are upgraded. With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

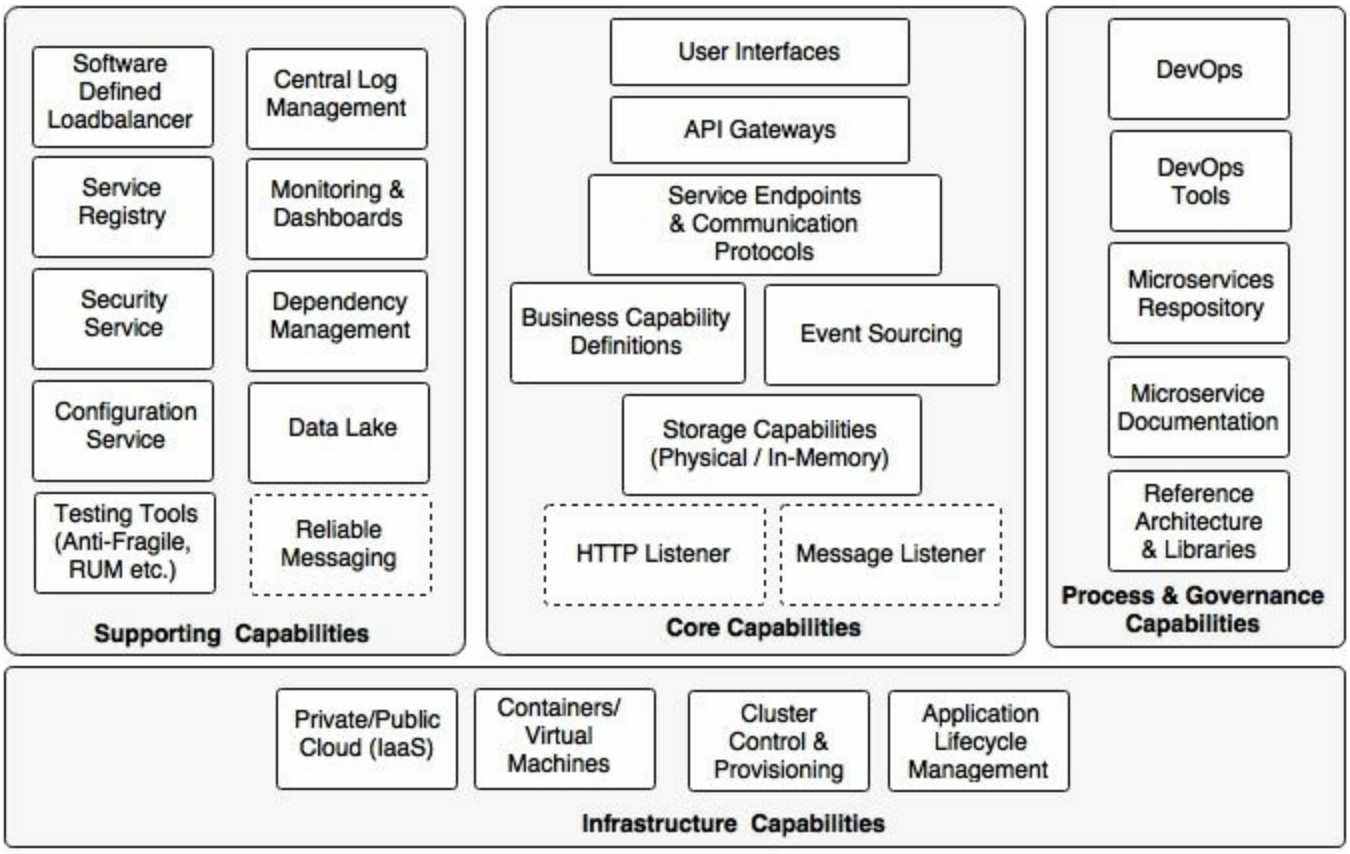
Microservices require a supporting elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

In a large deployment environment with many microservices, we may also need additional tools to manage VMs or containers that can further initiate or destroy services automatically.

The microservices capability model

Before we conclude this chapter, we will review a capability model for microservices based on the design guidelines and common pattern and solutions described in this chapter.

The following diagram depicts the microservices capability model:



The capability model is broadly classified in to four areas:

- **Core capabilities**: These are part of the microservices themselves
- **Supporting capabilities**: These are software solutions supporting core microservice implementations
- **Infrastructure capabilities**: These are infrastructure level expectations for a successful microservices implementation
- **Governance capabilities**: These are more of process, people, and reference information

Core capabilities

The core capabilities are explained as follows:

- **Service listeners** (HTTP/messaging): If microservices are enabled for a HTTP-based service endpoint, then the HTTP listener is embedded within the microservices, thereby eliminating the need to have any external application server requirement. The HTTP listener is started at the time of the application startup. If the microservice is based on asynchronous communication, then instead of an HTTP listener, a message listener is started. Optionally, other protocols could also be considered. There may not be any listeners if the microservice is a scheduled service. Spring Boot and Spring Cloud Streams provide this capability.
- **Storage capability**: The microservices have some kind of storage mechanisms to store state or transactional data pertaining to the business capability. This is optional, depending on the capabilities that are implemented. The storage could be either a physical storage (RDBMS such as MySQL; NoSQL such as Hadoop, Cassandra, Neo 4J, Elasticsearch, and so on), or it could be an in-memory store (cache like Ehcache, data grids like Hazelcast, Infinispan, and so on)
- **Business capability definition**: This is the core of microservices, where the business logic is implemented. This could be implemented in any applicable language such as Java, Scala, Conjure, Erlang, and so on. All required business logic to fulfill the function will be embedded within the microservices themselves.
- **Event sourcing**: Microservices send out state changes to the external world without really worrying about the targeted consumers of these events. These events could be consumed by other microservices, audit services, replication services, or external applications, and the like. This allows other microservices and applications to respond to state changes.
- **Service endpoints and communication protocols**: These define the APIs for external consumers to consume. These could be synchronous endpoints or asynchronous endpoints. Synchronous endpoints could be based on REST/JSON or any other protocols such as Avro, Thrift, Protocol Buffers, and so on. Asynchronous endpoints are through Spring Cloud Streams backed by RabbitMQ, other messaging servers, or other messaging style implementations such as ZeroMQ.
- **API gateway**: The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints. The API gateway is also useful for policy enforcements. It may also provide real time load balancing capabilities. There are many API gateways available in the market. Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.
- **User interfaces**: Generally, user interfaces are also part of microservices for users to interact with the business capabilities realized by the microservices. These could be implemented in any technology, and are channel- and device-agnostic.

Infrastructure capabilities

Certain infrastructure capabilities are required for a successful deployment, and managing large scale microservices. When deploying microservices at scale, not having proper infrastructure capabilities can be challenging, and can lead to failures:

- **Cloud**: Microservices implementation is difficult in a traditional data center environment with long lead times to provision infrastructures. Even a large number of infrastructures dedicated per microservice may not be very cost effective. Managing them internally in a data center may increase the cost of ownership and cost of operations. A cloud-like infrastructure is better for microservices deployment.
- **Containers or virtual machines**: Managing large physical machines is not cost effective, and they are also hard to manage. With physical machines, it is also hard to handle automatic fault tolerance. Virtualization is adopted by many organizations because of its ability to provide optimal use of physical resources. It also provides resource isolation. It also reduces the overheads in managing large physical infrastructure components. Containers are the next generation of virtual machines. VMWare, Citrix, and so on provide virtual machine technologies. Docker, Drawbridge, Rocket, and LXD are some of the containerizer technologies.
- **Cluster control and provisioning**: Once we have a large number of containers or virtual machines, it is hard to manage and maintain them automatically. Cluster control tools provide a uniform operating environment on top of the containers, and share the available capacity across multiple services. Apache Mesos and Kubernetes are examples of cluster control systems.
- **Application lifecycle management**: Application lifecycle management tools help to invoke applications when a new container is launched, or kill the application when the container shuts down. Application life cycle management allows for script application deployments and releases. It automatically detects failure scenario, and responds to those failures thereby ensuring the availability of the application. This works in conjunction with the cluster control software. Marathon partially addresses this capability.

Supporting capabilities

Supporting capabilities are not directly linked to microservices, but they are essential for large scale microservices development:

- **Software defined load balancer**: The load balancer should be smart enough to understand the changes in the deployment topology, and respond accordingly. This moves away from the traditional approach of configuring static IP addresses, domain aliases, or cluster addresses in the load balancer. When new servers are added to the environment, it should automatically detect this, and include them in the logical cluster by avoiding any manual interactions. Similarly, if a service instance is unavailable, it should take it out from the load balancer. A combination of Ribbon, Eureka, and Zuul provide this capability in Spring Cloud Netflix.
- **Central log management**: As explored earlier in this chapter, a capability is required to centralize all logs emitted by service instances with the correlation IDs. This helps in debugging, identifying performance bottlenecks, and predictive analysis. The result of this is fed back into the life cycle manager to take corrective actions.
- **Service registry**: A service registry provides a runtime environment for services to automatically publish their availability at runtime. A registry will be a good source of information to understand the services topology at any point. Eureka from Spring Cloud, Zookeeper, and Etcd are some of the service registry tools available.
- **Security service**: A distributed microservices ecosystem requires a central server for managing service security. This includes service authentication and token services. OAuth2-based services are widely used for microservices security. Spring Security and Spring Security OAuth are good candidates for building this capability.
- **Service configuration**: All service configurations should be externalized as discussed in the Twelve-Factor application principles. A central service for all configurations is a good choice. Spring Cloud Config server, and Archaius are out-of-the-box configuration servers.
- **Testing tools (anti-fragile, RUM, and so on)**: Netflix uses Simian Army for anti-fragile testing. Matured services need consistent challenges to see the reliability of the services, and how good fallback mechanisms are. Simian Army components create various error scenarios to explore the behavior of the system under failure scenarios.
- **Monitoring and dashboards**: Microservices also require a strong monitoring mechanism. This is not just at the infrastructure-level monitoring but also at the service level. Spring Cloud Netflix Turbine, Hysterix Dashboard, and the like provide service level information. End-to-end monitoring tools like AppDynamic, New Relic, Dynatrace, and other tools like statd, Sensu, and Spigo could add value to microservices monitoring.
- **Dependency and CI management**: We also need tools to discover runtime topologies, service dependencies, and to manage configurable items. A graph-based CMDB is the most obvious tool to manage these scenarios.
- **Data lake**: As discussed earlier in this chapter, we need a mechanism to combine data stored in different microservices, and perform near real-time analytics. A data lake is a good choice for achieving this. Data ingestion tools like Spring Cloud Data Flow, Flume, and Kafka are used to consume data. HDFS, Cassandra, and the like are used for storing data.

- **Reliable messaging**: If the communication is asynchronous, we may need a reliable messaging infrastructure service such as RabbitMQ or any other reliable messaging service. Cloud messaging or messaging as a service is a popular choice in Internet scale message-based service endpoints.

Process and governance capabilities

The last piece in the puzzle is the process and governance capabilities that are required for microservices:

- **DevOps**: The key to successful implementation of microservices is to adopt DevOps. DevOps compliment microservices development by supporting Agile development, high velocity delivery, automation, and better change management.
- **DevOps tools**: DevOps tools for Agile development, continuous integration, continuous delivery, and continuous deployment are essential for successful delivery of microservices. A lot of emphasis is required on automated functioning, real user testing, synthetic testing, integration, release, and performance testing.
- **Microservices repository**: A microservices repository is where the versioned binaries of microservices are placed. These could be a simple Nexus repository or a container repository such as a Docker registry.
- **Microservice documentation**: It is important to have all microservices properly documented. Swagger or API Blueprint are helpful in achieving good microservices documentation.
- **Reference architecture and libraries**: The reference architecture provides a blueprint at the organization level to ensure that the services are developed according to certain standards and guidelines in a consistent manner. Many of these could then be translated to a number of reusable libraries that enforce service development philosophies.

Summary

In this chapter, you learned about handling practical scenarios that will arise in microservices development.

You learned various solution options and patterns that could be applied to solve common microservices problems. We reviewed a number of challenges when developing large scale microservices, and how to address those challenges effectively.

We also built a capability reference model for a microservices-based ecosystem. The capability model helps in addressing gaps when building Internet scale microservices. The capability model learned in this chapter will be the backbone for this book. The remaining chapters will deep dive into the capability model.

In the next chapter, we will take a real-world problem and model it using the microservices architecture to see how to translate our learnings into practice.

Chapter 4. Microservices Evolution – A Case Study

Like SOA, a microservices architecture can be interpreted differently by different organizations, based on the problem in hand. Unless a sizable, real world problem is examined in detail, microservices concepts are hard to understand.

This chapter will introduce BrownField Airline (BF), a fictitious budget airline, and their journey from a monolithic **Passenger Sales and Service** (**PSS**) application to a next generation microservices architecture. This chapter examines the PSS application in detail, and explains the challenges, approach, and transformation steps of a monolithic system to a microservices-based architecture, adhering to the principles and practices that were explained in the previous chapter.

The intention of this case study is to get us as close as possible to a live scenario so that the architecture concepts can be set in stone.

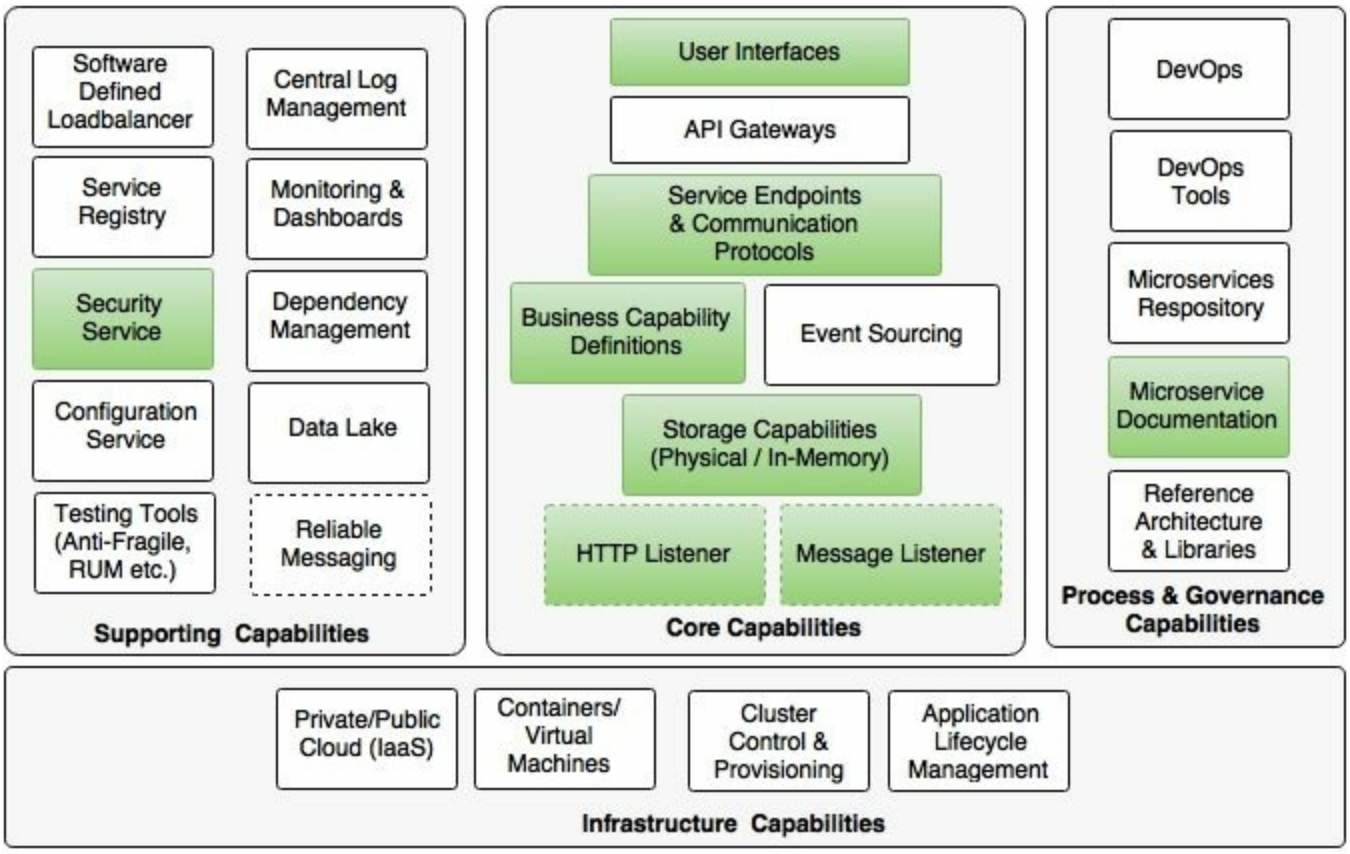
By the end of this chapter, you will have learned about the following:

- A real world case for migrating monolithic systems to microservices-based ones, with the BrownField Airline's PSS application as an example
- Various approaches and transition strategies for migrating a monolithic application to microservices
- Designing a new futuristic microservices system to replace the PSS application using Spring Framework components

Reviewing the microservices capability model

The examples in this chapter explore the following microservices capabilities from the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*:

- **HTTP Listener**
- **Message Listener**
- **Storage Capabilities (Physical/In-Memory)**
- **Business Capability Definitions**
- **Service Endpoints & Communication Protocols**
- **User Interfaces**
- **Security Service**
- **Microservice Documentation**



In Chapter 2, *Building Microservices with Spring Boot*, we explored all these capabilities in isolation including how to secure Spring Boot microservices. This chapter will build a comprehensive microservices example based on a real world case study.

Tip

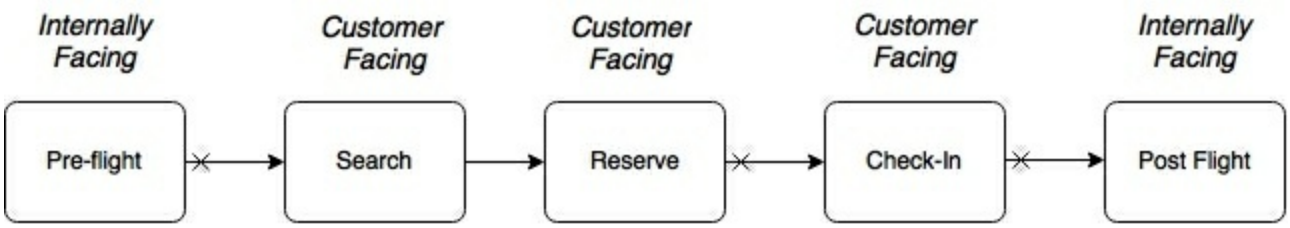
The full source code of this chapter is available under the `Chapter 4` projects in the code files.

Understanding the PSS application

BrownField Airline is one of the fastest growing low-cost, regional airlines, flying directly to more than 100 destinations from its hub. As a start-up airline, BrownField Airline started its operations with few destinations and few aircrafts. BrownField developed its home-grown PSS application to handle their passenger sales and services.

Business process view

This use case is considerably simplified for discussion purposes. The process view in the following diagram shows BrownField Airline's end-to-end passenger services operations covered by the current PSS solution:



The current solution is automating certain customer-facing functions as well as certain internally facing functions. There are two internally facing functions, **Pre-flight** and **Post-flight**. **Pre-flight** functions include the planning phase, used for preparing flight schedules, plans, aircrafts, and so on. **Post-flight** functions are used by the back office for revenue management, accounting, and so on. The **Search** and **Reserve** functions are part of the online seat reservation process, and the **Check-in** function is the process of accepting passengers at the airport. The **Check-in** function is also accessible to the end users over the Internet for online check-in.

The cross marks at the beginning of the arrows in the preceding diagram indicate that they are disconnected, and occur at different timelines. For example, passengers are allowed to book 360 days in advance, whereas the check-in generally happens 24 hours before flight departure.

Functional view

The following diagram shows the functional building blocks of BrownField Airline's PSS landscape. Each business process and its related subfunctions are represented in a row:

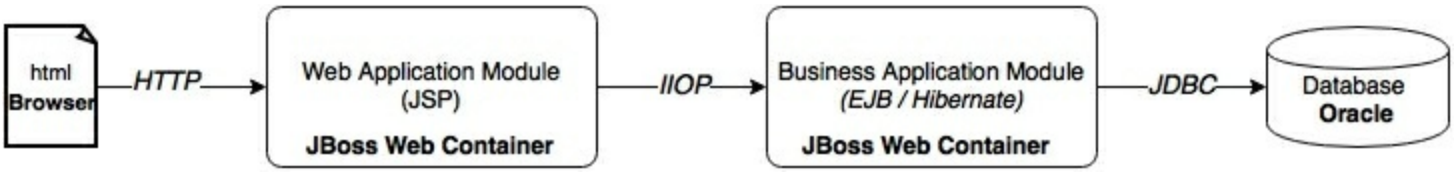


Each subfunction shown in the preceding diagram explains its role in the overall business process. Some subfunctions participate in more than one business process. For example, inventory is used in both search as well as in booking. To avoid any complication, this is not shown in the diagram. Data management and cross-cutting subfunctions are used across many business functions.

Architectural view

In order to effectively manage the end-to-end passenger operations, BrownField had developed an in-house PSS application, almost ten years back. This well-architected application was developed using Java and JEE technologies combined with the best-of-the-breed open source technologies available at the time.

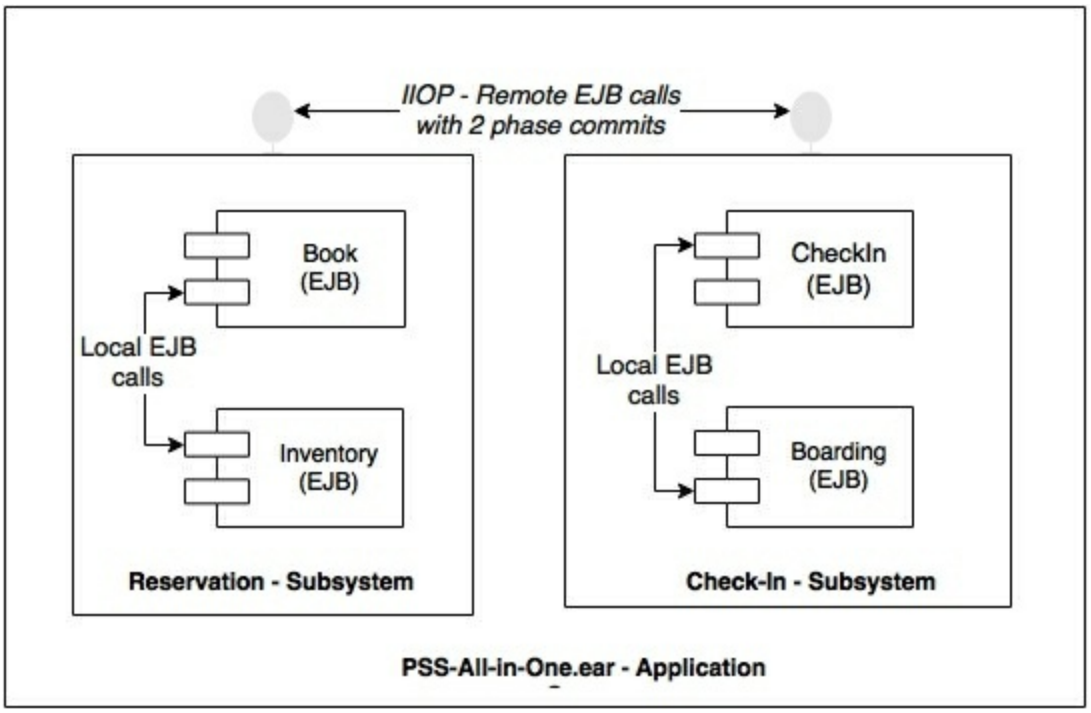
The overall architecture and technologies are shown in the following diagram:



The architecture has well-defined boundaries. Also, different concerns are separated into different layers. The web application was developed as an *N*-tier, component-based modular system. The functions interact with each other through well-defined service contracts defined in the form of EJB endpoints.

Design view

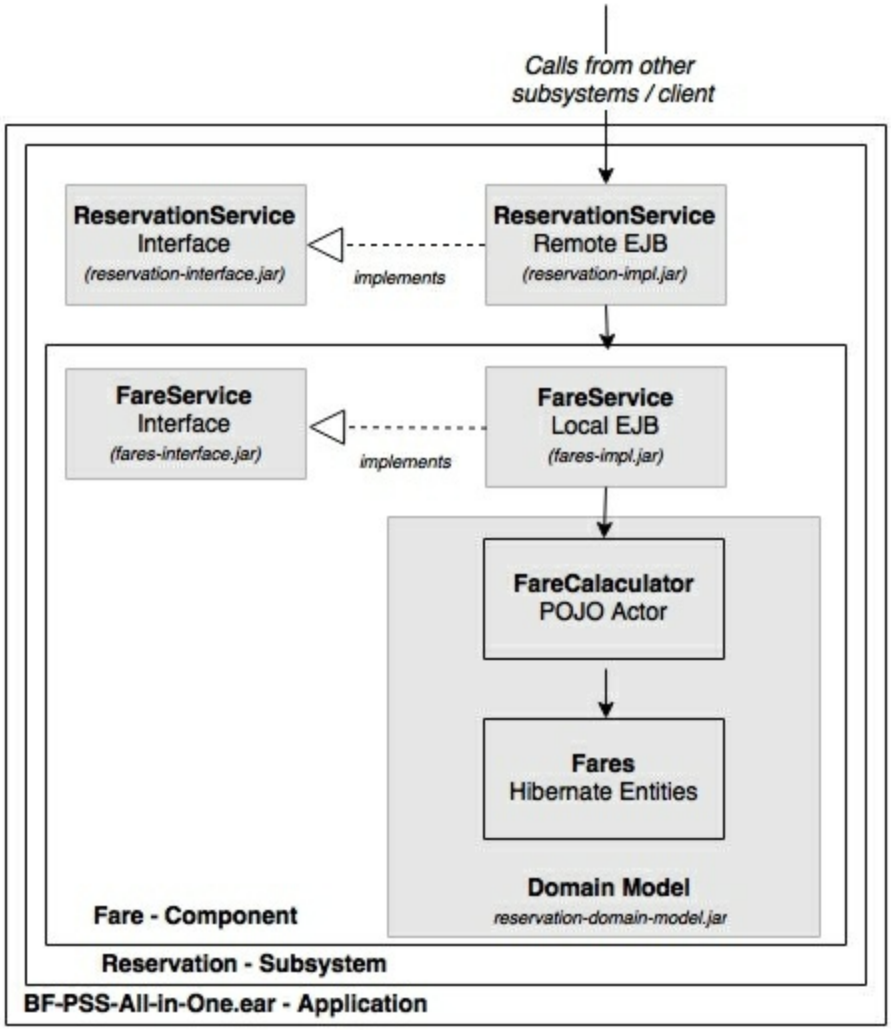
The application has many logical functional groupings or subsystems. Further, each subsystem has many components organized as depicted in the next diagram:



Subsystems interact with each other through remote EJB calls using the IIOP protocol. The transactional boundaries span across subsystems. Components within the subsystems communicate with each other through local EJB component interfaces. In theory, since subsystems use remote EJB endpoints, they could run on different physically separated application servers. This was one of the design goals.

Implementation view

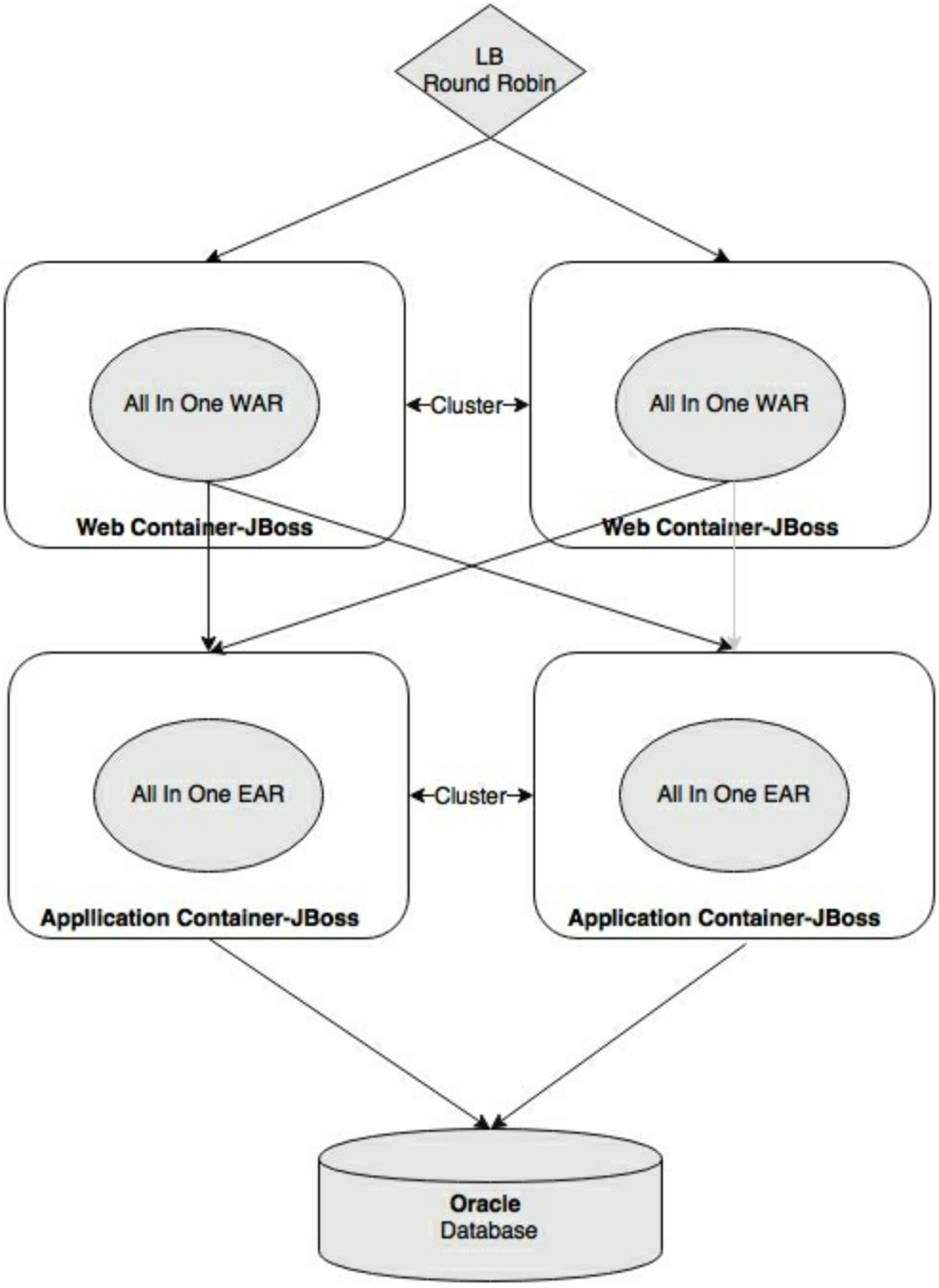
The implementation view in the following diagram showcases the internal organization of a subsystem and its components. The purpose of the diagram is also to show the different types of artifacts:



In the preceding diagram, the gray-shaded boxes are treated as different Maven projects, and translate into physical artifacts. Subsystems and components are designed adhering to the *program to an interface* principle. Interfaces are packaged as separate JAR files so that clients are abstracted from the implementations. The complexity of the business logic is buried in the domain model. Local EJBs are used as component interfaces. Finally, all subsystems are packaged into a single all-in-one EAR, and deployed in the application server.

Deployment view

The application's initial deployment was simple and straightforward as shown in the next diagram:



The web modules and business modules were deployed into separate application server clusters. The application was scaled horizontally by adding more and more application servers to the cluster.

Zero downtime deployments were handled by creating a standby cluster, and gracefully diverting the traffic to that cluster. The standby cluster is destroyed once the primary cluster is patched with the new version and brought back to service. Most of the database changes were designed for backward compatibility, but breaking changes were promoted with application outages.

Death of the monolith

The PSS application was performing well, successfully supporting all business requirements as well as the expected service levels. The system had no issues in scaling with the organic growth of the business in the initial years.

The business has seen tremendous growth over a period of time. The fleet size increased significantly, and new destinations got added to the network. As a result of this rapid growth, the number of bookings has gone up, resulting in a steep increase in transaction volumes, up to 200 - to 500 - fold of what was originally estimated.

Pain points

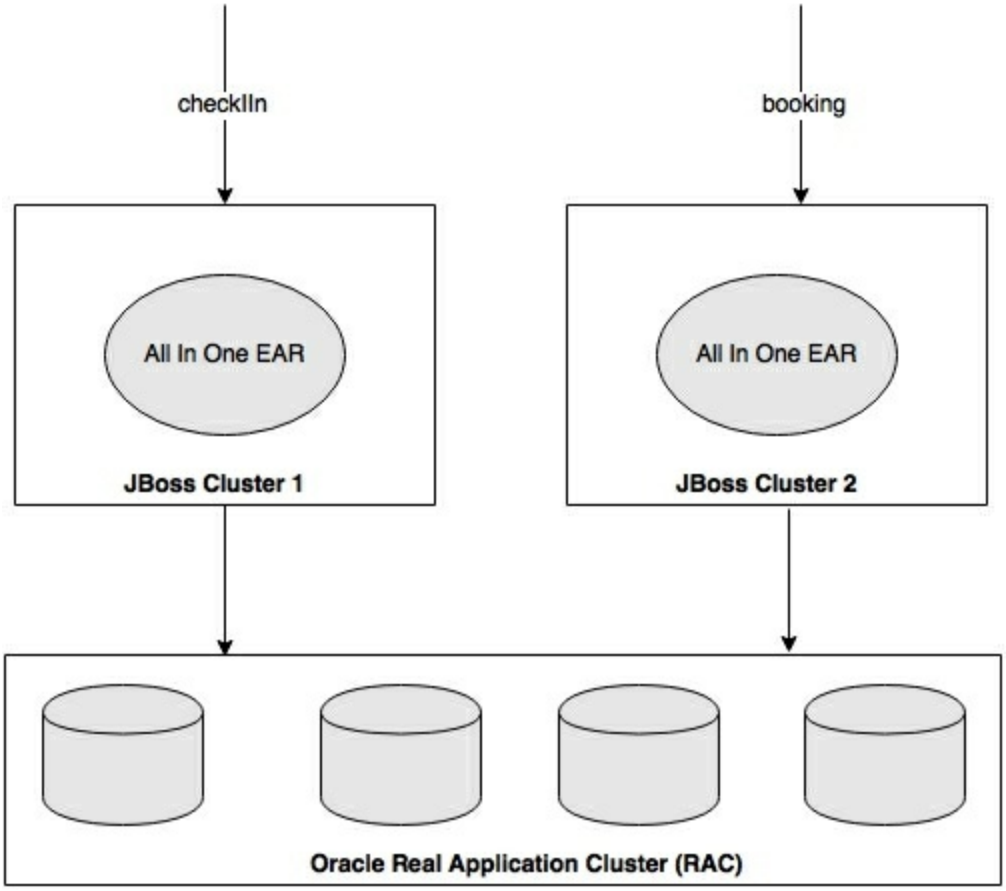
The rapid growth of the business eventually put the application under pressure. Odd stability issues and performance issues surfaced. New application releases started breaking the working code. Moreover, the cost of change and the speed of delivery started impacting the business operations profoundly.

An end-to-end architecture review was ordered, and it exposed the weaknesses of the system as well as the root causes of many failures, which were as follows:

- **Stability**: The stability issues are primarily due to stuck threads, which limit the application server's capability to accept more transactions. The stuck threads are mainly due to database table locks. Memory issues are another contributor to the stability issues. There were also issues in certain resource intensive operations that were impacting the whole application.
- **Outages**: The outage window increased largely because of the increase in server startup time. The root cause of this issue boiled down to the large size of the EAR. Message pile up during any outage windows causes heavy usage of the application immediately after an outage window. Since everything is packaged in a single EAR, any small application code change resulted in full redeployment. The complexity of the zero downtime deployment model described earlier, together with the server startup times increased both the number of outages and their duration.
- **Agility**: The complexity of the code also increased considerably over time, partially due to the lack of discipline in implementing the changes. As a result, changes became harder to implement. Also, the impact analysis became too complex to perform. As a result, inaccurate impact analysis often led to fixes that broke the working code. The application build time went up severely, from a few minutes to hours, causing unacceptable drops in development productivity. The increase in build time also led to difficulty in build automation, and eventually stopped **continuous integration** (**CI**) and unit testing.

Stop gap fix

Performance issues were partially addressed by applying the Y-axis scale method in the scale cube, as described in Chapter 1, *Demystifying Microservices*. The all-encompassing EAR is deployed into multiple disjoint clusters. A software proxy was installed to selectively route the traffic to designated clusters as shown in the following diagram:



This helped BrownField's IT to scale the application servers. Therefore, the stability issues were controlled. However, this soon resulted in a bottleneck at the database level. Oracle's **Real Application Cluster** (**RAC**) was implemented as a solution to this problem at the database layer.

This new scaling model reduced the stability issues, but at a premium of increased complexity and cost of ownership. The technology debt also increased over a period of time, leading to a state where a complete rewrite was the only option for reducing this technology debt.

Retrospection

Although the application was well-architected, there was a clear segregation between the functional components. They were loosely coupled, programmed to interfaces, with access through standards-based interfaces, and had a rich domain model.

The obvious question is, how come such a well-architected application failed to live up to the expectations? What else could the architects have done?

It is important to understand what went wrong over a period of time. In the context of this book, it is also important to understand how microservices can avoid the recurrence of these scenarios. We will examine some of these scenarios in the subsequent sections.

Shared data

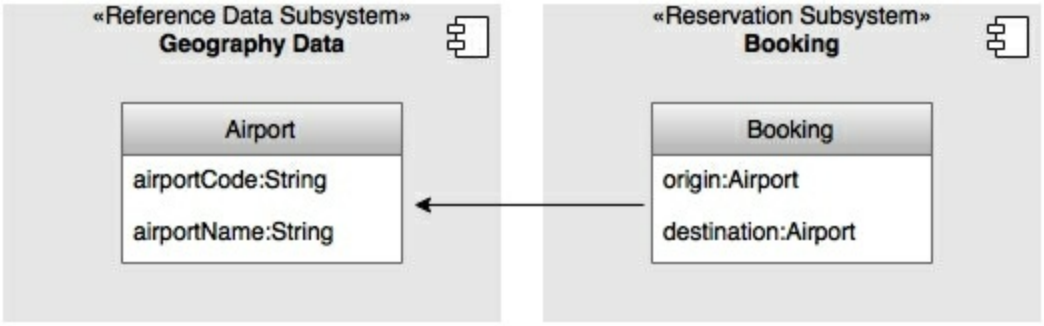
Almost all functional modules require reference data such as the airline's details, airplane details, a list of airports and cities, countries, currencies, and so on. For example, fare is calculated based on the point of origin (city), a flight is between an origin and a destination (airports), check-in is at the origin airport (airport), and so on. In some functions, the reference data is a part of the information model, whereas in some other functions, it is used for validation purposes.

Much of this reference data is neither fully static nor fully dynamic. Addition of a country, city, airport, or the like could happen when the airline introduces new routes. Aircraft reference data could change when the airline purchases a new aircraft, or changes an existing airplane's seat configuration.

One of the common usage scenarios of reference data is to filter the operational data based on certain reference data. For instance, say a user wishes to see all the flights to a country. In this case, the flow of events could be as follows: find all the cities in the selected country, then all airports in the cities, and then fire a request to get all the flights to the list of resulting airports identified in that country.

The architects considered multiple approaches when designing the system. Separating the reference data as an independent subsystem like other subsystems was one of the options considered, but this could lead to performance issues. The team took the decision to follow an exception approach for handling reference data as compared to other transactions. Considering the nature of the query patterns discussed earlier, the approach was to use the reference data as a shared library.

In this case, the subsystems were allowed to access the reference data directly using pass-by-reference semantic data instead of going through the EJB interfaces. This also meant that irrespective of the subsystems, hibernate entities could use the reference data as a part of their entity relationships:



As depicted in the preceding diagram, the **Booking** entity in the reservation subsystem is allowed to use the reference data entities, in this case **Airport**, as part of their relationships.

Single database

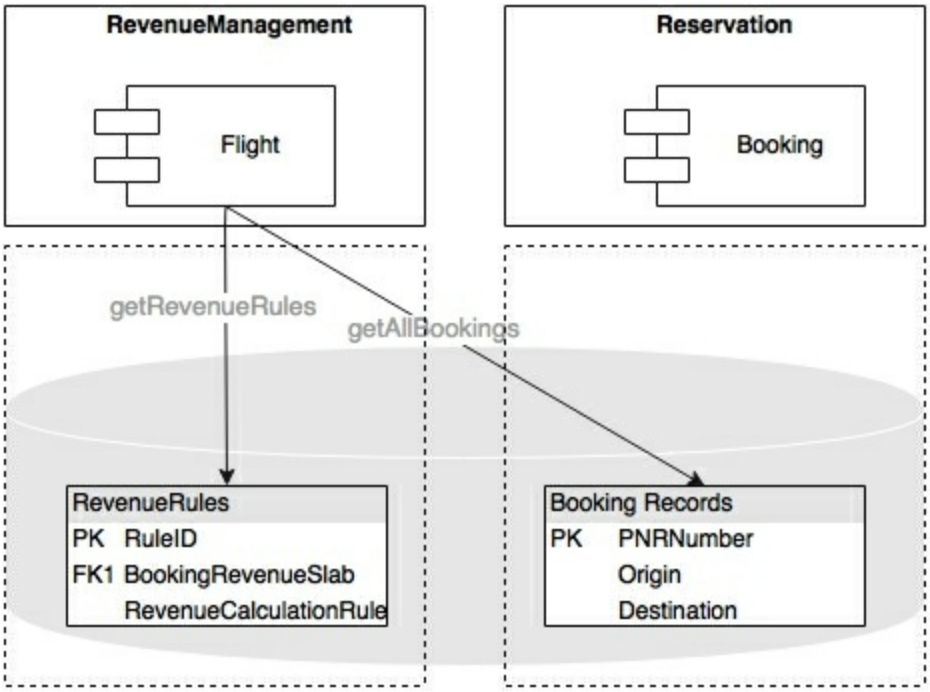
Though enough segregation was enforced at the middle tier, all functions pointed to a single database, even to the same database schema. The single schema approach opened a plethora of issues.

Native queries

The Hibernate framework provides a good abstraction over the underlying databases. It generates efficient SQL statements, in most of the cases targeting the database using specific dialects. However, sometimes, writing native JDBC SQLs offers better performance and resource efficiency. In some cases, using native database functions gives an even better performance.

The single database approach worked well at the beginning. But over a period of time, it opened up a loophole for the developers by connecting database tables owned by different subsystems. Native JDBC SQL was a good vehicle for doing this.

The following diagram shows an example of connecting two tables owned by two subsystems using a native JDBC SQL:



As shown in the preceding diagram, the Accounting component requires all booking records for a day, for a given city, from the Booking component to process the day-end billing. The subsystem-based design enforces Accounting to make a service call to Booking to get all booking records for a given city. Assume this results in *N* booking records. Now, for each booking record, Accounting has to execute a database call to find the applicable rules based on the fare code attached to each booking record. This could result in *N+1* JDBC calls, which is inefficient. Workarounds, such as batch queries or parallel and batch executions, are available, but this would lead to increased coding efforts and higher complexity. The developers tackled this issue with a native JDBC query as an easy-to-implement shortcut. Essentially, this approach could reduce the number of calls from *N+1* to a single database call, with minimal coding efforts.

This habit continued with many JDBC native queries connecting tables across multiple components and subsystems. This resulted not only in tightly coupled components, but also led to undocumented, hard-to-detect code.

Stored procedures

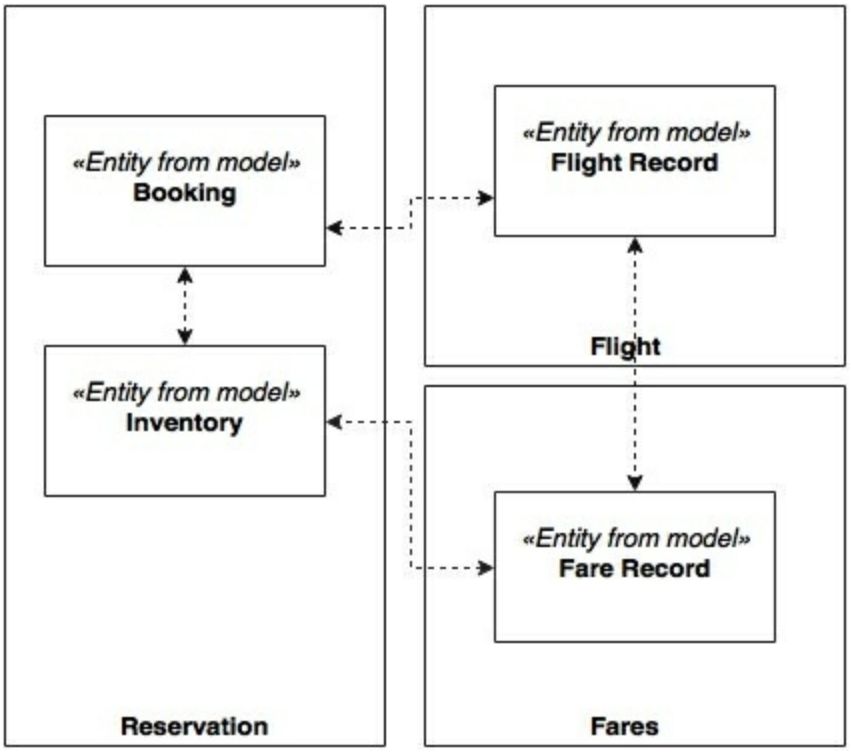
Another issue that surfaced as a result of the use of a single database was the use of complex stored procedures. Some of the complex data-centric logic written at the middle layer was not performing well, causing slow response, memory issues, and thread-blocking issues.

In order to address this problem, the developers took the decision to move some of the complex business logic from the middle tier to the database tier by implementing the logic directly within the stored procedures. This decision resulted in better performance of some of the transactions, and removed some of the stability issues. More and more procedures were added over a period of time. However, this eventually broke the application's modularity.

Domain boundaries

Though the domain boundaries were well established, all the components were packaged as a single EAR file. Since all the components were set to run on a single container, there was no stopping the developers referencing objects across these boundaries. Over a period of time, the project teams changed, delivery pressure increased, and the complexity grew tremendously. The developers started looking for quick solutions rather than the right ones. Slowly, but steadily, the modular nature of the application went away.

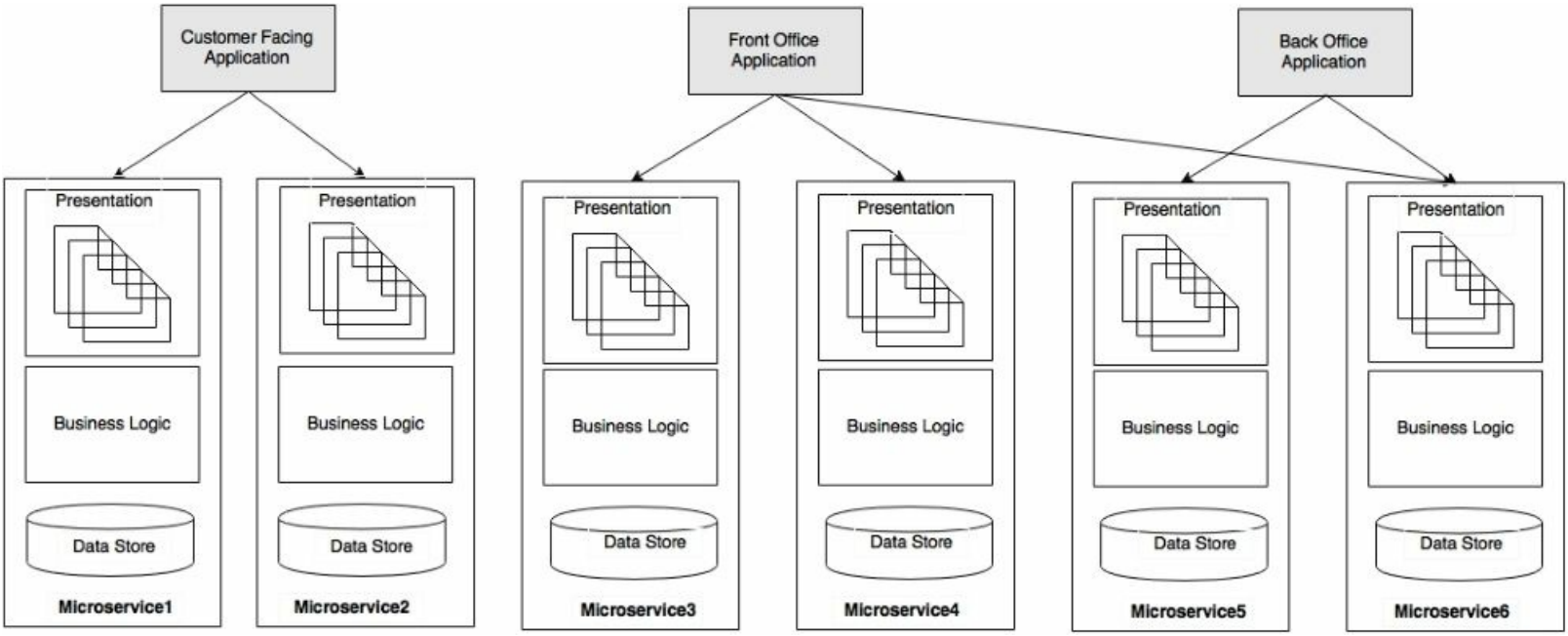
As depicted in the following diagram, hibernate relationships were created across subsystem boundaries:



Microservices to the rescue

There are not many improvement opportunities left to support the growing demand of BrownField Airline's business. BrownField Airline was looking to re-platform the system with an evolutionary approach rather than a revolutionary model.

Microservices is an ideal choice in these situations—for transforming a legacy monolithic application with minimal disruption to the business:



As shown in the preceding diagram, the objective is to move to a microservices-based architecture aligned to the business capabilities. Each microservice will hold the data store, the business logic, and the presentation layer.

The approach taken by BrownField Airline is to build a number of web portal applications targeting specific user communities such as customer facing, front office, and back office. The advantage of this approach lies in the flexibility for modeling, and also in the possibility to treat different communities differently. For example, the policies, architecture, and testing approaches for the Internet facing layer are different from the intranet-facing web application. Internet-facing applications may take advantage of **CDNs** (**Content Delivery Networks**) to move pages as close to the customer as possible, whereas intranet applications could serve pages directly from the data center.

The business case

When building business cases for migration, one of the commonly asked questions is "how does the microservices architecture avoid resurfacing of the same issues in another five years' time?"

Microservices offers a full list of benefits, which you learned in Chapter 1, *Demystifying Microservices*, but it is important to list a few here that are critical in this situation:

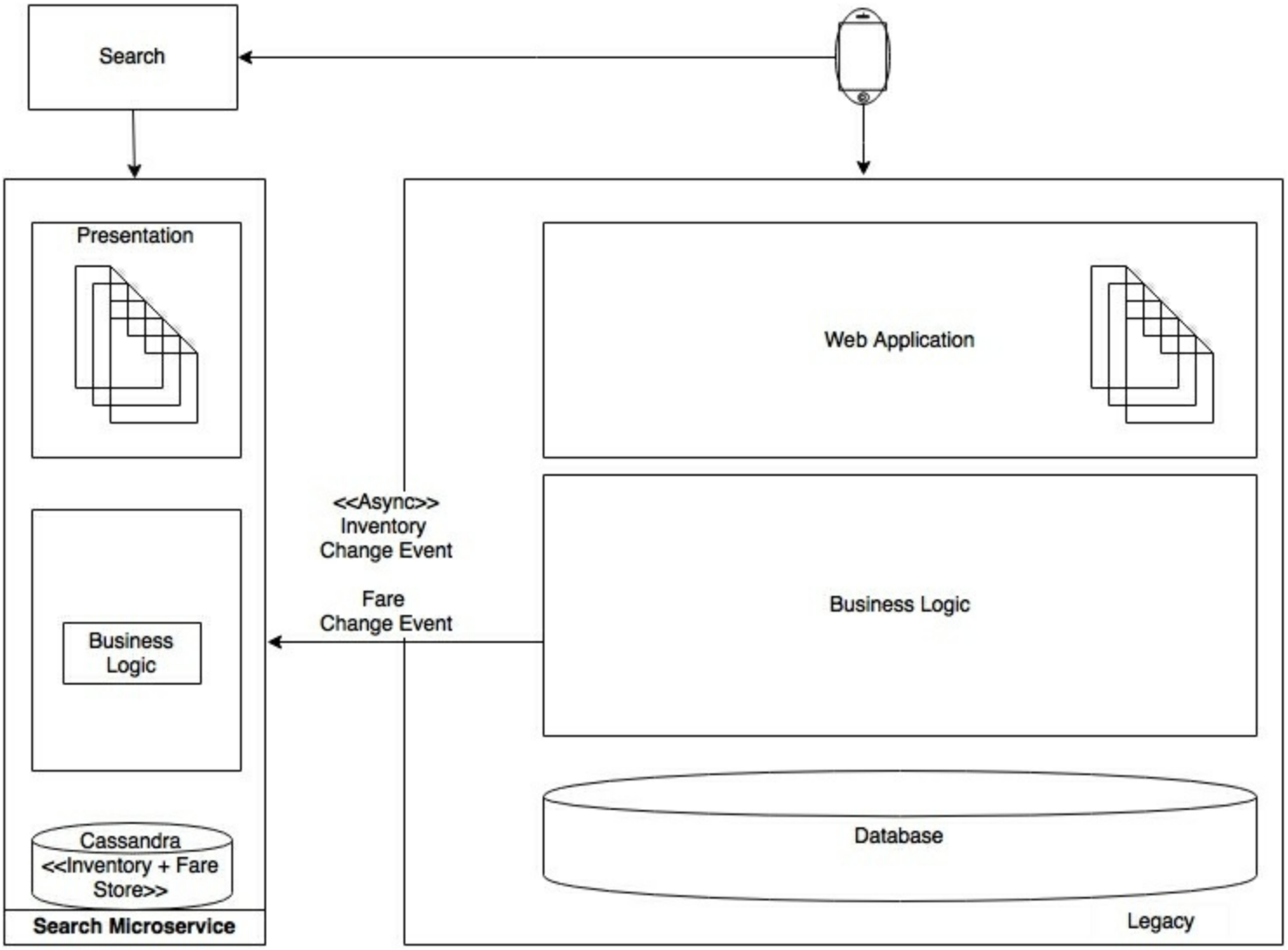
- **Service dependencies**: While migrating from monolithic applications to microservices, the dependencies are better known, and therefore the architects and developers are much better placed to avoid breaking dependencies and to future-proof dependency issues. Lessons from the monolithic application helps architects and developers to design a better system.
- **Physical boundaries**: Microservices enforce physical boundaries in all areas including the data store, the business logic, and the presentation layer. Access across subsystems or microservices are truly restricted due to their physical isolation. Beyond the physical boundaries, they could even run on different technologies.
- **Selective scaling**: Selective scale out is possible in microservices architecture. This provides a much more cost-effective scaling mechanism compared to the Y-scale approach used in the monolithic scenario.
- **Technology obsolescence**: Technology migrations could be applied at a microservices level rather than at the overall application level. Therefore, it does not require a humongous investment.

Plan the evolution

It is not simple to break an application that has millions of lines of code, especially if the code has complex dependencies. How do we break it? More importantly, where do we start, and how do we approach this problem?

Evolutionary approach

The best way to address this problem is to establish a transition plan, and gradually migrate the functions as microservices. At every step, a microservice will be created outside of the monolithic application, and traffic will be diverted to the new service as shown in the following diagram:



In order to run this migration successfully, a number of key questions need to be answered from the transition point of view:

- Identification of microservices' boundaries
- Prioritizing microservices for migration
- Handling data synchronization during the transition phase
- Handling user interface integration, working with old and new user interfaces
- Handling of reference data in the new system
- Testing strategy to ensure the business capabilities are intact and correctly reproduced
- Identification of any prerequisites for microservice development such as microservices capabilities, frameworks, processes, and so on

Identification of microservices boundaries

The first and foremost activity is to identify the microservices' boundaries. This is the most interesting part of the problem, and the most difficult part as well. If identification of the boundaries is not done properly, the migration could lead to more complex manageability issues.

Like in SOA, a service decomposition is the best way to identify services. However, it is important to note that decomposition stops at a business capability or bounded context. In SOA, service decomposition goes further into an atomic, granular service level.

A top-down approach is typically used for domain decomposition. The bottom-up approach is also useful in the case of breaking an existing system, as it can utilize a lot of practical knowledge, functions, and behaviors of the existing monolithic application.

The previous decomposition step will give a potential list of microservices. It is important to note that this isn't the final list of microservices, but it serves as a good starting point. We will run through a number of filtering mechanisms to get to a final list. The first cut of functional decomposition will, in this case, be similar to the diagram shown under the functional view introduced earlier in this chapter.

Analyze dependencies

The next step is to analyze the dependencies between the initial set of candidate microservices that we created in the previous section. At the end of this activity, a dependency graph will be produced.

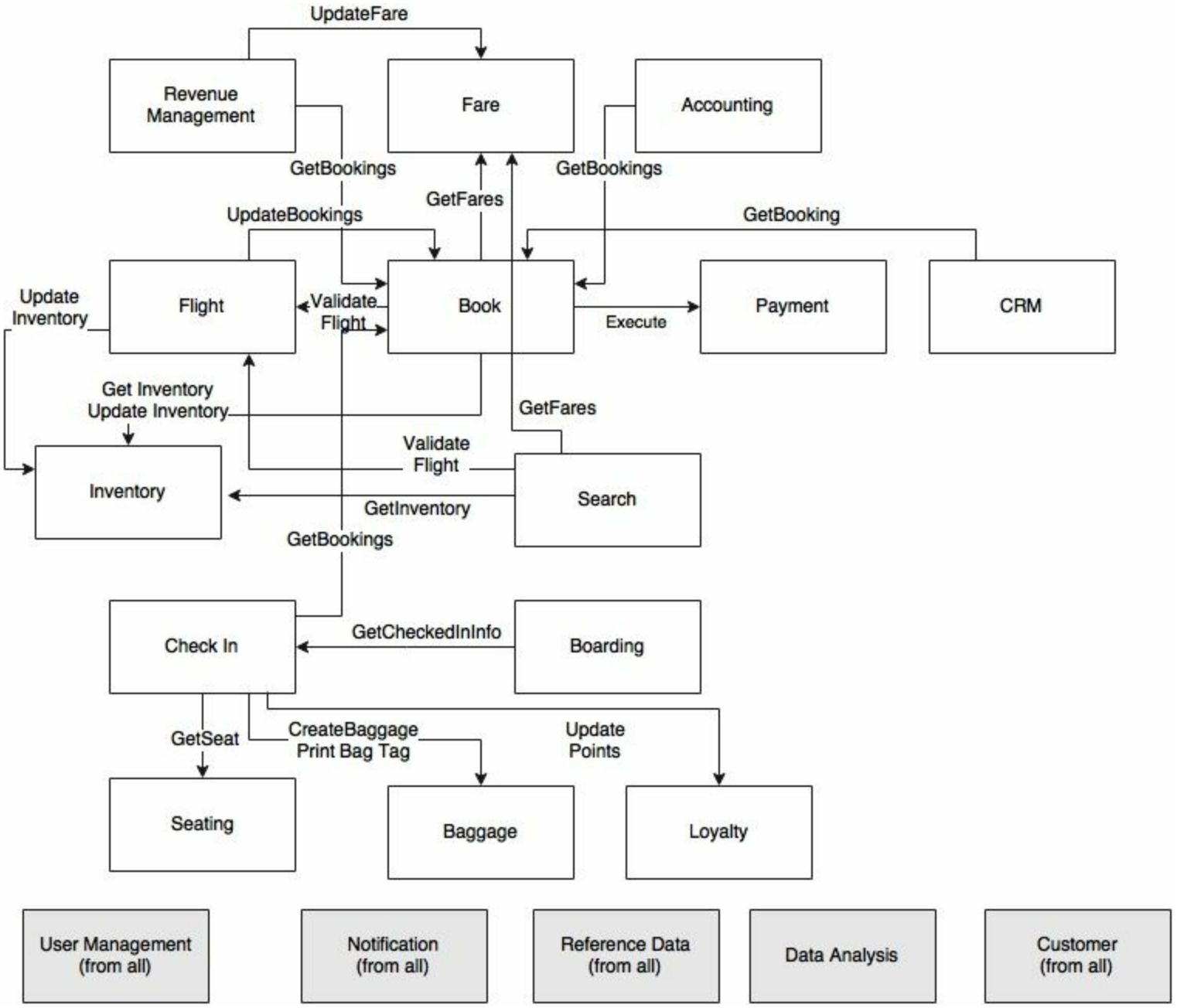
Note

A team of architects, business analysts, developers, release management and support staff is required for this exercise.

One way to produce a dependency graph is to list out all the components of the legacy system and overlay dependencies. This could be done by combining one or more of the approaches listed as follows:

- Analyzing the manual code and regenerating dependencies.
- Using the experience of the development team to regenerate dependencies.
- Using a Maven dependency graph. There are a number of tools we could use to regenerate the dependency graph, such as PomExplorer, PomParser, and so on.
- Using performance engineering tools such as AppDynamics to identify the call stack and roll up dependencies.

Let us assume that we reproduce the functions and their dependencies as shown in the following diagram:



There are many dependencies going back and forth between different modules. The bottom layer shows cross-cutting capabilities that are used across multiple modules. At this point, the modules are more like spaghetti than autonomous units.

The next step is to analyze these dependencies, and come up with a better, simplified dependency map.

Events as opposed to query

Dependencies could be query-based or event-based. Event-based is better for scalable systems. Sometimes, it is possible to convert query-based communications to event-based ones. In many cases, these dependencies exist because either the business organizations are managed like that, or by virtue of the way the old system handled the business scenario.

From the previous diagram, we can extract the Revenue Management and the Fares services:



Revenue Management is a module used for calculating optimal fare values, based on the booking demand forecast. In case of a fare change between an origin and a destination, Update Fare on the Fare module is called by Revenue Management to update the respective fares in the Fare module.

An alternate way of thinking is that the Fare module is subscribed to Revenue Management for any changes in fares, and Revenue Management publishes whenever there is a fare change. This reactive programming approach gives an added flexibility by which the Fares and the Revenue Management modules could stay independent, and connect them through a reliable messaging system. This same pattern could be applied in many other scenarios from Check-In to the Loyalty and Boarding modules.

Next, examine the scenario of CRM and Booking:



This scenario is slightly different from the previously explained scenario. The CRM module is used to manage passenger complaints. When CRM receives a complaint, it retrieves the corresponding passenger's Booking data. In reality, the number of complaints are negligibly small when compared to the number of bookings. If we blindly apply the previous pattern where CRM subscribes to all bookings, we will find that it is not cost effective:



Examine another scenario between the Check-in and Booking modules. Instead of Check-in calling the Get Bookings service on Booking, can Check-in listen to booking events? This is possible, but the challenge here is that a booking can happen 360 days in advance, whereas Check-in generally starts only 24 hours before the fight departure. Duplicating all bookings and booking changes in the Check-in module 360 days in advance would not be a wise decision as Check-in does not require this data until 24 hours before the flight departure.

An alternate option is that when check-in opens for a flight (24 hours before departure), Check-in calls a service on the Booking module to get a snapshot of the bookings for a given flight. Once this is done, Check-in could subscribe for booking events specifically for that flight. In this case, a combination of query-based as well as event-based approaches is used. By doing so, we reduce the unnecessary events and storage apart from reducing the number of queries between these two services.

In short, there is no single policy that rules all scenarios. Each scenario requires logical thinking, and then the most appropriate pattern is applied.

Events as opposed to synchronous updates

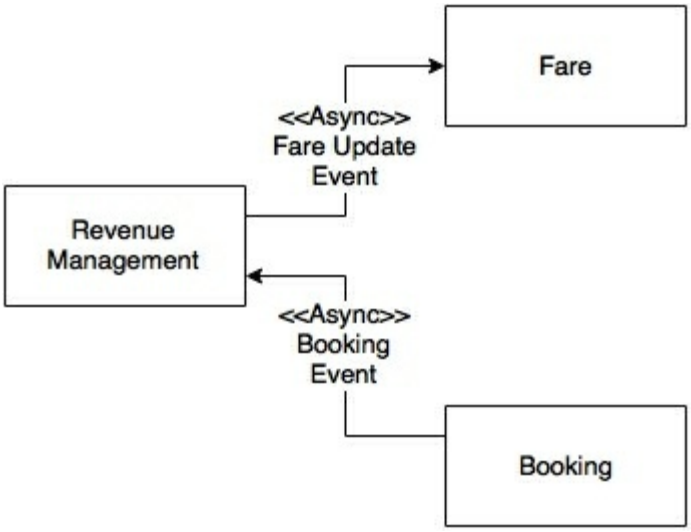
Apart from the query model, a dependency could be an update transaction as well. Consider the scenario between Revenue Management and Booking:



In order to do a forecast and analysis of the current demand, Revenue Management requires all bookings across all flights. The current approach, as depicted in the dependency graph, is that Revenue Management has a schedule job that calls Get Booking on Booking to get all incremental bookings (new and changed) since the last synchronization.

An alternative approach is to send new bookings and the changes in bookings as soon as they take place in the Booking module as an asynchronous push. The same pattern could be applied in many other scenarios such as from Booking to Accounting, from Flight to Inventory, and also from Flight to Booking. In this approach, the source service publishes all state-change events to a topic. All interested parties could subscribe to this event stream and store locally. This approach removes many hard wirings, and keeps the systems loosely coupled.

The dependency is depicted in the next diagram:



In this case depicted in the preceding diagram, we changed both dependencies and converted them to asynchronous events.

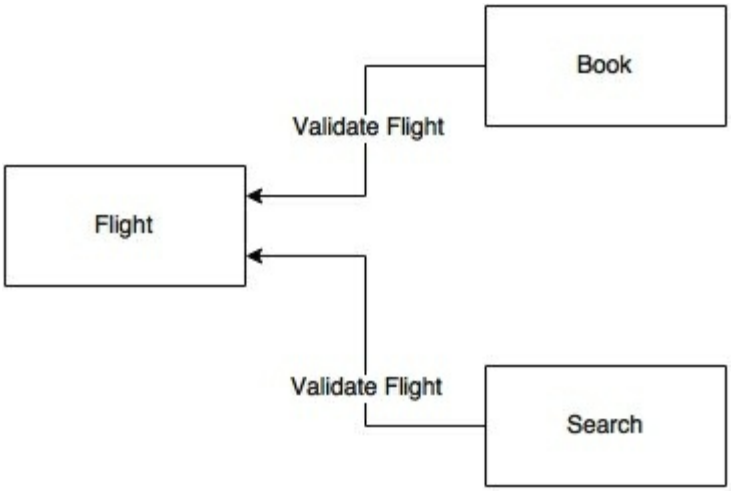
One last case to analyze is the Update Inventory call from the Booking module to the Inventory module:



When a booking is completed, the inventory status is updated by depleting the inventory stored in the Inventory service. For example, when there are 10 economy class seats available, at the end of the booking, we have to reduce it to 9. In the current system, booking and updating inventory are executed within the same transaction boundaries. This is to handle a scenario in which there is only one seat left, and multiple customers are trying to book. In the new design, if we apply the same event-driven pattern, sending the inventory update as an event to Inventory may leave the system in an inconsistent state. This needs further analysis, which we will address later in this chapter.

Challenge requirements

In many cases, the targeted state could be achieved by taking another look at the requirements:



There are two Validate Flight calls, one from Booking and another one from the Search module. The Validate Flight call is to validate the input flight data coming from different channels. The end objective is to avoid incorrect data stored or serviced. When a customer does a flight search, say "BF100", the system validates this flight to see the following things:

- Whether this is a valid flight?
- Whether the flight exists on that particular date?
- Are there any booking restrictions set on this flight?

An alternate way of solving this is to adjust the inventory of the flight based on these given conditions. For example, if there is a restriction on the flight, update the inventory as zero. In this case, the intelligence will remain with Flight, and it keeps updating the inventory. As far as Search and Booking are concerned, both just look up the inventory instead of validating flights for every request. This approach is more efficient as compared to the original approach.

Next we will review the Payment use case. Payment is typically a disconnected function due to the nature of security constraints such as PCIDSS-like standards. The most obvious way to capture a payment is to redirect a browser to a payment page hosted in the Payment service. Since card handling applications come under the purview of PCIDSS, it is wise to remove any direct dependencies from the Payment service. Therefore, we can remove the Booking-to-Payment direct dependency, and opt for a UI-level integration.

Challenge service boundaries

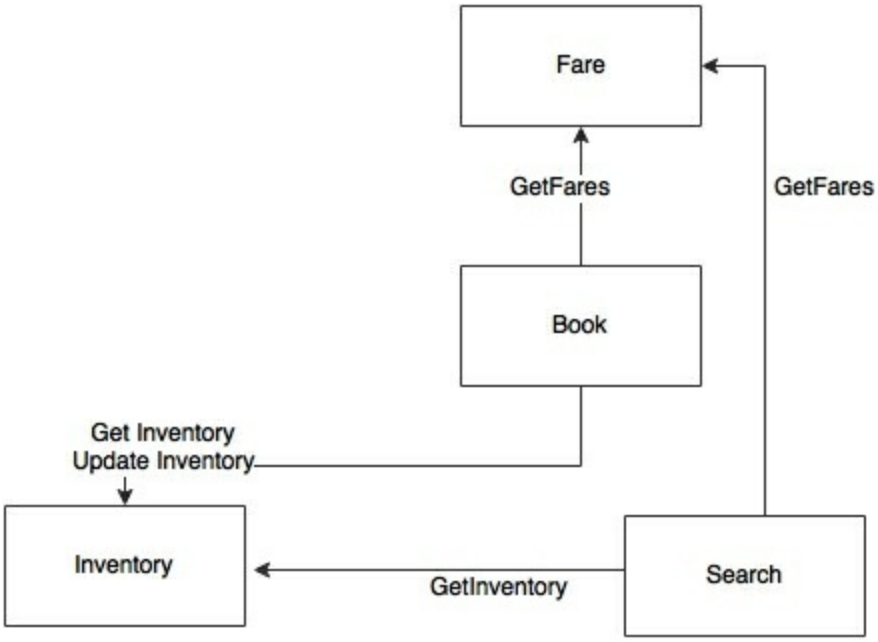
In this section, we will review some of the service boundaries based on the requirements and dependency graph, considering Check-in and its dependencies to Seating and Baggage.

The Seating function runs a few algorithms based on the current state of the seat allocation in the airplane, and finds out the best way to position the next passenger so that the weight and balance requirements can be met. This is based on a number of predefined business rules. However, other than Check-in, no other module is interested in the Seating function. From a business capability

perspective, Seating is just a function of Check-in, not a business capability by itself. Therefore, it is better to embed this logic inside Check-in itself.

The same is applicable to Baggage as well. BrownField has a separate baggage handling system. The Baggage function in the PSS context is to print the baggage tag as well as store the baggage data against the Check-in records. There is no business capability associated with this particular functionality. Therefore, it is ideal to move this function to Check-in itself.

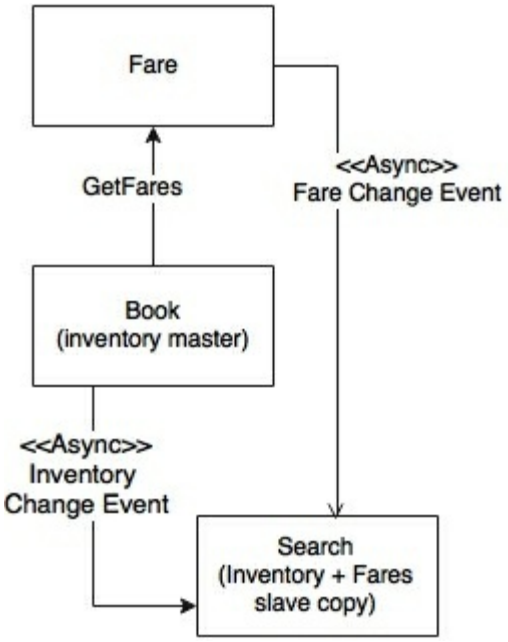
The Book, Search, and Inventory functions, after redesigning, are shown in the following diagram:



Similarly, Inventory and Search are more supporting functions of the Booking module. They are not aligned with any of the business capabilities as such. Similar to the previous judgement, it is ideal to move both the Search and Inventory functions to Booking. Assume, for the time being, that Search, Inventory, and Booking are moved to a single microservice named Reservation.

As per the statistics of BrownField, search transactions are 10 times more frequent than the booking transactions. Moreover, search is not a revenue-generating transaction when compared to booking. Due to these reasons, we need different scalability models for search and booking. Booking should not get impacted if there is a sudden surge of transactions in search. From the business point of view, dropping a search transaction in favor of saving a valid booking transaction is more acceptable.

This is an example of a polyglot requirement, which overrules the business capability alignment. In this case, it makes more sense to have Search as a service separate from the Booking service. Let us assume that we remove Search. Only Inventory and Booking remain under Reservation. Now Search has to hit back to Reservation to perform inventory searches. This could impact the booking transactions:



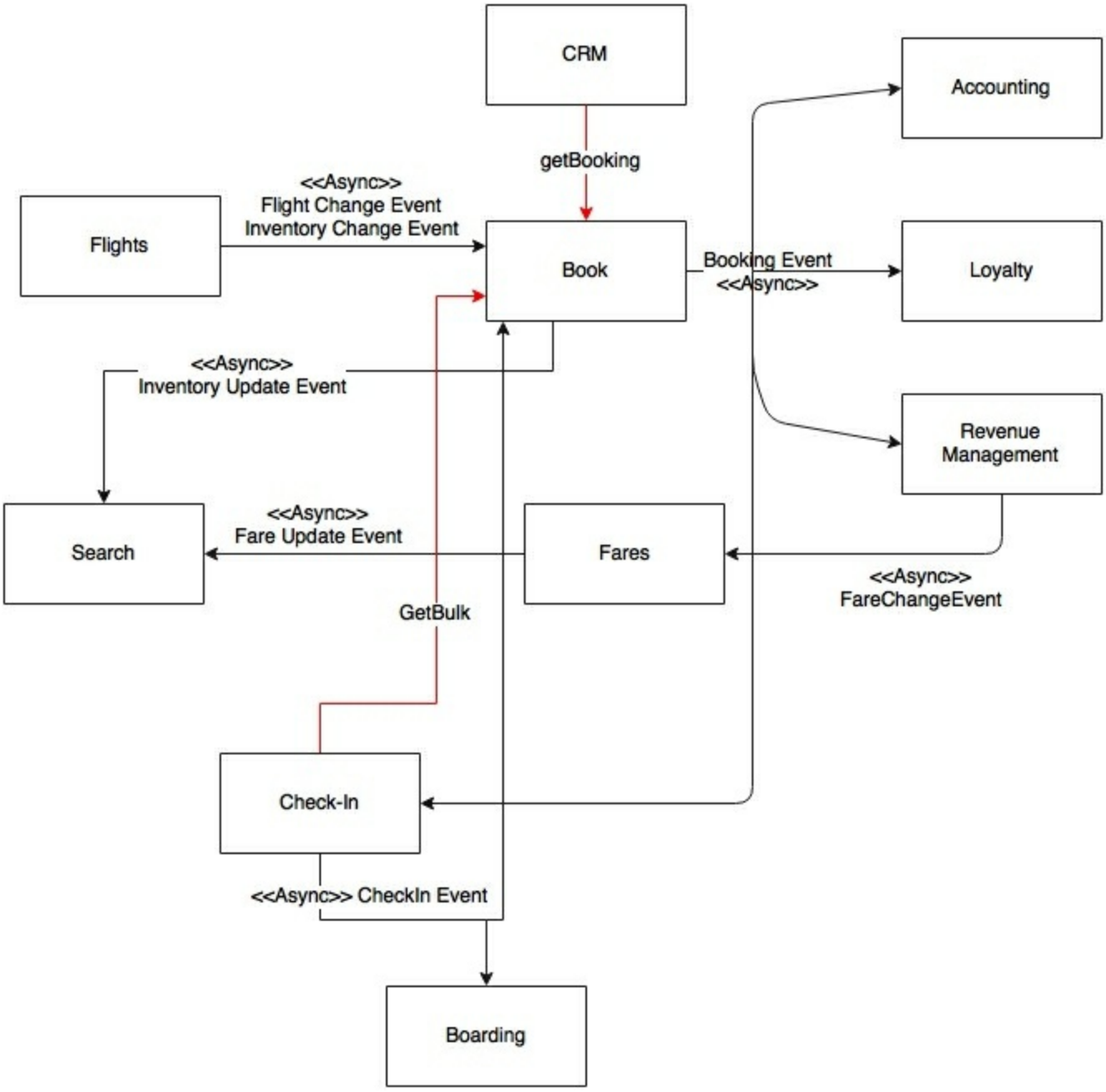
A better approach is to keep Inventory along with the Booking module, and keep a read-only copy of the inventory under Search, while continuously synchronizing the inventory data over a reliable messaging system. Since both Inventory and Booking are collocated, this will also solve the need to have two-phase commits. Since both of them are local, they could work well with local transactions.

Let us now challenge the Fare module design. When a customer searches for a flight between A and B for a given date, we would like to show the flights and fares together. That means that our read-only copy of inventory can also combine both fares as well as inventory. Search will then subscribe to Fare for any fare change events. The intelligence still stays with the Fare service, but it keeps sending fare updates to the cached fare data under Search.

Final dependency graph

There are still a few synchronized calls, which, for the time being, we will keep as they are.

By applying all these changes, the final dependency diagram will look like the following one:



Now we can safely consider each box in the preceding diagram as a microservice. We have nailed down many dependencies, and modeled many of them as asynchronous as well. The overall system is more or less designed in the reactive style. There are still some synchronized calls shown in the diagram with bold lines, such as Get Bulk from Check-In, Get Booking from CRM, and Get Fare from Booking. These synchronous calls are essentially required as per the trade-off analysis.

Prioritizing microservices for migration

We have identified a first-cut version of our microservices-based architecture. As the next step, we will analyze the priorities, and identify the order of migration. This could be done by considering multiple factors explained as follows:

- **Dependency**: One of the parameters for deciding the priority is the dependency graph. From the service dependency graph, services with less dependency or no dependency at all are easy to migrate, whereas complex dependencies are way harder. Services with complex dependencies will also need dependent modules to be migrated along with them.

  Accounting, Loyalty, CRM, and Boarding have less dependencies as compared to Booking and Check-in. Modules with high dependencies will also have higher risks in their migration.
- **Transaction volume**: Another parameter that can be applied is analyzing the transaction volumes. Migrating services with the highest transaction volumes will relieve the load on the existing system. This will have more value from an IT support and maintenance perspective. However, the downside of this approach is the higher risk factor.

  As stated earlier, Search requests are ten times higher in volume as compared to Booking requests. Requests for Check-in are the third-highest in volume transaction after Search and Booking.
- **Resource utilization**: Resource utilization is measured based on the current utilizations such as CPU, memory, connection pools, thread pools, and so on. Migrating resource intensive services out of the legacy system provides relief to other services. This helps the remaining modules to function better.

  Flight, Revenue Management, and Accounting are resource-intensive services, as they involve data-intensive transactions such as forecasting, billing, flight schedule changes, and so on.
- **Complexity**: Complexity is perhaps measured in terms of the business logic associated with a service such as function points, lines of code, number of tables, number of services, and others. Less complex modules are easy to migrate as compared to the more complex ones.

  Booking is extremely complex as compared to the Boarding, Search, and Check-in services.
- **Business criticality**: The business criticality could be either based on revenue or customer experience. Highly critical modules deliver higher business value.

  Booking is the most revenue-generating service from the business stand point, whereas Check-in is business critical as it could lead to flight departure delays, which could lead to revenue loss as well as customer dissatisfaction.
- **Velocity of changes**: Velocity of change indicates the number of change requests targeting a function in a short time frame. This translates to speed and agility of delivery. Services with high velocity of change requests are better candidates for migration as compared to stable modules.

  Statistics show that Search, Booking, and Fares go through frequent changes, whereas Check-in

is the most stable function.

- **Innovation**: Services that are part of a disruptive innovative process need to get priority over back office functions that are based on more established business processes. Innovations in legacy systems are harder to achieve as compared to applying innovations in the microservices world.

  Most of the innovations are around Search, Booking, Fares, Revenue Management, and Check-in as compared to back office Accounting.

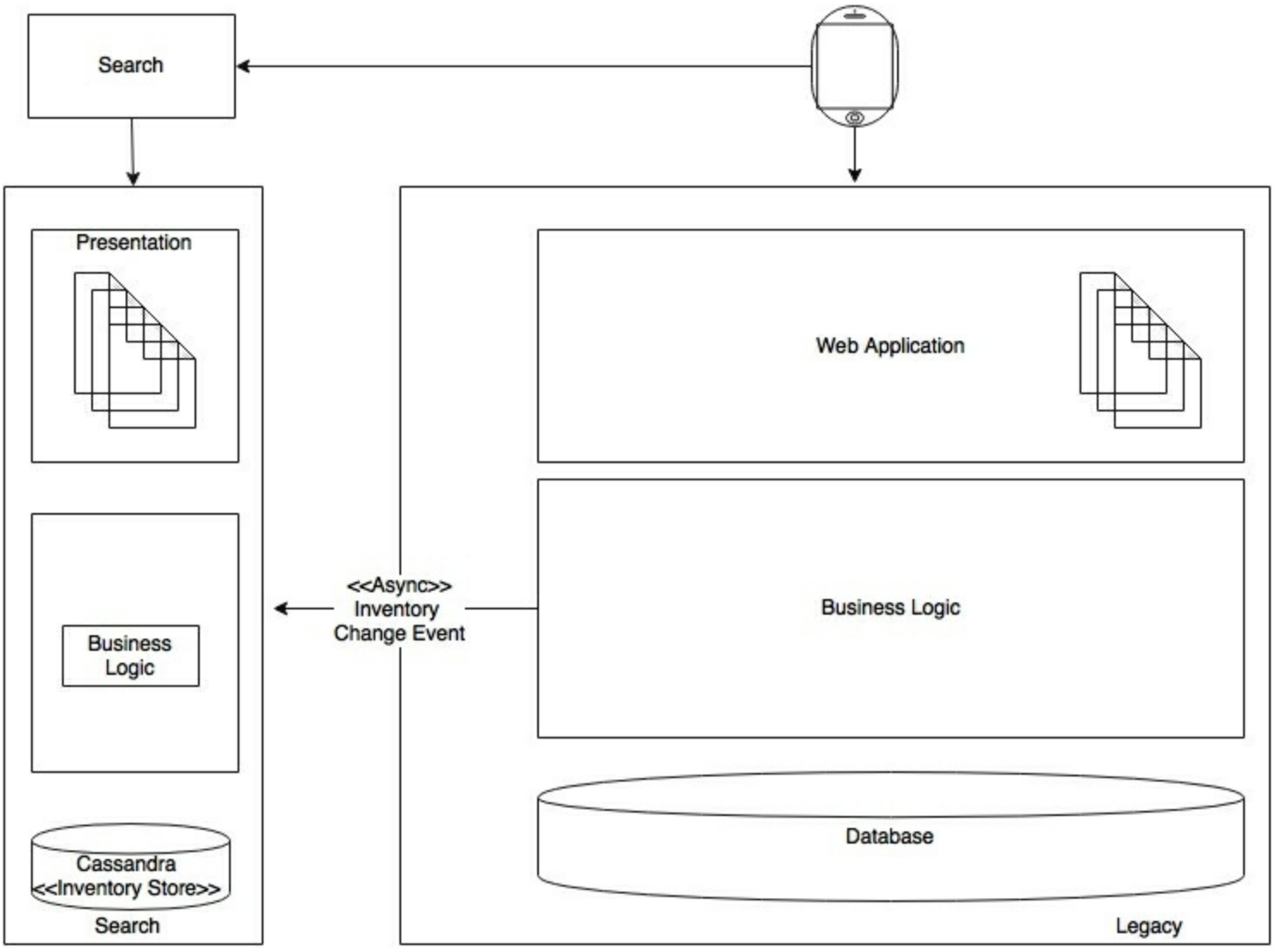
Based on BrownField's analysis, Search has the highest priority, as it requires innovation, has high velocity of changes, is less business critical, and gives better relief for both business and IT. The Search service has minimal dependency with no requirements to synchronize data back to the legacy system.

Data synchronization during migration

During the transition phase, the legacy system and the new microservices will run in parallel. Therefore, it is important to keep the data synchronized between the two systems.

The simplest option is to synchronize the data between the two systems at the database level by using any data synchronization tool. This approach works well when both the old and the new systems are built on the same data store technologies. The complexity will be higher if the data store technologies are different. The second problem with this approach is that we allow a backdoor entry, hence exposing the microservices' internal data store outside. This is against the principle of microservices.

Let us take this on a case-by-case basis before we can conclude with a generic solution. The following diagram shows the data migration and synchronization aspect once Search is taken out:

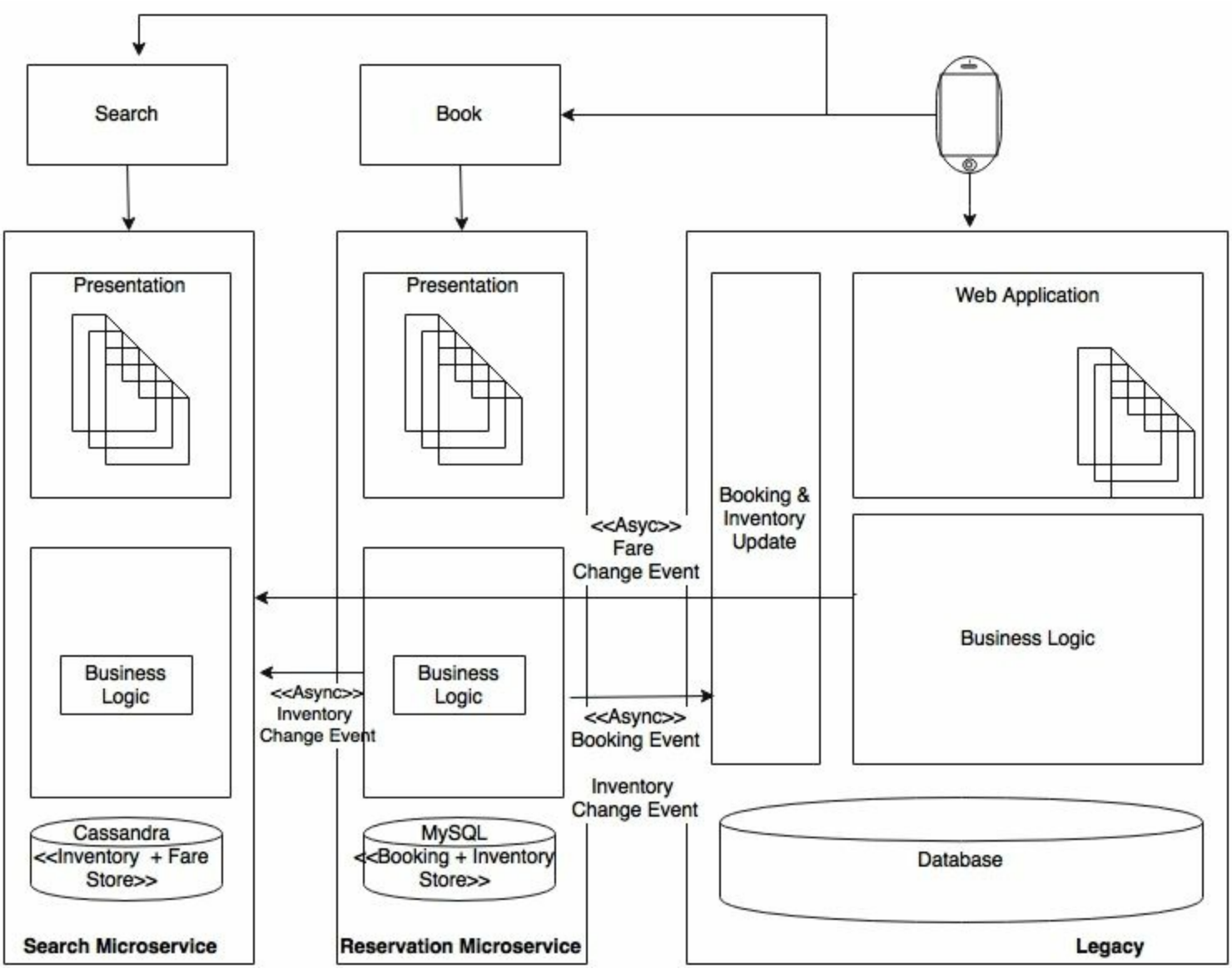


Let us assume that we use a NoSQL database for keeping inventory and fares under the Search service. In this particular case, all we need is the legacy system to supply data to the new service using asynchronous events. We will have to make some changes in the existing system to send the fare

changes or any inventory changes as events. The Search service then accepts these events, and stores them locally into the local NoSQL store.

This is a bit more tedious in the case of the complex Booking service.

In this case, the new Booking microservice sends the inventory change events to the Search service. In addition to this, the legacy application also has to send the fare change events to Search. Booking will then store the new Booking service in its My SQL data store.



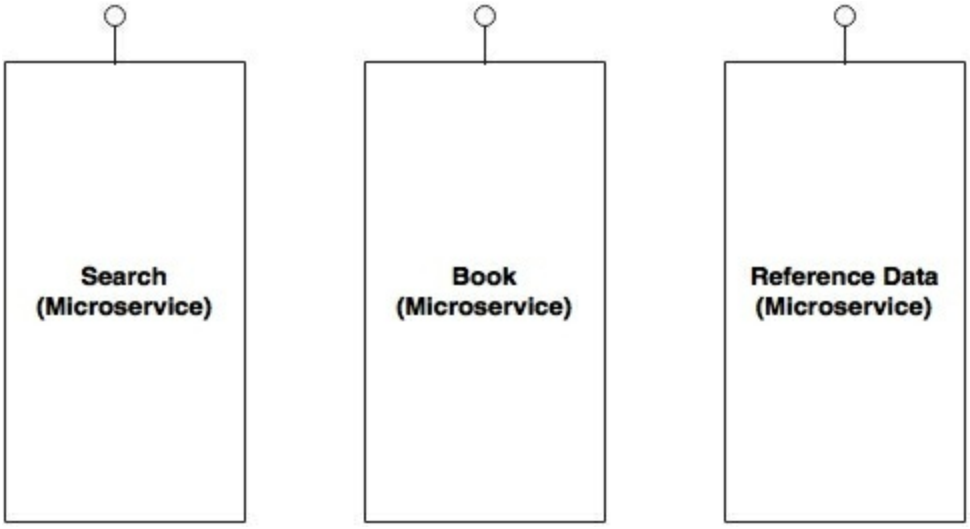
The most complex piece, the Booking service, has to send the booking events and the inventory events back to the legacy system. This is to ensure that the functions in the legacy system continue to work as before. The simplest approach is to write an update component which accepts the events and updates the old booking records table so that there are no changes required in the other legacy modules. We will continue this until none of the legacy components are referring the booking and inventory data. This will help us minimize changes in the legacy system, and therefore, reduce the risk of failures.

In short, a single approach may not be sufficient. A multi-pronged approach based on different

patterns is required.

Managing reference data

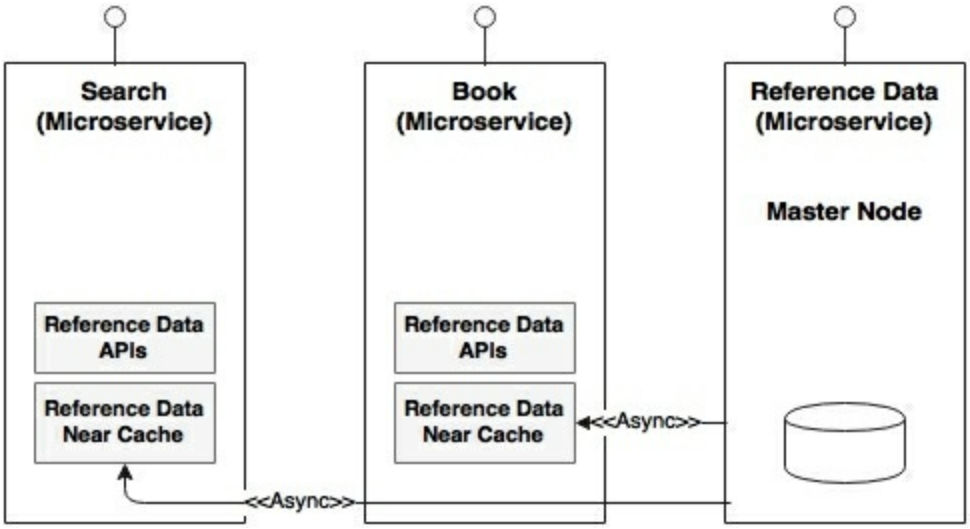
One of the biggest challenges in migrating monolithic applications to microservices is managing reference data. A simple approach is to build the reference data as another microservice itself as shown in the following diagram:



In this case, whoever needs reference data should access it through the microservice endpoints. This is a well-structured approach, but could lead to performance issues as encountered in the original legacy system.

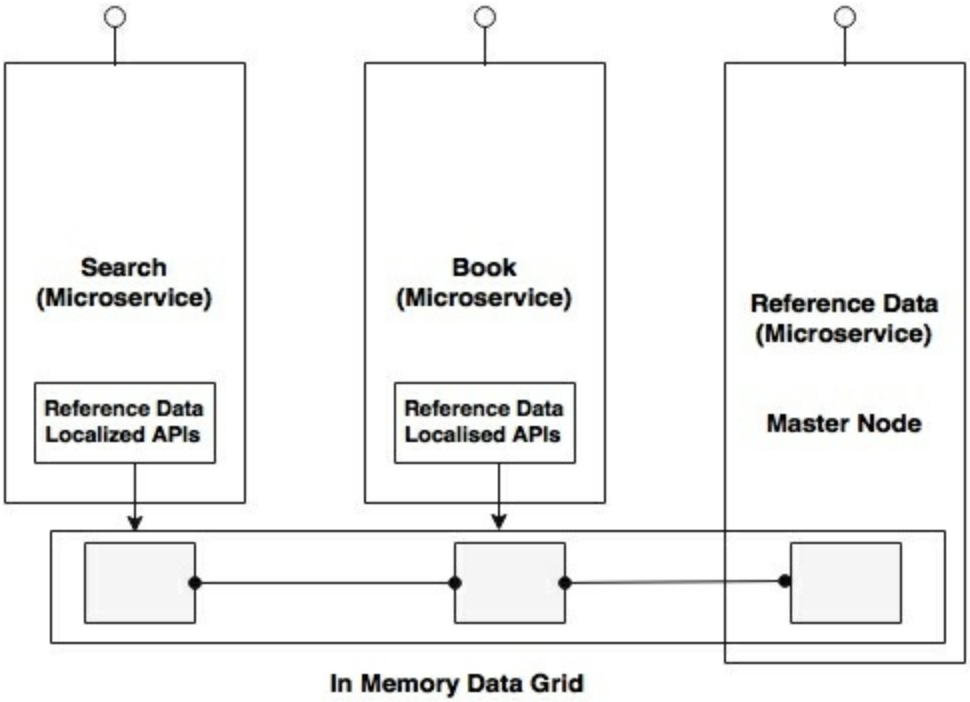
An alternate approach is to have reference data as a microservice service for all the admin and CRUD functions. A near cache will then be created under each service to incrementally cache data from the master services. A thin reference data access proxy library will be embedded in each of these services. The reference data access proxy abstracts whether the data is coming from cache or from a remote service.

This is depicted in the next diagram. The master node in the given diagram is the actual reference data microservice:



The challenge is to synchronize the data between the master and the slave. A subscription mechanism is required for those data caches that change frequently.

A better approach is to replace the local cache with an in-memory data grid, as shown in the following diagram:

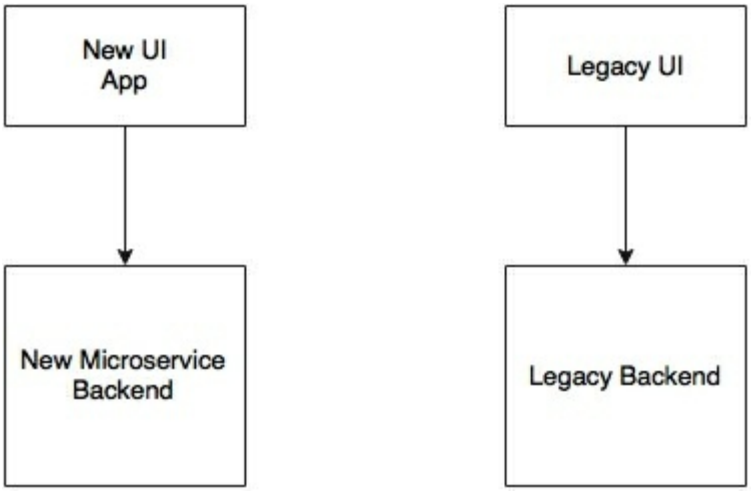


The reference data microservice will write to the data grid, whereas the proxy libraries embedded in other services will have read-only APIs. This eliminates the requirement to have subscription of data, and is much more efficient and consistent.

User interfaces and web applications

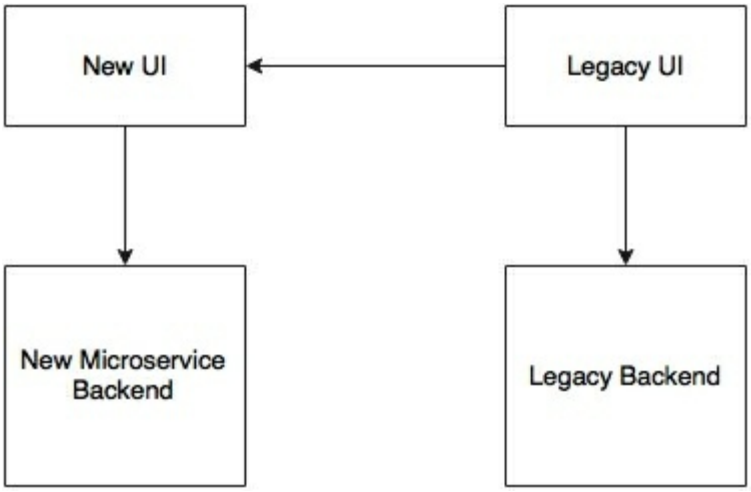
During the transition phase, we have to keep both the old and new user interfaces together. There are three general approaches usually taken in this scenario.

The first approach is to have the old and new user interfaces as separate user applications with no link between them, as depicted in the following diagram:



A user signs in to the new application as well as into the old application, much like two different applications, with no **single sign-on** (**SSO**) between them. This approach is simple, and there is no overhead. In most of the cases, this may not be acceptable to the business unless it is targeted at two different user communities.

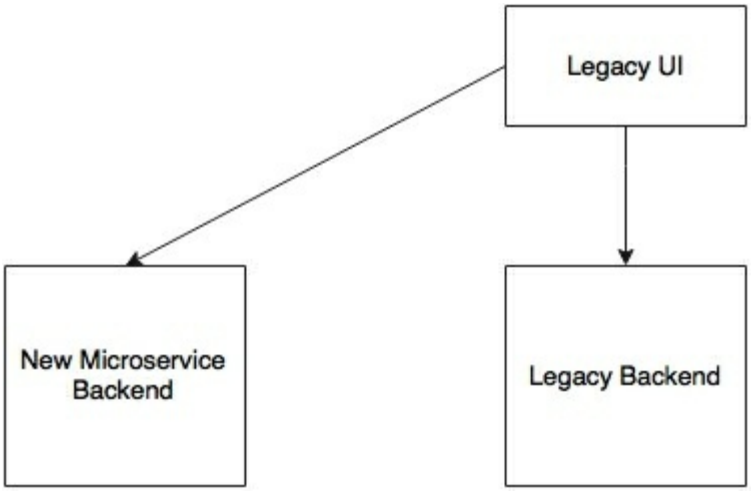
The second approach is to use the legacy user interface as the primary application, and then transfer page controls to the new user interfaces when the user requests pages of the new application:



In this case, since the old and the new applications are web-based applications running in a web

browser window, users will get a seamless experience. SSO has to be implemented between the old and the new user interfaces.

The third approach is to integrate the existing legacy user interface directly to the new microservices backend, as shown in the next diagram:



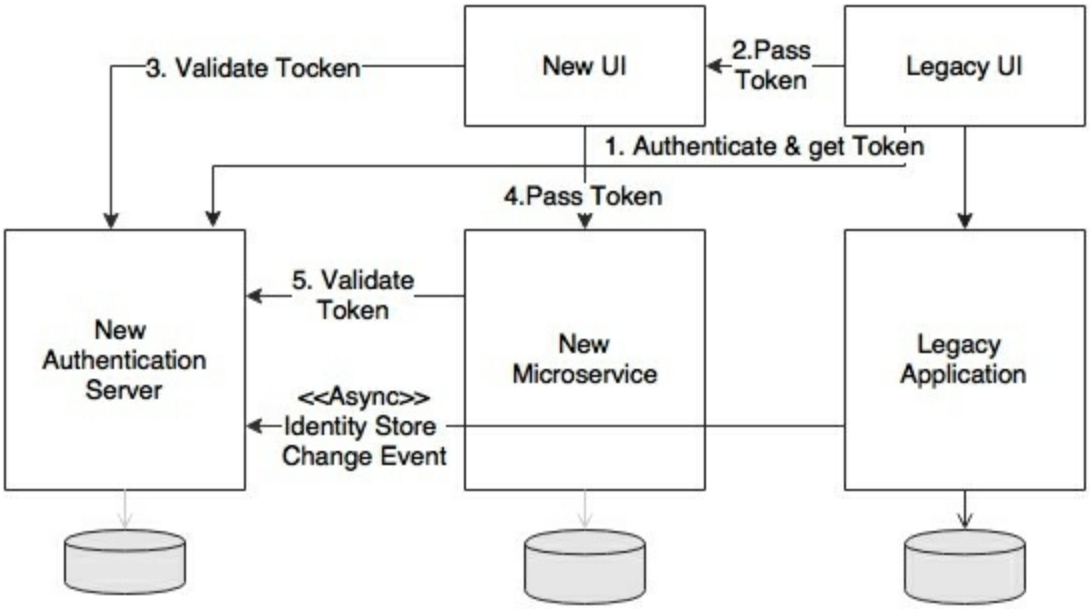
In this case, the new microservices are built as headless applications with no presentation layer. This could be challenging, as it may require many changes in the old user interface such as introducing service calls, data model conversions, and so on.

Another issue in the last two cases is how to handle the authentication of resources and services.

Session handling and security

Assume that the new services are written based on Spring Security with a token-based authorization strategy, whereas the old application uses a custom-built authentication with its local identity store.

The following diagram shows how to integrate between the old and the new services:



The simplest approach, as shown in the preceding diagram, is to build a new identity store with an authentication service as a new microservice using Spring Security. This will be used for all our future resource and service protections, for all microservices.

The existing user interface application authenticates itself against the new authentication service, and secures a token. This token will be passed to the new user interface or new microservice. In both cases, the user interface or microservice will make a call to the authentication service to validate the given token. If the token is valid, then the UI or microservice accepts the call.

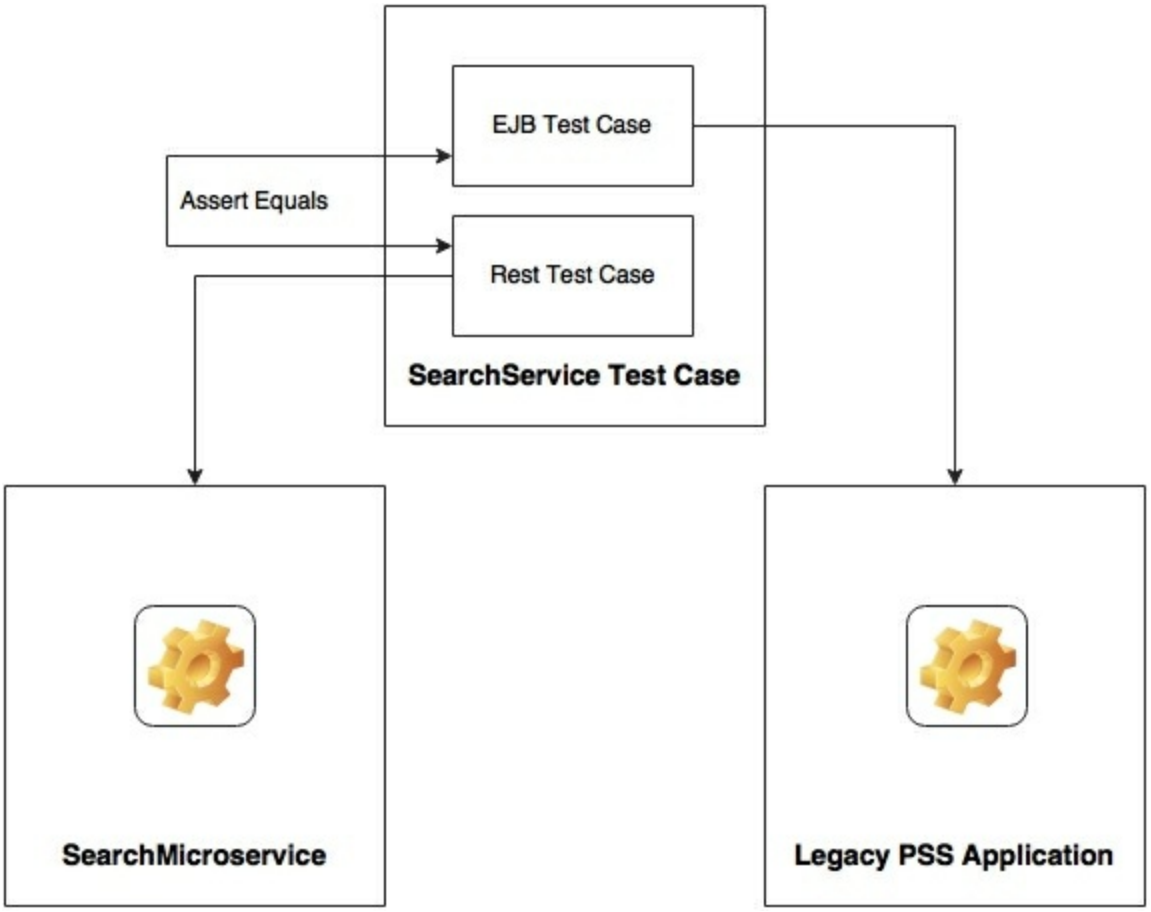
The catch here is that the legacy identity store has to be synchronized with the new one.

Test strategy

One important question to answer from a testing point of view is how can we ensure that all functions work in the same way as before the migration?

Integration test cases should be written for the services that are getting migrated before the migration or refactoring. This ensures that once migrated, we get the same expected result, and the behavior of the system remains the same. An automated regression test pack has to be in place, and has to be executed every time we make a change in the new or old system.

In the following diagram, for each service we need one test against the EJB endpoint, and another one against the microservices endpoint:



Building ecosystem capabilities

Before we embark on actual migration, we have to build all of the microservice's capabilities mentioned under the capability model, as documented in Chapter 3, *Applying Microservices Concepts*. These are the prerequisites for developing microservices-based systems.

In addition to these capabilities, certain application functions are also required to be built upfront such as reference data, security and SSO, and Customer and Notification. A data warehouse or a data lake is also required as a prerequisite. An effective approach is to build these capabilities in an incremental fashion, delaying development until it is really required.

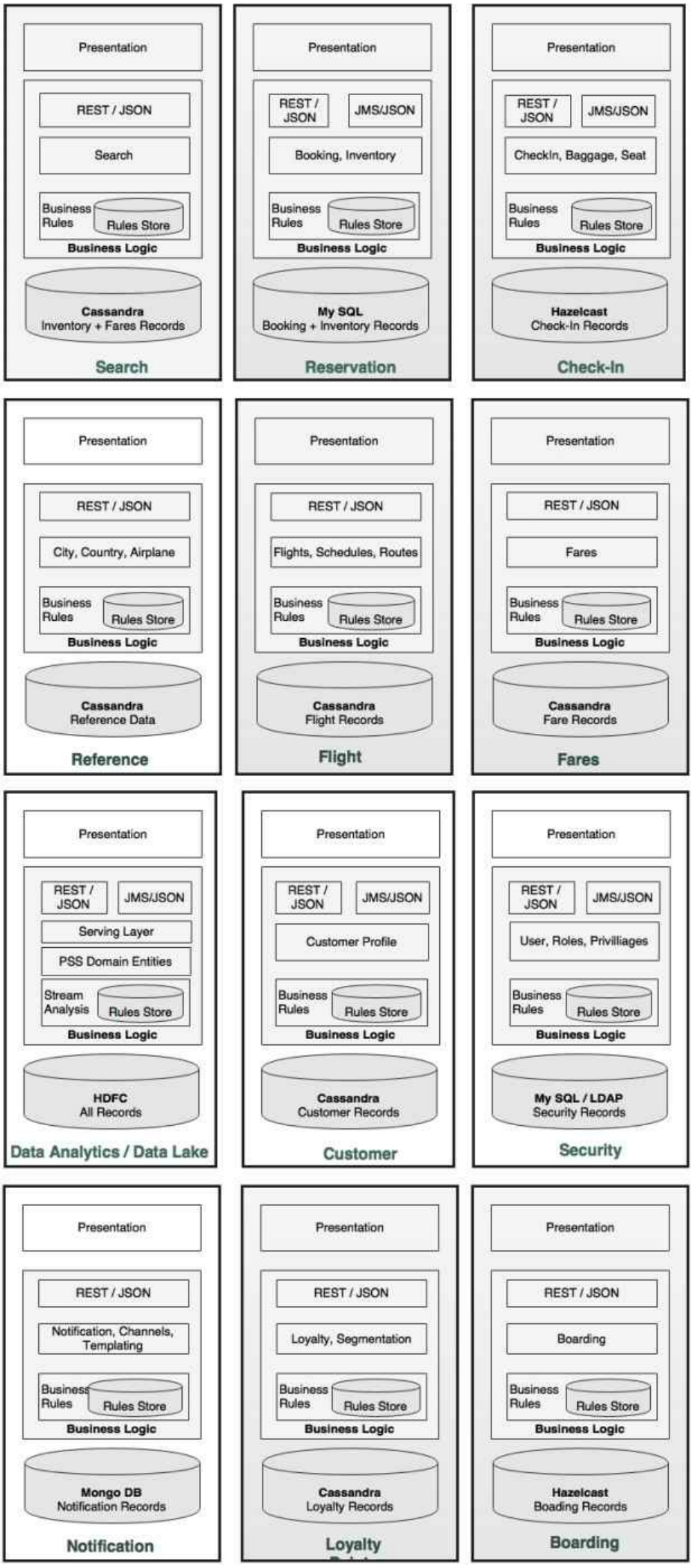
Migrate modules only if required

In the previous chapters, we have examined approaches and steps for transforming from a monolithic application to microservices. It is important to understand that it is not necessary to migrate all modules to the new microservices architecture, unless it is really required. A major reason is that these migrations incur cost.

We will review a few such scenarios here. BrownField has already taken a decision to use an external revenue management system in place of the PSS revenue management function. BrownField is also in the process of centralizing their accounting functions, and therefore, need not migrate the accounting function from the legacy system. Migration of CRM does not add much value at this point to the business. Therefore, it is decided to keep the CRM in the legacy system itself. The business has plans to move to a SaaS-based CRM solution as part of their cloud strategy. Also note that stalling the migration halfway through could seriously impact the complexity of the system.

Target architecture

The architecture blueprint shown in the following diagram consolidates earlier discussions into an architectural view. Each block in the diagram represents a microservice. The shaded boxes are core microservices, and the others are supporting microservices. The diagram also shows the internal capabilities of each microservice. User management is moved under security in the target architecture:

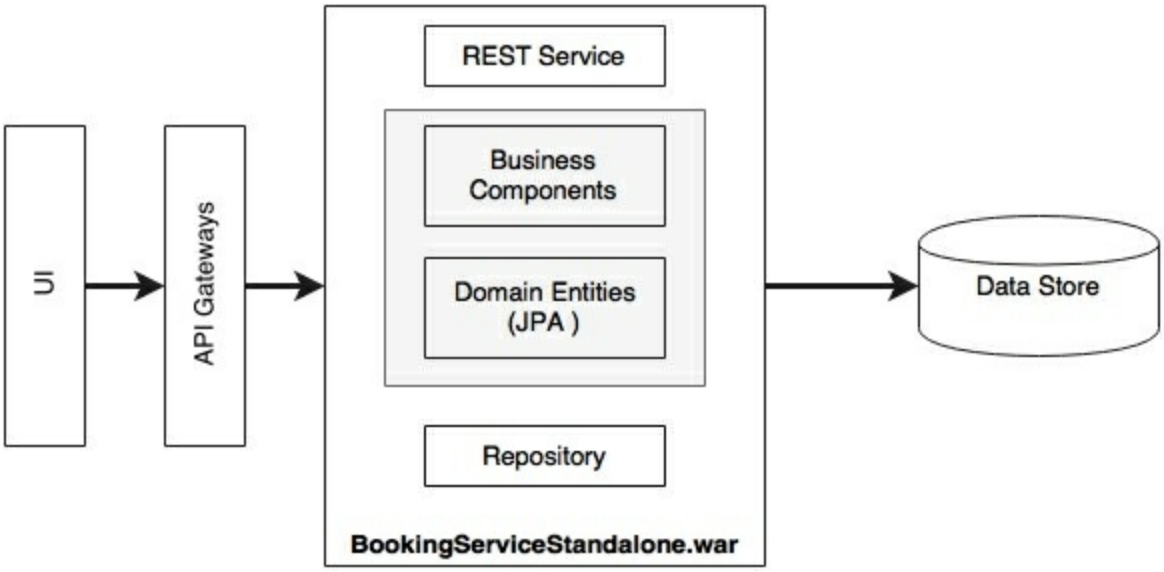

Presentation
REST / JSON
Search
Business Rules
Rules Store
Business Logic
Cassandra
Inventory + Fares Records
Search
Presentation
REST / JSON
JMS/JSON
Booking, Inventory
Business Rules
Rules Store
Business Logic
My SQL
Booking + Inventory Records
Reservation
Presentation
REST / JSON
JMS/JSON
CheckIn, Baggage, Seat
Business Rules
Rules Store
Business Logic
Hazelcast
Check-In Records
Check-In
Presentation
REST / JSON
City, Country, Airplane
Business Rules
Rules Store
Business Logic
Cassandra
Reference Data
Reference
Presentation
REST / JSON
Flights, Schedules, Routes
Business Rules
Rules Store
Business Logic
Cassandra
Flight Records
Flight
Presentation
REST / JSON
Fares
Business Rules
Rules Store
Business Logic
Cassandra
Fare Records
Fares
Presentation
REST / JSON
JMS/JSON
Serving Layer
PSS Domain Entities
Stream Analysis
Rules Store
Business Logic
HDFC
All Records
Data Analytics / Data Lake
Presentation
REST / JSON
JMS/JSON
Customer Profile
Business Rules
Rules Store
Business Logic
Cassandra
Customer Records
Customer
Presentation
REST / JSON
JMS/JSON
User, Roles, Privilliages
Business Rules
Rules Store
Business Logic
My SQL / LDAP
Security Records
Security
Presentation
REST / JSON
Notification, Channels, Templating
Business Rules
Rules Store
Business Logic
Mongo DB
Notification Records
Notification
Presentation
REST / JSON
Loyalty, Segmentation
Business Rules
Rules Store
Business Logic
Cassandra
Loyalty Records
Loyalty
Presentation
REST / JSON
Boarding
Business Rules
Rules Store
Business Logic
Hazelcast
Boading Records
Boarding

Each service has its own architecture, typically consisting of a presentation layer, one or more service endpoints, business logic, business rules, and database. As we can see, we use different selections of databases that are more suitable for each microservice. Each one is autonomous with minimal orchestration between the services. Most of the services interact with each other using the service endpoints.

Internal layering of microservices

In this section, we will further explore the internal structure of microservices. There is no standard to be followed for the internal architecture of a microservice. The rule of thumb is to abstract realizations behind simple service endpoints.

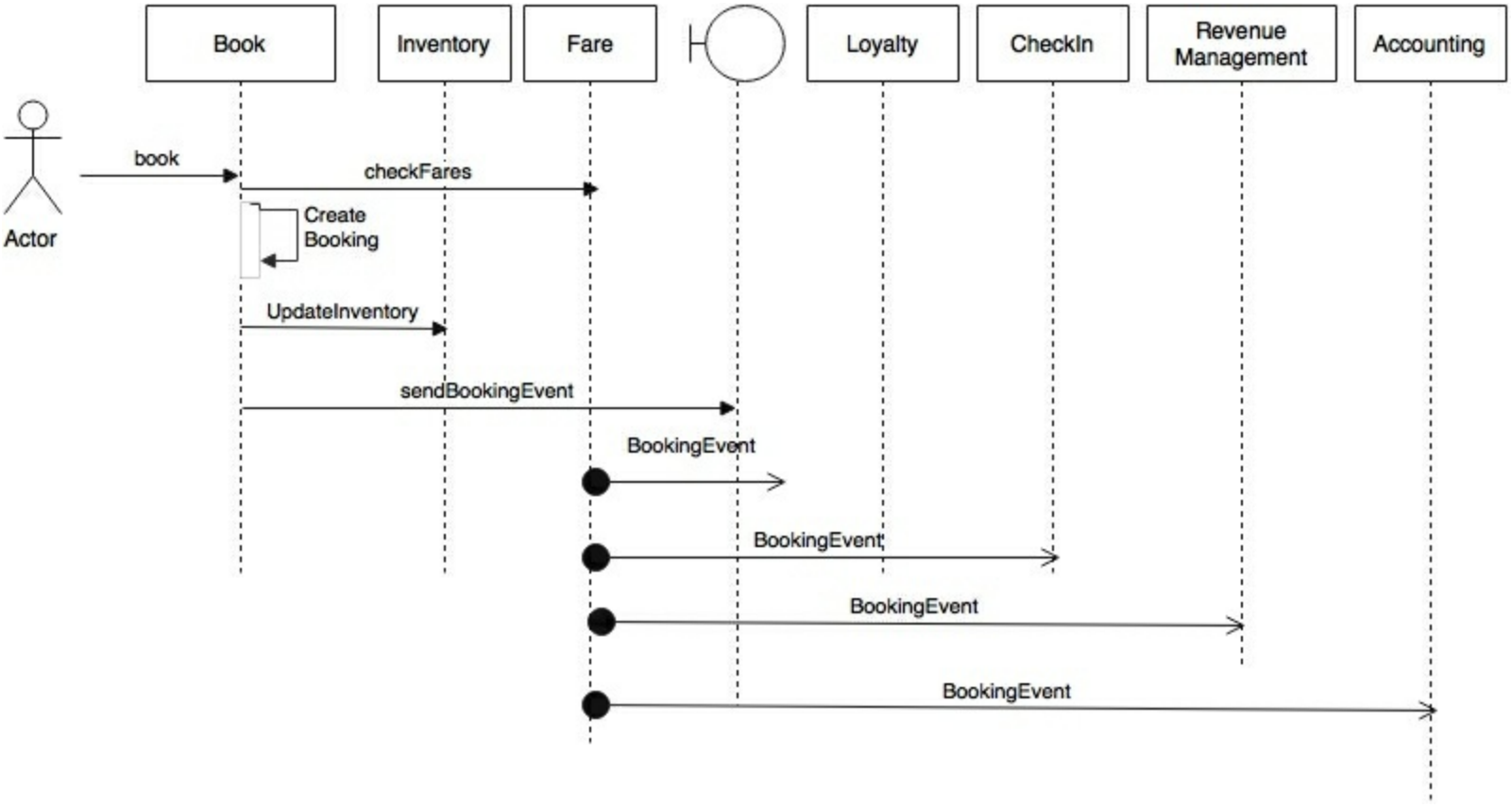
A typical structure would look like the one shown in the following diagram:



The UI accesses REST services through a service gateway. The API gateway may be one per microservice or one for many microservices—it depends on what we want to do with the API gateway. There could be one or more rest endpoints exposed by microservices. These endpoints, in turn, connect to one of the business components within the service. Business components then execute all the business functions with the help of domain entities. A repository component is used for interacting with the backend data store.

Orchestrating microservices

The logic of the booking orchestration and the execution of rules sits within the Booking service. The brain is still inside the Booking service in the form of one or more booking business components. Internally, business components orchestrate private APIs exposed by other business components or even external services:



As shown in the preceding diagram, the booking service internally calls to update the inventory of its own component other than calling the Fare service.

Is there any orchestration engine required for this activity? It depends on the requirements. In complex scenarios, we may have to do a number of things in parallel. For example, creating a booking internally applies a number of booking rules, it validates the fare, and it validates the inventory before creating a booking. We may want to execute them in parallel. In such cases, we may use Java concurrency APIs or reactive Java libraries.

In extremely complex situations, we may opt for an integration framework such as Spring Integration or Apache Camel in embedded mode.

Integration with other systems

In the microservices world, we use an API gateway or a reliable message bus for integrating with other non-microservices.

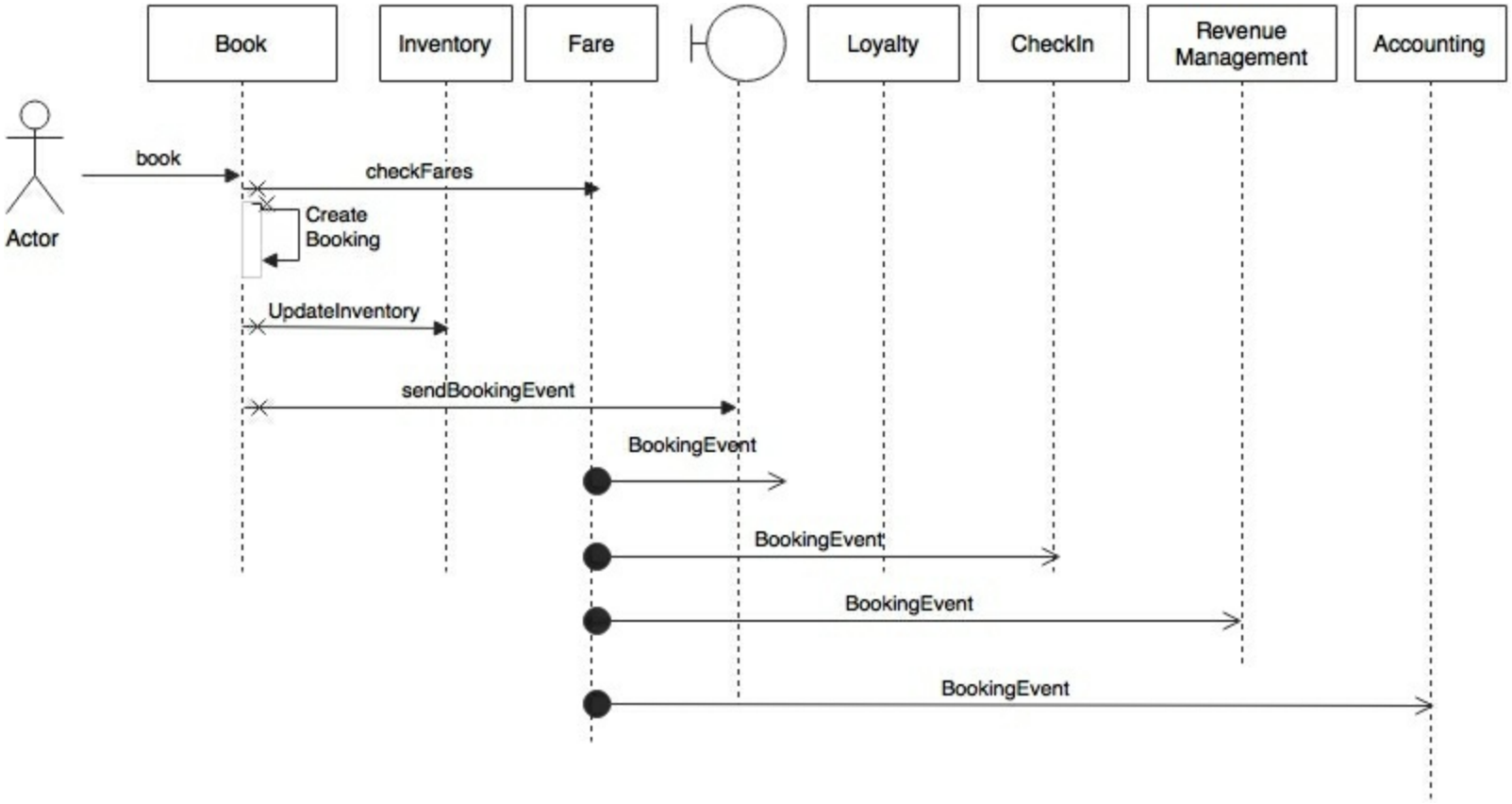
Let us assume that there is another system in BrownField that needs booking data. Unfortunately, the system is not capable of subscribing to the booking events that the Booking microservice publishes. In such cases, an **Enterprise Application integration** (**EAI**) solution could be employed, which listens to our booking events, and then uses a native adaptor to update the database.

Managing shared libraries

Certain business logic is used in more than one microservice. Search and Reservation, in this case, use inventory rules. In such cases, these shared libraries will be duplicated in both the microservices.

Handling exceptions

Examine the booking scenario to understand the different exception handling approaches. In the following service sequence diagram, there are three lines marked with a cross mark. These are the potential areas where exceptions could occur:

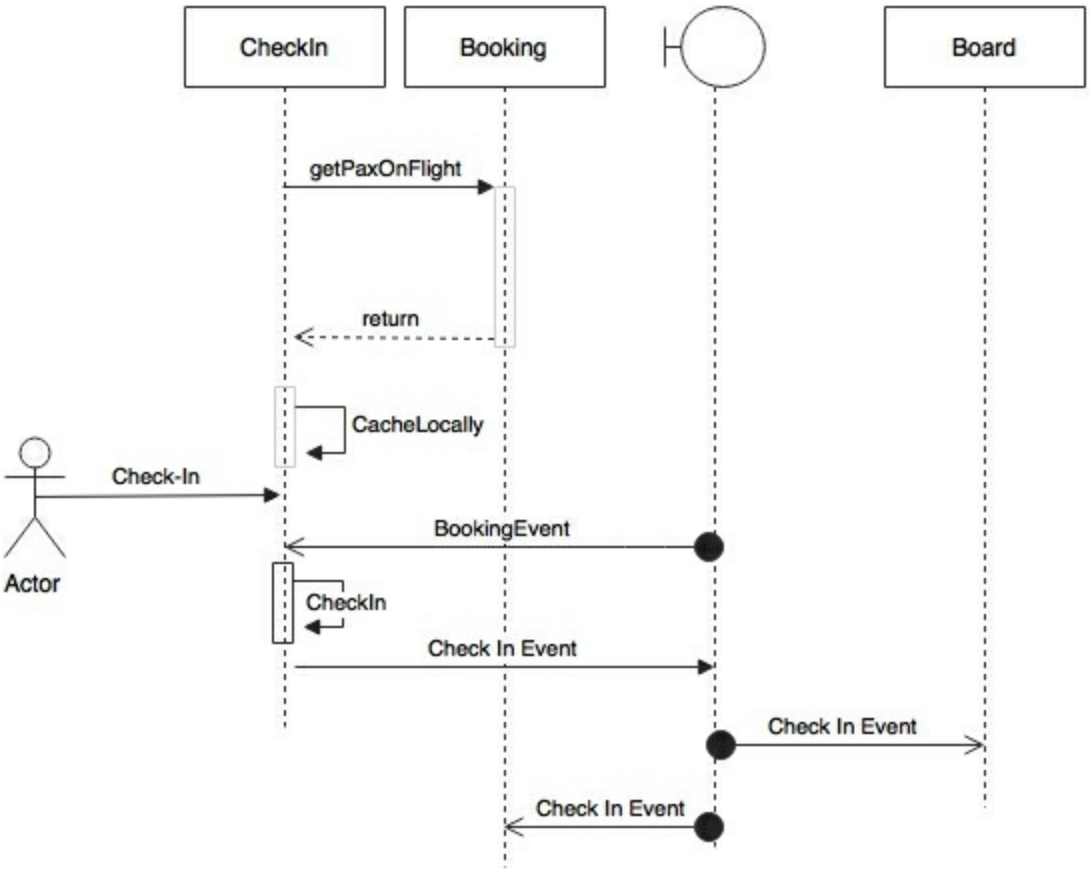


There is a synchronous communication between Booking and Fare. What if the Fare service is not available? If the Fare service is not available, throwing an error back to the user may cause revenue loss. An alternate thought is to trust the fare which comes as part of the incoming request. When we serve search, the search results will have the fare as well. When the user selects a flight and submits, the request will have the selected fare. In case the Fare service is not available, we trust the incoming request, and accept the Booking. We will use a circuit breaker and a fallback service which simply creates the booking with a special status, and queues the booking for manual action or a system retry.

What if creating the booking fails? If creating a booking fails unexpectedly, a better option is to throw a message back to the user. We could try alternative options, but that could increase the overall complexity of the system. The same is applicable for inventory updates.

In the case of creating a booking and updating the inventory, we avoid a situation where a booking is created, and an inventory update somehow fails. As the inventory is critical, it is better to have both, create booking and update inventory, to be in a local transaction. This is possible as both components are under the same subsystem.

If we consider the Check-in scenario, Check-in sends an event to Boarding and Booking as shown in the next diagram:



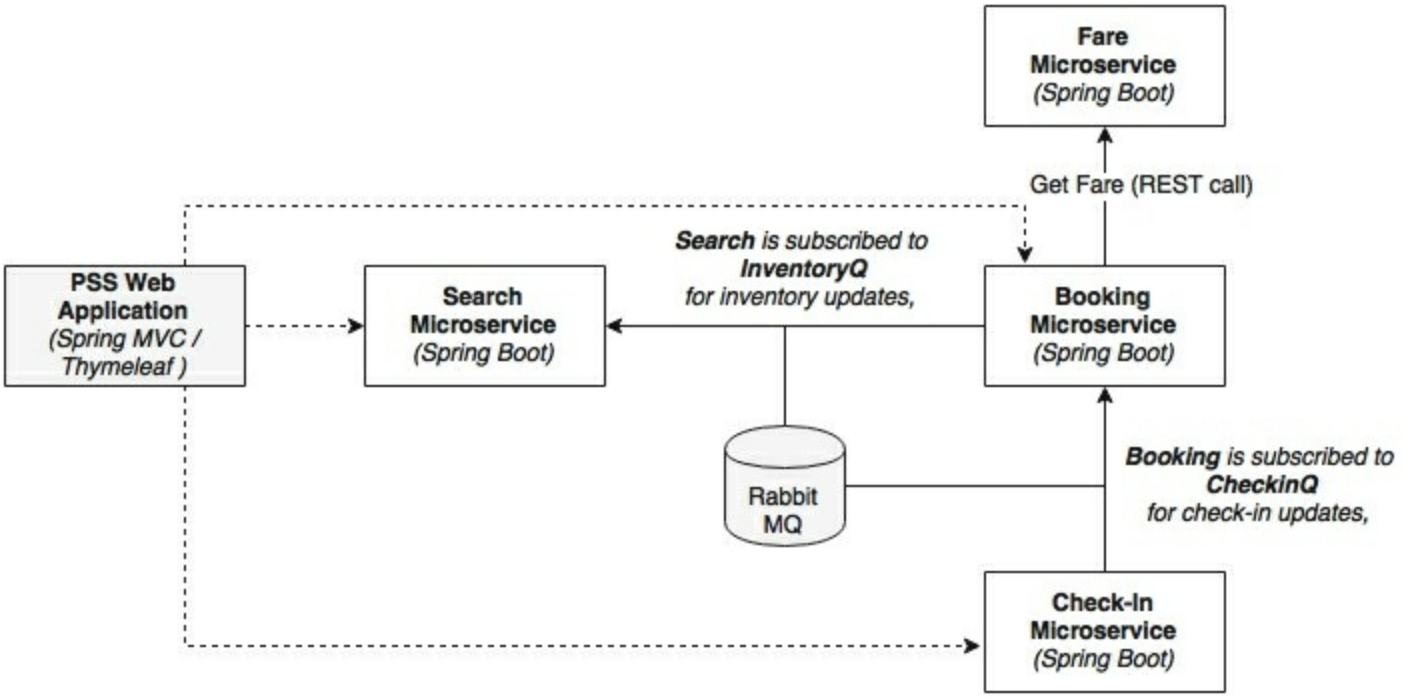
Consider a scenario where the Check-in services fail immediately after the Check-in Complete event is sent out. The other consumers processed this event, but the actual check-in is rolled back. This is because we are not using a two-phase commit. In this case, we need a mechanism for reverting that event. This could be done by catching the exception, and sending another Check-in Cancelled event.

In this case, note that to minimize the use of compensating transactions, sending the Check-in event is moved towards the end of the Check-in transaction. This reduces the chance of failure after sending out the event.

On the other hand, what if the check-in is successful, but sending the event failed? We could think of two approaches. The first approach would be to invoke a fallback service to store it locally, and then use another sweep-and-scan program to send the event at a later time. It could even retry multiple times. This could add more complexity and may not be efficient in all cases. An alternate approach is to throw the exception back to the user so that the user can retry. However, this might not always be good from a customer engagement standpoint. On the other hand, the earlier option is better for the system's health. A trade-off analysis is required to find out the best solution for the given situation.

Target implementation view

The next diagram represents the implementation view of the BrownField PSS microservices system:



As shown in the preceding diagram, we are implementing four microservices as an example: Search, Fare, Booking, and Check-in. In order to test the application, there is a website application developed using Spring MVC with Thymeleaf templates. The asynchronous messaging is implemented with the help of RabbitMQ. In this sample implementation, the default H2 database is used as the in-memory store for demonstration purposes.

The code in this section demonstrates all the capabilities highlighted in the *Reviewing the microservices capability model* section of this chapter.

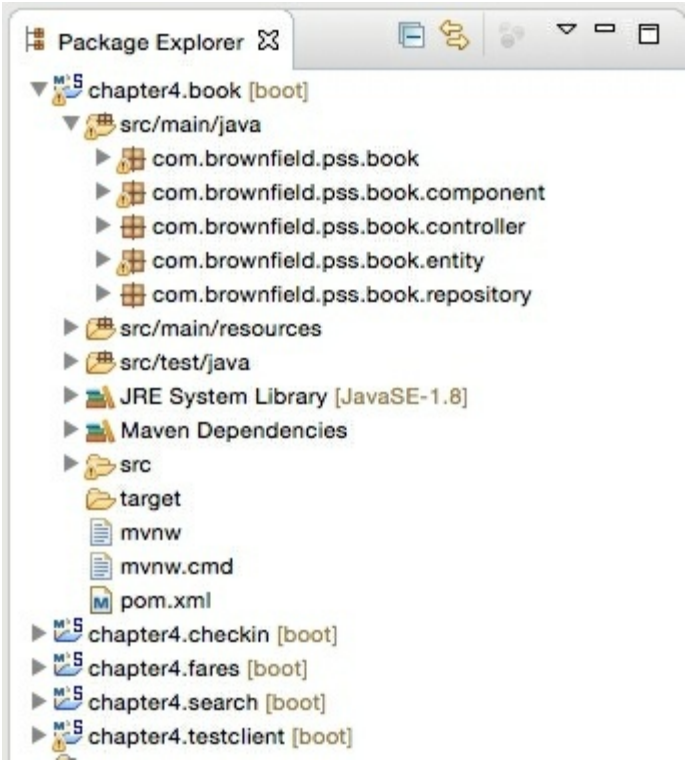
Implementation projects

The basic implementation of the BrownField Airline's PSS microservices system has five core projects as summarized in the following table. The table also shows the port range used for these projects to ensure consistency throughout the book:

Microservice	Projects	Port Range
Book microservice	`chapter4.book`	`8060-8069`
Check-in microservice	`chapter4.checkin`	`8070-8079`
Fare microservice	`chapter4.fares`	`8080-8089`
Search microservice	`chapter4.search`	`8090-8099`
Website	`chapter4.website`	`8001`

The website is the UI application for testing the PSS microservices.

All microservice projects in this example follow the same pattern for package structure as shown in the following screenshot:



The different packages and their purposes are explained as follows:

- The root folder (`com.brownfield.pss.book`) contains the default Spring Boot application.
- The `component` package hosts all the service components where the business logic is implemented.
- The `controller` package hosts the REST endpoints and the messaging endpoints. Controller classes internally utilize the component classes for execution.
- The `entity` package contains the JPA entity classes for mapping to the database tables.
- Repository classes are packaged inside the `repository` package, and are based on Spring Data JPA.

Running and testing the project

Follow the steps listed next to build and test the microservices developed in this chapter:

1. Build each of the projects using Maven. Ensure that the `test` flag is switched off. The test programs assume other dependent services are up and running. It fails if the dependent services are not available. In our example, Booking and Fare have direct dependencies. We will learn how to circumvent this dependency in Chapter 7, *Logging and Monitoring Microservices*:

   ```
   mvn -Dmaven.test.skip=true install
   ```

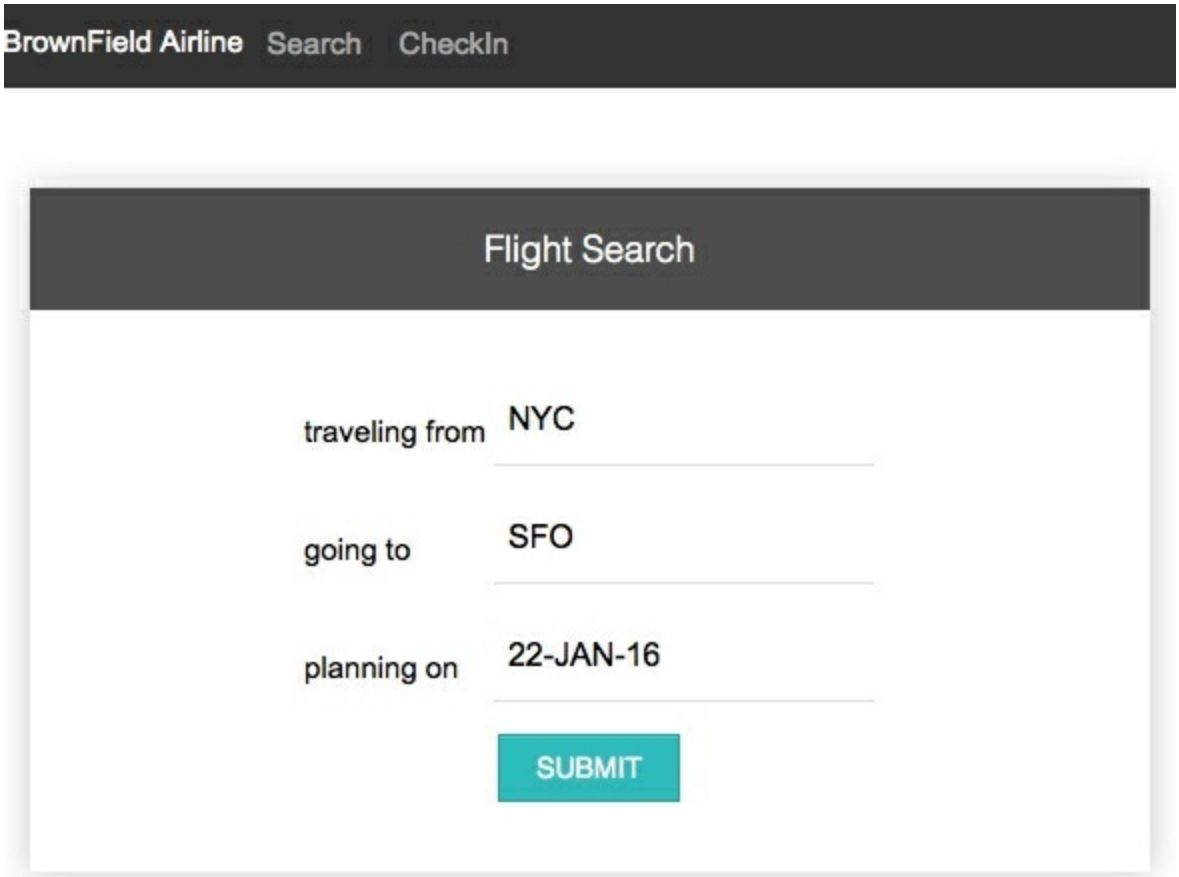
2. Run the RabbitMQ server:

   ```
   rabbitmq_server-3.5.6/sbin$ ./rabbitmq-server
   ```

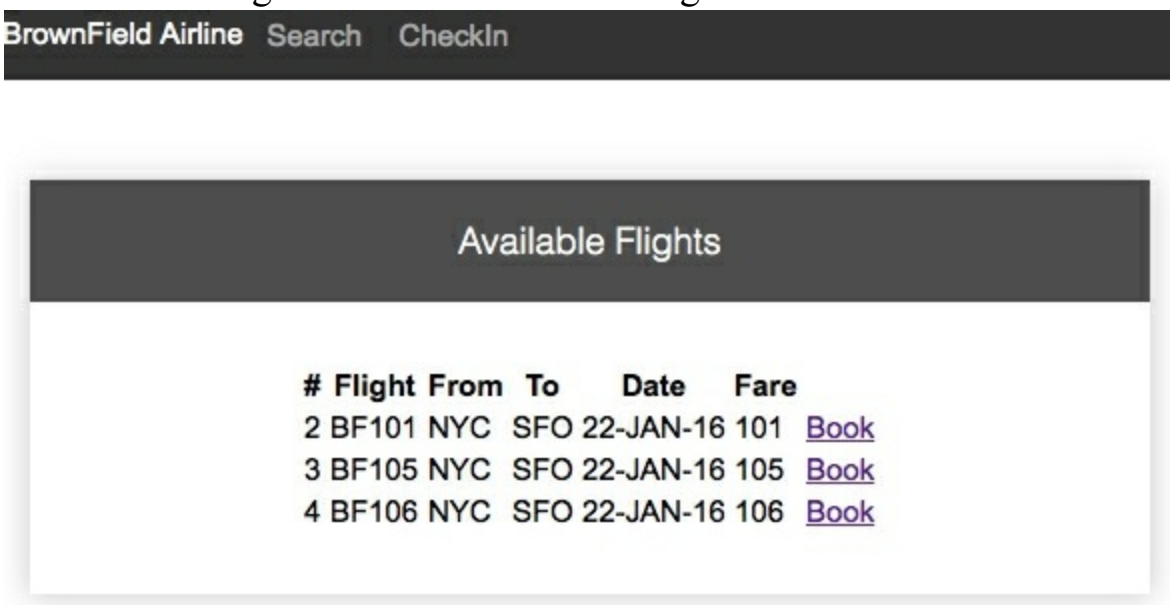
3. Run the following commands in separate terminal windows:

   ```
   java -jar target/fares-1.0.jar
   java -jar target/search-1.0.jar
   java -jar target/checkin-1.0.jar
   java -jar target/book-1.0.jar
   java -jar target/website-1.0.jar
   ```

4. The website project has a `CommandLineRunner`, which executes all the test cases at startup. Once all the services are successfully started, open `http://localhost:8001` in a browser.
5. The browser asks for basic security credentials. Use `guest` or `guest123` as the credentials. This example only shows the website security with a basic authentication mechanism. As explained in Chapter 2, *Building Microservices with Spring Boot*, service-level security can be achieved using OAuth2.
6. Entering the correct security credentials displays the following screen. This is the home screen of our BrownField PSS application:

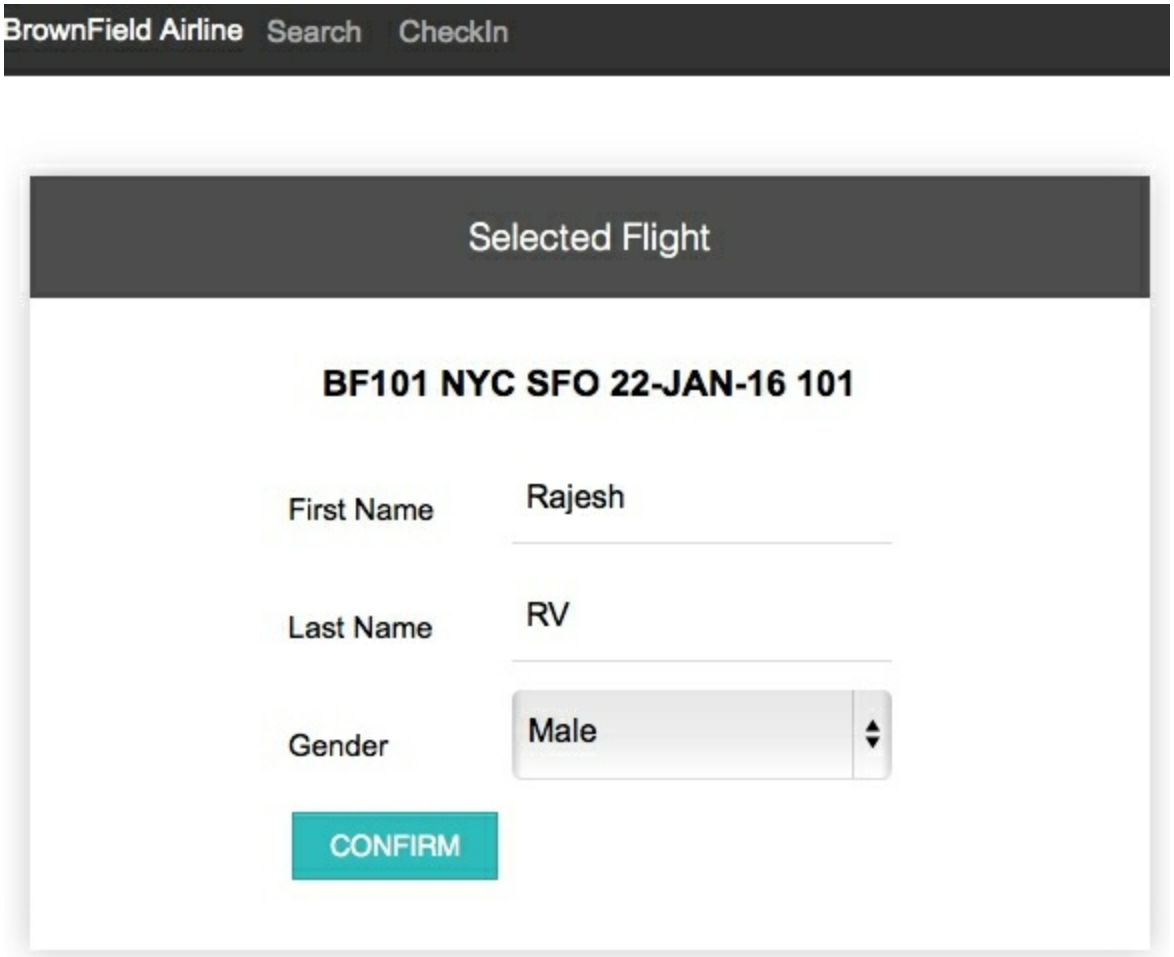


7. The **SUBMIT** button invokes the Search microservice to fetch the available flights that meet the conditions mentioned on the screen. A few flights are pre-populated at the startup of the Search microservice. Edit the Search microservice code to feed in additional flights, if required.
8. The output screen with a list of flights is shown in the next screenshot. The **Book** link will take us to the booking screen for the selected flight:

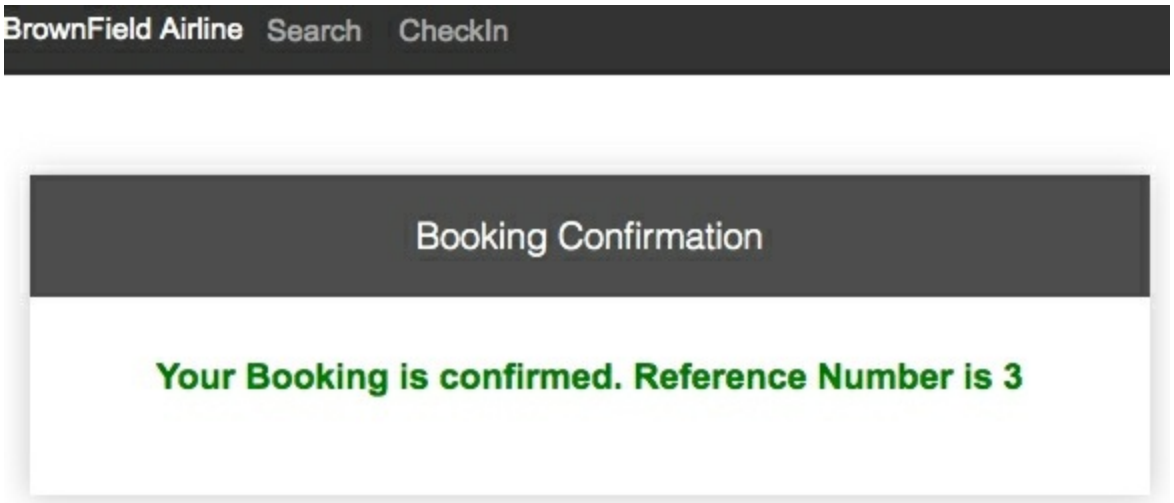


9. The following screenshot shows the booking screen. The user can enter the passenger details,

and create a booking by clicking on the **CONFIRM** button. This invokes the Booking microservice, and internally, the Fare service as well. It also sends a message back to the Search microservice:

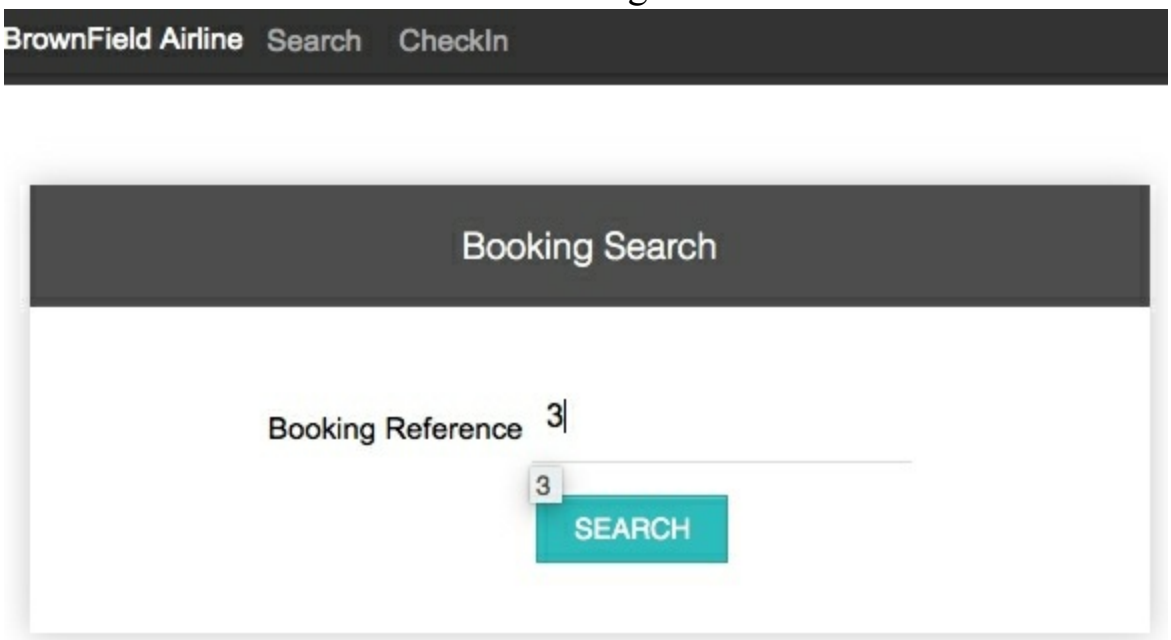


10. If booking is successful, the next confirmation screen is displayed with a booking reference number:

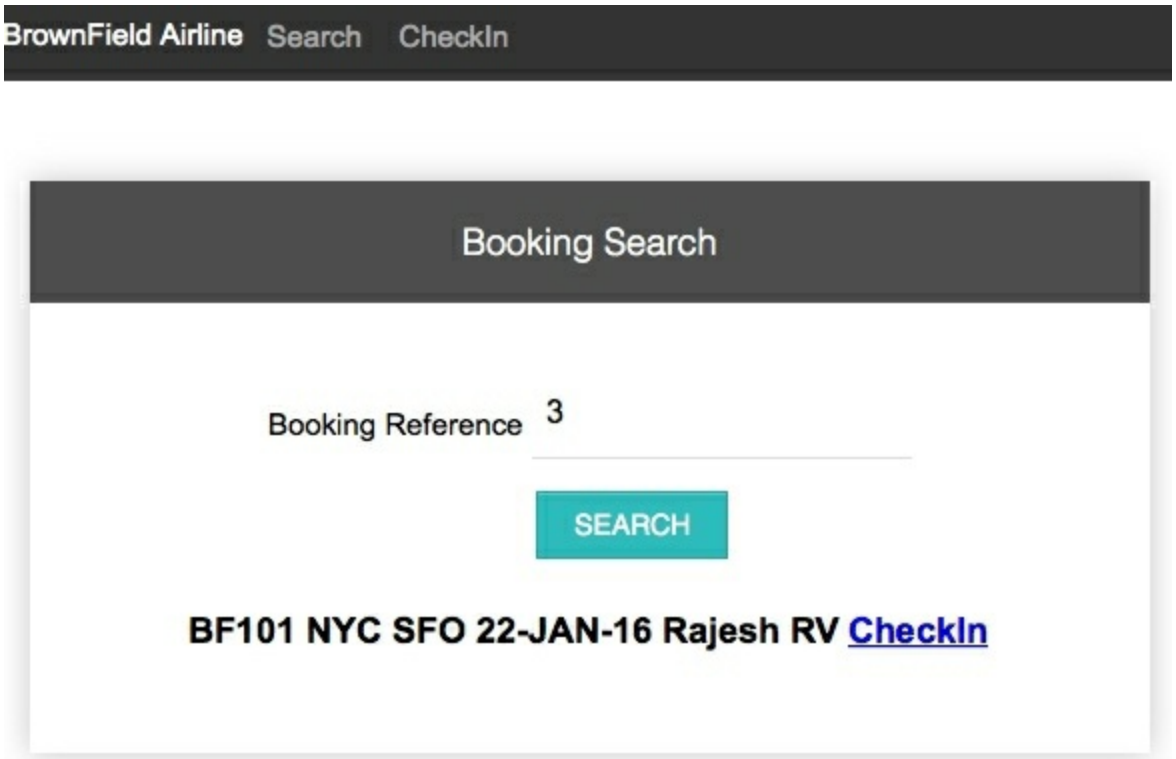


11. Let us test the Check-in microservice. This can be done by clicking on **CheckIn** in the menu at

the top of the screen. Use the booking reference number obtained in the previous step to test Check-in. This is shown in the following screenshot:

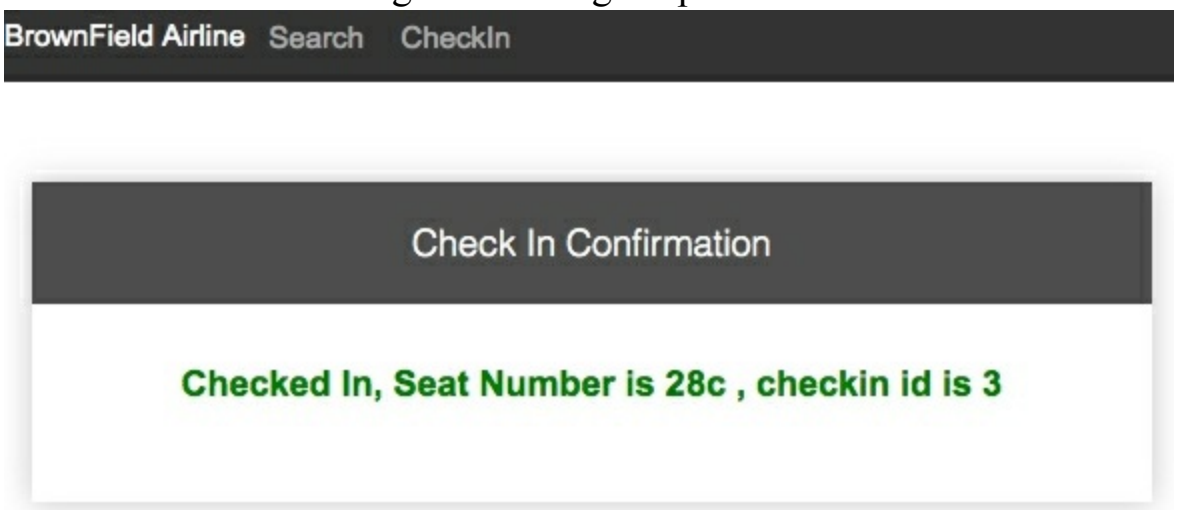


12. Clicking on the **SEARCH** button in the previous screen invokes the Booking microservice, and retrieves the booking information. Click on the **CheckIn** link to perform the check-in. This invokes the Check-in microservice:



13. If check-in is successful, it displays the confirmation message, as shown in the next screenshot, with a confirmation number. This is done by calling the Check-in service internally. The Check-

in service sends a message to Booking to update the check-in status:



Summary

In this chapter, we implemented and tested the BrownField PSS microservice with basic Spring Boot capabilities. We learned how to approach a real use case with a microservices architecture.

We examined the various stages of a real-world evolution towards microservices from a monolithic application. We also evaluated the pros and cons of multiple approaches, and the obstacles encountered when migrating a monolithic application. Finally, we explained the end-to-end microservices design for the use case that we examined. Design and implementation of a fully-fledged microservice implementation was also validated.

In the next chapter, we will see how the Spring Cloud project helps us to transform the developed BrownField PSS microservices to an Internet-scale deployment.

Chapter 5. Scaling Microservices with Spring Cloud

In order to manage Internet-scale microservices, one requires more capabilities than what are offered by the Spring Boot framework. The Spring Cloud project has a suite of purpose-built components to achieve these additional capabilities effortlessly.

This chapter will provide a deep insight into the various components of the Spring Cloud project such as Eureka, Zuul, Ribbon, and Spring Config by positioning them against the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*. It will demonstrate how the Spring Cloud components help to scale the BrownField Airline's PSS microservices system, developed in the previous chapter.

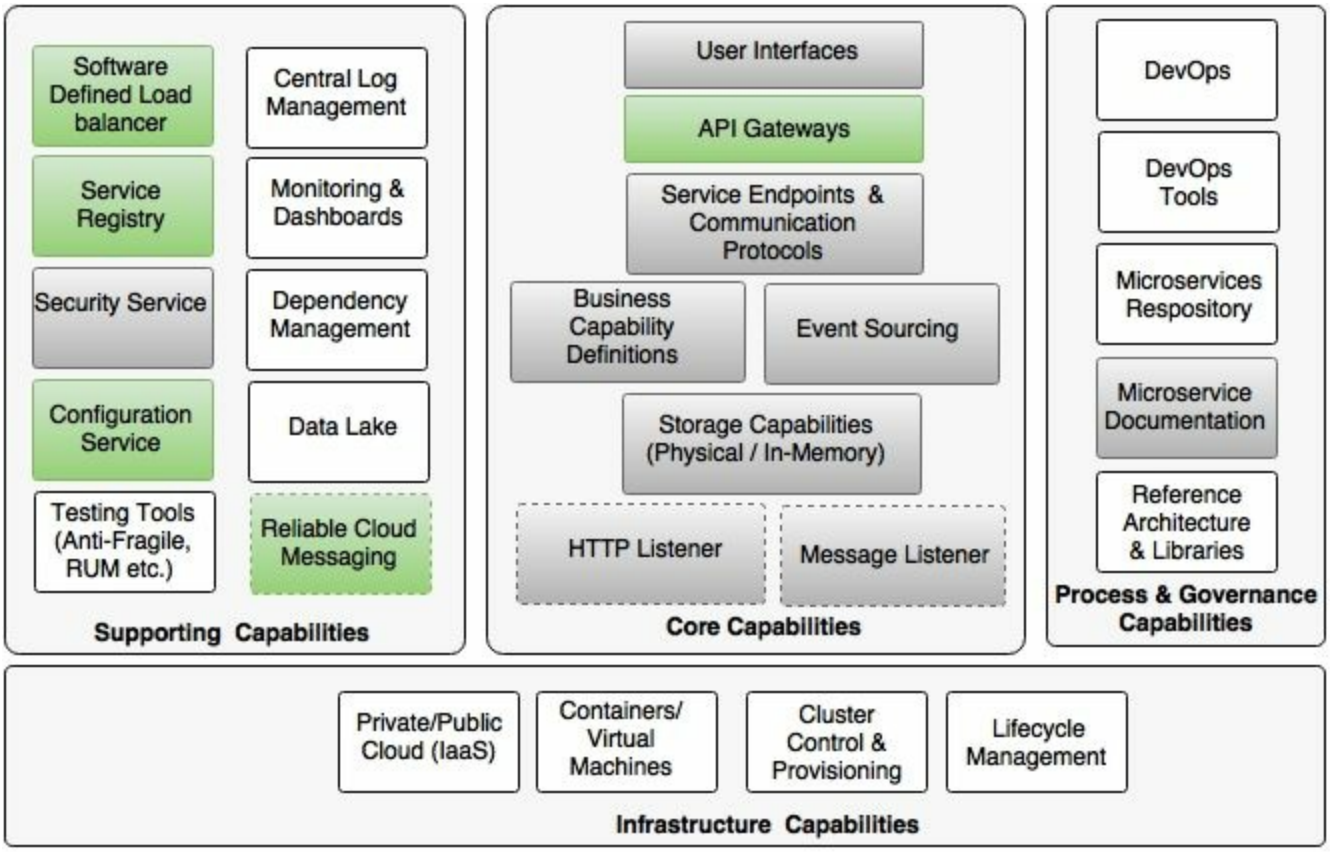
By the end of this chapter, you will learn about the following:

- The Spring Config server for externalizing configuration
- The Eureka server for service registration and discovery
- The relevance of Zuul as a service proxy and gateway
- The implementation of automatic microservice registration and service discovery
- Spring Cloud messaging for asynchronous microservice composition

Reviewing microservices capabilities

The examples in this chapter explore the following microservices capabilities from the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*:

- **Software Defined Load Balancer**
- **Service Registry**
- **Configuration Service**
- **Reliable Cloud Messaging**
- **API Gateways**



Reviewing BrownField's PSS implementation

In Chapter 4, *Microservices Evolution – A Case Study*, we designed and developed a microservice-based PSS system for BrownField Airlines using the Spring framework and Spring Boot. The implementation is satisfactory from the development point of view, and it serves the purpose for low volume transactions. However, this is not good enough for deploying large, enterprise-scale deployments with hundreds or even thousands of microservices.

In Chapter 4, *Microservices Evolution – A Case Study*, we developed four microservices: Search, Booking, Fares, and Check-in. We also developed a website to test the microservices.

We have accomplished the following items in our microservice implementation so far:

- Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
- Each microservice implements certain business functions using the Spring framework.
- Each microservice stores its own persistent data using H2, an in-memory database
- Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener
- RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging
- Swagger is integrated with all microservices for documenting the REST APIs.
- An OAuth2-based security mechanism is developed to protect the microservices

What is Spring Cloud?

The Spring Cloud project is an umbrella project from the Spring team that implements a set of common patterns required by distributed systems, as a set of easy-to-use Java Spring libraries. Despite its name, Spring Cloud by itself is not a cloud solution. Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that adhere to the Twelve-Factor application principles. By using Spring Cloud, developers just need to focus on building business capabilities using Spring Boot, and leverage the distributed, fault-tolerant, and self-healing capabilities available out of the box from Spring Cloud.

The Spring Cloud solutions are agnostic to the deployment environment, and can be developed and deployed in a desktop PC or in an elastic cloud. The cloud-ready solutions that are developed using Spring Cloud are also agnostic and portable across many cloud providers such as Cloud Foundry, AWS, Heroku, and so on. When not using Spring Cloud, developers will end up using services natively provided by the cloud vendors, resulting in deep coupling with the PaaS providers. An alternate option for developers is to write quite a lot of boilerplate code to build these services. Spring Cloud also provides simple, easy-to-use Spring-friendly APIs, which abstract the cloud provider's service APIs such as those APIs coming with AWS Notification Service.

Built on Spring's "convention over configuration" approach, Spring Cloud defaults all configurations, and helps the developers get off to a quick start. Spring Cloud also hides the complexities, and provides simple declarative configurations to build systems. The smaller footprints of the Spring Cloud components make it developer friendly, and also make it easy to develop cloud-native applications.

Spring Cloud offers many choices of solutions for developers based on their requirements. For example, the service registry can be implemented using popular options such as Eureka, ZooKeeper, or Consul. The components of Spring Cloud are fairly decoupled, hence, developers get the flexibility to pick and choose what is required.

Note

What is the difference between Spring Cloud and Cloud Foundry?

Spring Cloud is a developer kit for developing Internet-scale Spring Boot applications, whereas Cloud Foundry is an open-source Platform as a Service for building, deploying, and scaling applications.

Spring Cloud releases

The Spring Cloud project is an overarching Spring project that includes a combination of different components. The versions of these components are defined in the `spring-cloud-starter-parent` BOM.

In this book, we are relying on the `Brixton.RELEASE` version of the Spring Cloud:

```
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>Brixton.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
  </dependency>
```

The `spring-cloud-starter-parent` defines different versions of its subcomponents as follows:

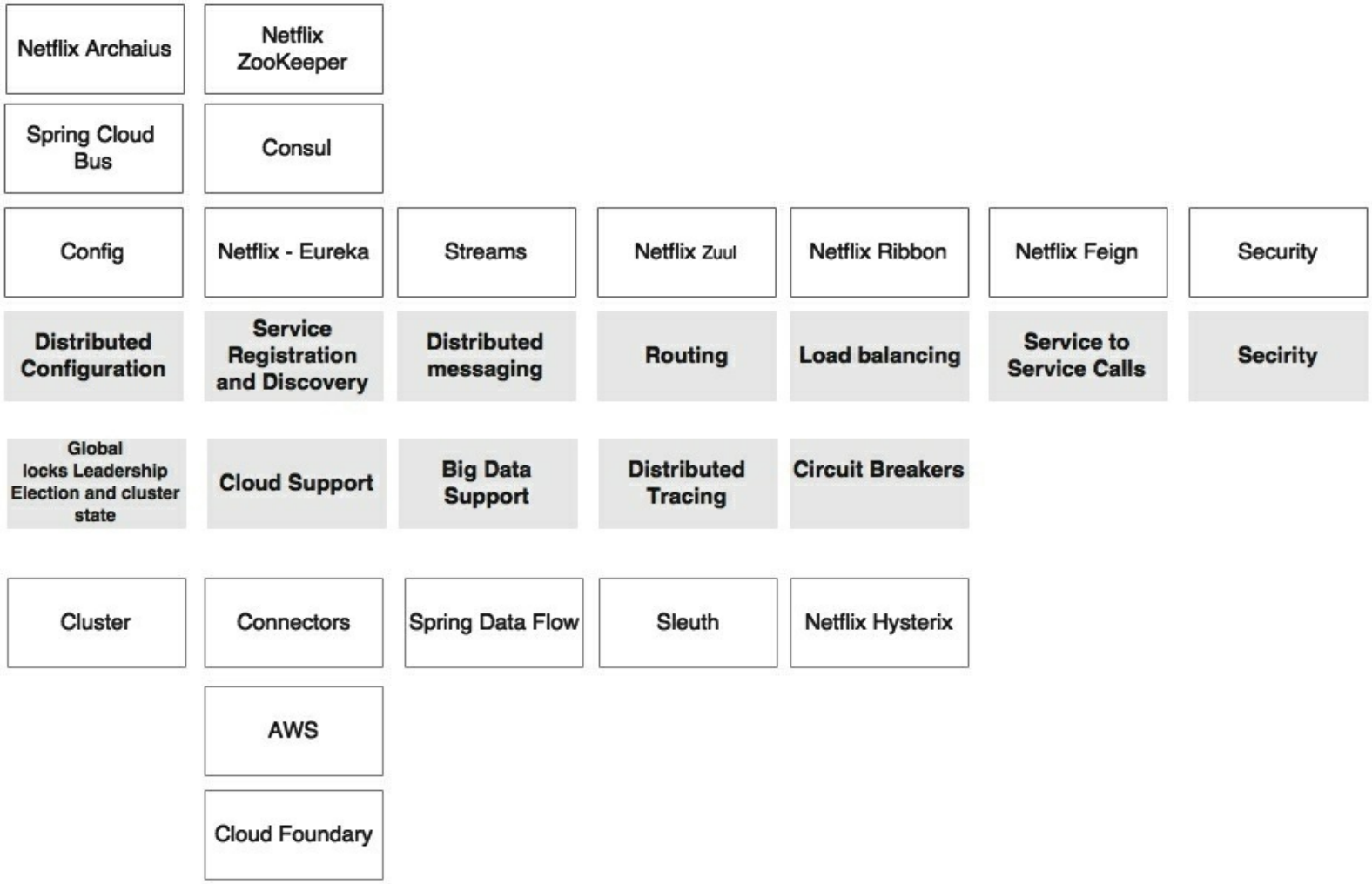
```
<spring-cloud-aws.version>1.1.0.RELEASE</spring-cloud-aws.version>
<spring-cloud-bus.version>1.1.0.RELEASE</spring-cloud-bus.version>
<spring-cloud-cloudfoundry.version>1.0.0.RELEASE</spring-cloud-
cloudfoundry.version>
<spring-cloud-commons.version>1.1.0.RELEASE</spring-cloud-commons.version>
<spring-cloud-config.version>1.1.0.RELEASE</spring-cloud-config.version>
<spring-cloud-netflix.version>1.1.0.RELEASE</spring-cloud-netflix.version>
<spring-cloud-security.version>1.1.0.RELEASE</spring-cloud-security.version>
<spring-cloud-cluster.version>1.0.0.RELEASE</spring-cloud-cluster.version>
<spring-cloud-consul.version>1.0.0.RELEASE</spring-cloud-consul.version>
<spring-cloud-sleuth.version>1.0.0.RELEASE</spring-cloud-sleuth.version>
<spring-cloud-stream.version>1.0.0.RELEASE</spring-cloud-stream.version>
<spring-cloud-zookeeper.version>1.0.0.RELEASE </spring-cloud-zookeeper.version>
```

Note

The names of the Spring Cloud releases are in an alphabetic sequence, starting with A, following the names of the London Tube stations. **Angel** was the first release, and **Brixton** is the second release.

Components of Spring Cloud

Each Spring Cloud component specifically addresses certain distributed system capabilities. The grayed-out boxes at the bottom of the following diagram show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud subprojects addressing these capabilities:



The Spring Cloud capabilities are explained as follows:

- **Distributed configuration**: Configuration properties are hard to manage when there are many microservice instances running under different profiles such as development, test, production, and so on. It is, therefore, important to manage them centrally, in a controlled way. The distributed configuration management module is to externalize and centralize microservice configuration parameters. Spring Cloud Config is an externalized configuration server with Git or SVN as the backing repository. Spring Cloud Bus provides support for propagating configuration changes to multiple subscribers, generally a microservice instance. Alternately, ZooKeeper or HashiCorp's Consul can also be used for distributed configuration management.
- **Routing**: Routing is an API gateway component, primarily used similar to a reverse proxy that forwards requests from consumers to service providers. The gateway component can also perform software-based routing and filtering. Zuul is a lightweight API gateway solution that

offers fine-grained controls to developers for traffic shaping and request/response transformations.

- **Load balancing**: The load balancer capability requires a software-defined load balancer module which can route requests to available servers using a variety of load balancing algorithms. Ribbon is a Spring Cloud subproject which supports this capability. Ribbon can work as a standalone component, or integrate and work seamlessly with Zuul for traffic routing.
- **Service registration and discovery**: The service registration and discovery module enables services to programmatically register with a repository when a service is available and ready to accept traffic. The microservices advertise their existence, and make them discoverable. The consumers can then look up the registry to get a view of the service availability and the endpoint locations. The registry, in many cases, is more or less a dump. But the components around the registry make the ecosystem intelligent. There are many subprojects existing under Spring Cloud which support registry and discovery capability. Eureka, ZooKeeper, and Consul are three subprojects implementing the registry capability.
- **Service-to-service calls**: The Spring Cloud Feign subproject under Spring Cloud offers a declarative approach for making RESTful service-to-service calls in a synchronous way. The declarative approach allows applications to work with **POJO** (**Plain Old Java Object**) interfaces instead of low-level HTTP client APIs. Feign internally uses reactive libraries for communication.
- **Circuit breaker**: The circuit breaker subproject implements the circuit breaker pattern. The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service. It also automatically reconnects back to the primary service when the service is back to normal. It finally provides a monitoring dashboard for monitoring the service state changes. The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard respectively.
- **Global locks, leadership election and cluster state**: This capability is required for cluster management and coordination when dealing with large deployments. It also offers global locks for various purposes such as sequence generation. The Spring Cloud Cluster project implements these capabilities using Redis, ZooKeeper, and Consul.
- **Security**: Security capability is required for building security for cloud-native distributed systems using externalized authorization providers such as OAuth2. The Spring Cloud Security project implements this capability using customizable authorization and resource servers. It also offers SSO capabilities, which are essential when dealing with many microservices.
- **Big data support**: The big data support capability is a capability that is required for data services and data flows in connection with big data solutions. The Spring Cloud Streams and the Spring Cloud Data Flow projects implement these capabilities. The Spring Cloud Data Flow is the re-engineered version of Spring XD.
- **Distributed tracing**: The distributed tracing capability helps to thread and correlate transitions that are spanned across multiple microservice instances. Spring Cloud Sleuth implements this by providing an abstraction on top of various distributed tracing mechanisms such as Zipkin and HTrace with the support of a 64-bit ID.
- **Distributed messaging**: Spring Cloud Stream provides declarative messaging integration on top of reliable messaging solutions such as Kafka, Redis, and RabbitMQ.

- **Cloud support**: Spring Cloud also provides a set of capabilities that offers various connectors, integration mechanisms, and abstraction on top of different cloud providers such as the Cloud Foundry and AWS.

Spring Cloud and Netflix OSS

Many of the Spring Cloud components which are critical for microservices' deployment came from the **Netflix Open Source Software** (**Netflix OSS**) center. Netflix is one of the pioneers and early adaptors in the microservices space. In order to manage large scale microservices, engineers at Netflix came up with a number of homegrown tools and techniques for managing their microservices. These are fundamentally crafted to fill some of the software gaps recognized in the AWS platform for managing Netflix services. Later, Netflix open-sourced these components, and made them available under the Netflix OSS platform for public use. These components are extensively used in production systems, and are battle-tested with large scale microservice deployments at Netflix.

Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism, well-integrated and aligned with Spring Boot and the Spring framework.

Setting up the environment for BrownField PSS

In this chapter, we will amend the BrownField PSS microservices developed in Chapter 4, *Microservices Evolution – A Case Study*, using Spring Cloud capabilities. We will also examine how to make these services enterprise grade using Spring Cloud components.

Subsequent sections of this chapter will explore how to scale the microservices developed in the previous chapter for cloud scale deployments, using some out-of-the-box capabilities provided by the Spring Cloud project. The rest of this chapter will explore Spring Cloud capabilities such as configuration using the Spring Config server, Ribbon-based service load balancing, service discovery using Eureka, Zuul for API gateway, and finally, Spring Cloud messaging for message-based service interactions. We will demonstrate the capabilities by modifying the BrownField PSS microservices developed in Chapter 4, *Microservices Evolution – A Case Study*.

In order to prepare the environment for this chapter, import and rename (`chapter4.*` to `chapter5.*`) projects into a new STS workspace.

Note

The full source code of this chapter is available under the `Chapter 5` projects in the code files.

Spring Cloud Config

The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties. The Spring Config server also supports version control of the configuration properties.

In the earlier examples with Spring Boot, all configuration parameters were read from a property file packaged inside the project, either `application.properties` or `application.yaml`. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which advocate one-time build and moving of the binaries across environments.

A better approach is to use the concept of profiles. Profiles, as discussed in Chapter 2, *Building Microservices with Spring Boot*, is used for partitioning different properties for different environments. The profile-specific configuration will be named `application-{profile}.properties`. For example, `application-development.properties` represents a property file targeted for the development environment.

However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.

There are alternate ways to externalize the configuration properties from the application deployment package. Configurable properties can also be read from an external source in a number of ways:

- From an external JNDI server using JNDI namespace (`java:comp/env`)
- Using the Java system properties (`System.getProperties()`) or using the `-D` command line option
- Using the `PropertySource` configuration:

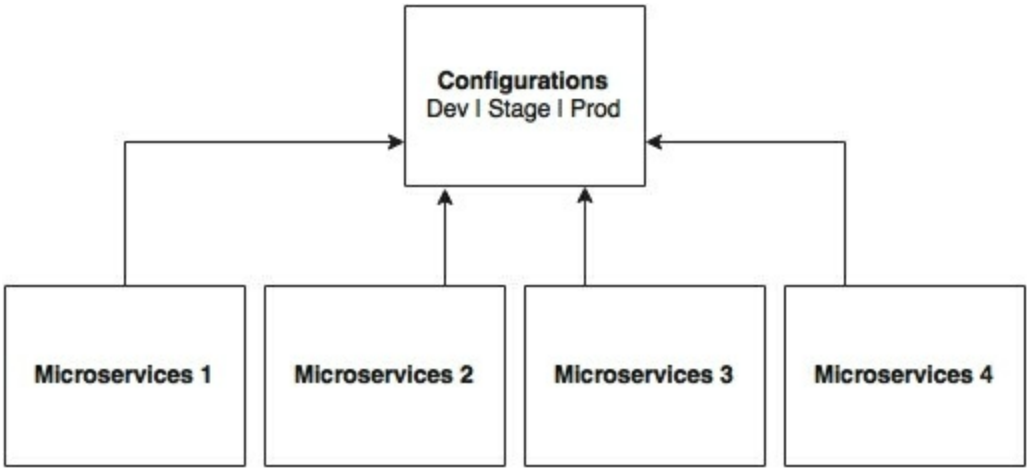
```
@PropertySource("file:${CONF_DIR}/application.properties")
  public class ApplicationConfig {
}
```

- Using a command-line parameter pointing a file to an external location:

```
java -jar myproject.jar --spring.config.location=
```

JNDI operations are expensive, lack flexibility, have difficulties in replication, and are not version controlled. `System.properties` is not flexible enough for large-scale deployments. The last two options rely on a local or a shared filesystem mounted on the server.

For large scale deployments, a simple yet powerful centralized configuration management solution is required:



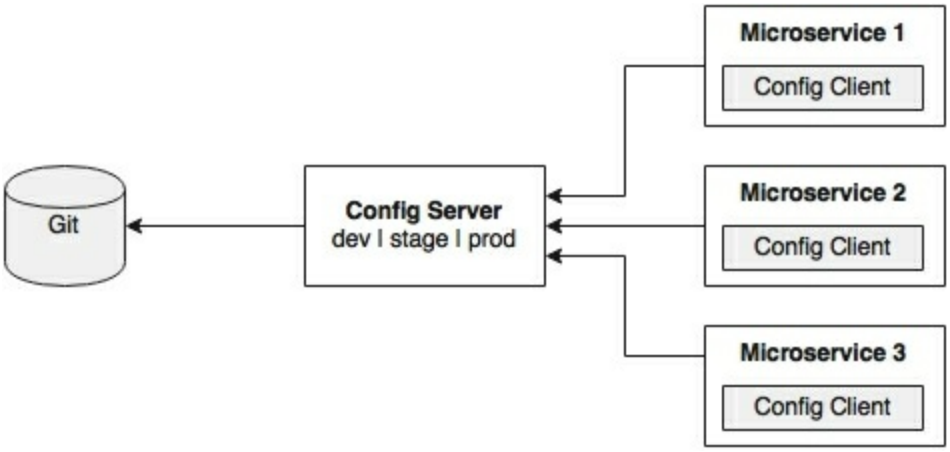
As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. The microservices then locally cache these parameters to improve performance. The Config server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes. The Config server also uses profiles to resolve values specific to an environment.

As shown in the following screenshot, there are multiple options available under the Spring Cloud project for building the configuration server. **Config Server**, **Zookeeper Configuration**, and **Consul Configuration** are available as options. However, this chapter will only focus on the Spring Config server implementation:



The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.

The Spring Cloud Config server architecture is shown in the following diagram:

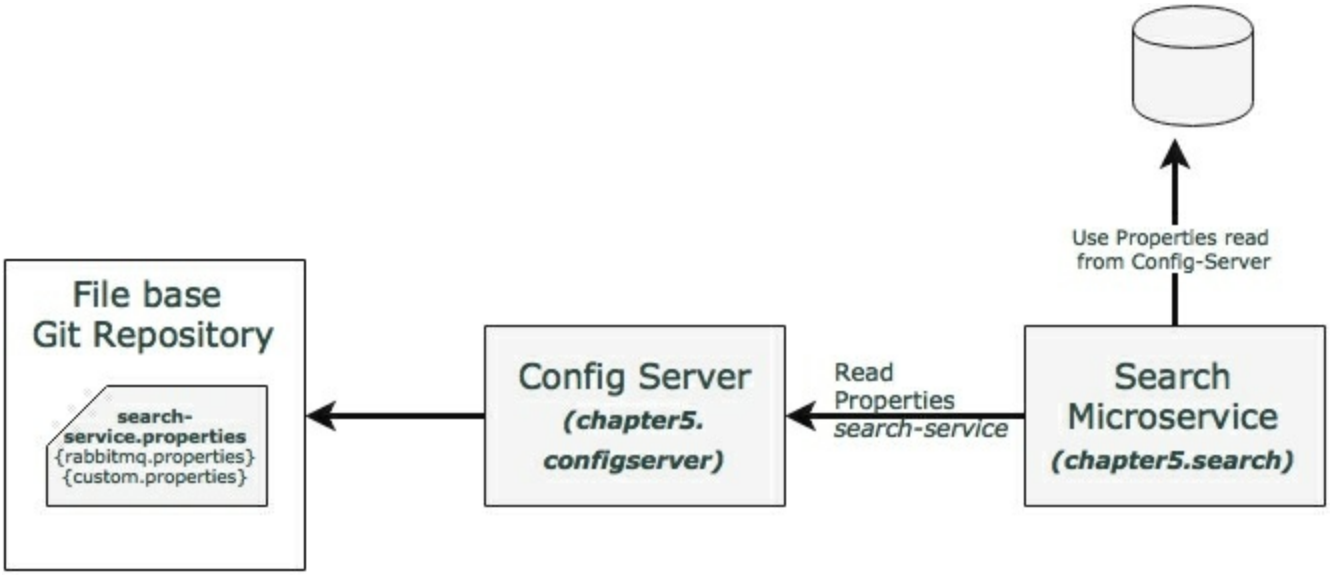


As shown in the preceding diagram, the Config client embedded in the Spring Boot microservices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level configurations such as trade limit per day, or infrastructure-related configurations such as server URLs, credentials, and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for `bootstrap.yaml` or `bootstrap.properties` for loading initial configuration properties. To make this work in a Spring Boot application, rename the `application.*` file to `bootstrap.*`.

What's next?

The next few sections demonstrate how to use the Config server in a real-world scenario. In order to do this, we will modify our search microservice (`chapter5.search`) to use the Config server. The following diagram depicts the scenario:



In this example, the Search service will read the Config server at startup by passing the service name. In this case, the service name of the search service will be `search-service`. The properties configured for the `search-service` include the RabbitMQ properties as well as a custom property.

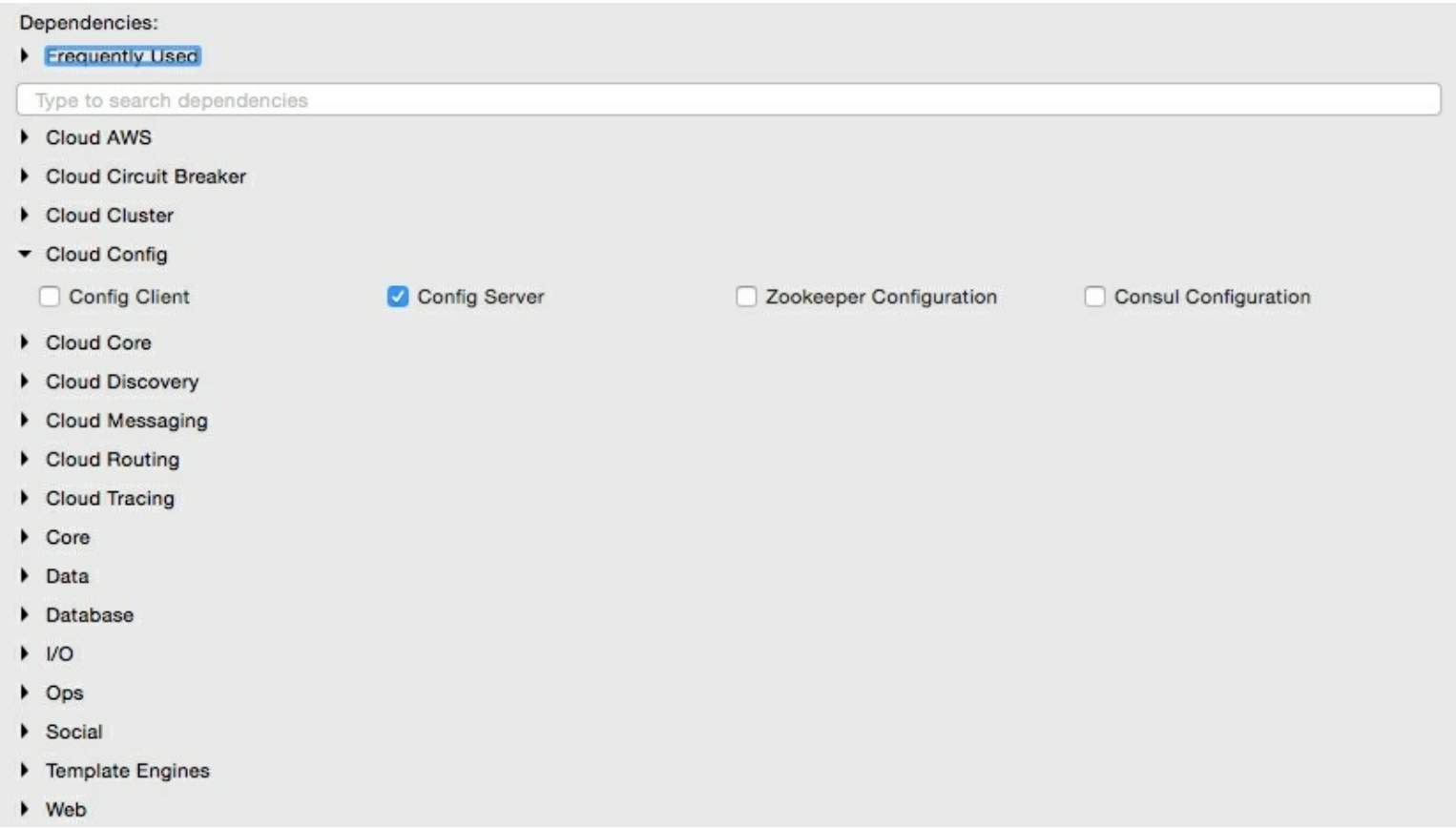
Note

The full source code of this section is available under the `chapter5.configserver` project in the code files.

Setting up the Config server

The following steps need to be followed to create a new Config server using STS:

1. Create a new **Spring Starter Project**, and select **Config Server** and **Actuator** as shown in the following diagram:



2. Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at https://github.com/spring-cloud-samples/config-repo. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.
3. Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended. The Config server in this chapter will use a local filesystem-based Git repository for demonstration purposes.
4. Enter the commands listed next to set up a local Git repository:

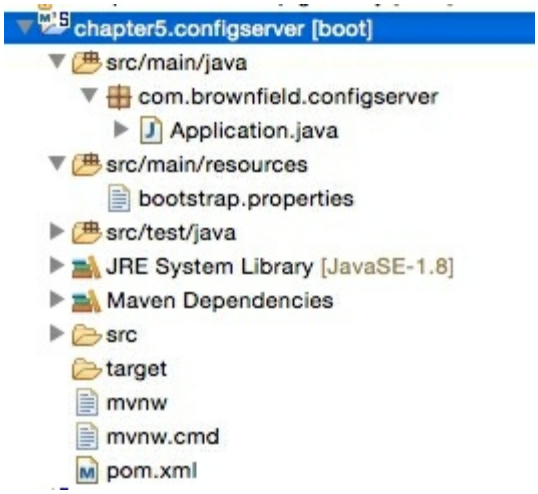
```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo message : helloworld > application.properties
$ git add -A .
$ git commit -m "Added sample application.properties"
```

This code snippet creates a new Git repository on the local filesystem. A property file named

`application.properties` with a `message` property and value `helloworld` is also created.

The file `application.properties` is created for demonstration purposes. We will change this in the subsequent sections.

5. The next step is to change the configuration in the Config server to use the Git repository created in the previous step. In order to do this, rename the file `application.properties` to `bootstrap.properties`:



6. Edit the contents of the new `bootstrap.properties` file to match the following:

```
server.port=8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

Port `8888` is the default port for the Config server. Even without configuring `server.port`, the Config server should bind to `8888`. In the Windows environment, an extra `/` is required in the file URL.

7. Optionally, rename the default package of the auto-generated `Application.java` from `com.example` to `com.brownfield.configserver`. Add `@EnableConfigServer` in `Application.java`:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication {
```

8. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
9. Visit `http://localhost:8888/env` to see whether the server is running. If everything is fine, this will list all environment configurations. Note that `/env` is an actuator endpoint.
10. Check `http://localhost:8888/application/default/master` to see the properties specific to `application.properties`, which were added in the earlier step. The browser will display the properties configured in `application.properties`. The browser should display contents similar to the following:

```
{"name":"application","profiles":
["default"],"label":"master","version":"6046fd2ff4fa09d3843767660d963866ffcc
7d28","propertySources":[{"name":"file:///Users/rvlabs /config-repo
```

/application.properties","source":{"message":"helloworld"}}]}

Understanding the Config server URL

In the previous section, we used `http://localhost:8888/application/default/master` to explore the properties. How do we interpret this URL?

The first element in the URL is the application name. In the given example, the application name should be `application`. The application name is a logical name given to the application, using the `spring.application.name` property in `bootstrap.properties` of the Spring Boot application. Each application must have a unique name. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID. If there is an application with the name `myapp`, then there should be a `myapp.properties` in the configuration repository to store all the properties related to that application.

The second part of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as `Dev`, `Test`, `Stage`, `Prod`, and the like, or segregating server configurations such as `Primary`, `Secondary`, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names that will be used for matching the file name in the repository. The default profile is named `default`. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for the development environment whereas the second is for the production environment:

```
application-development.properties
application-production.properties
```

These are accessible using the following URLs respectively:

- http://localhost:8888/application/development
- http://localhost:8888/application/production

The last part of the URL is the label, and is named `master` by default. The label is an optional Git label that can be used, if required.

In short, the URL is based on the following pattern:
`http://localhost:8888/{name}/{profile}/{label}`.

The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration:

- http://localhost:8888/application/default
- http://localhost:8888/application/master
- http://localhost:8888/application/default/master

There is an option to have different Git repositories for different profiles. This makes sense for production systems, since the access to different repositories could be different.

Accessing the Config Server from clients

In the previous section, a Config server is set up and accessed using a web browser. In this section, the Search microservice will be modified to use the Config server. The Search microservice will act as a Config client.

Follow these steps to use the Config server instead of reading properties from the `application.properties` file:

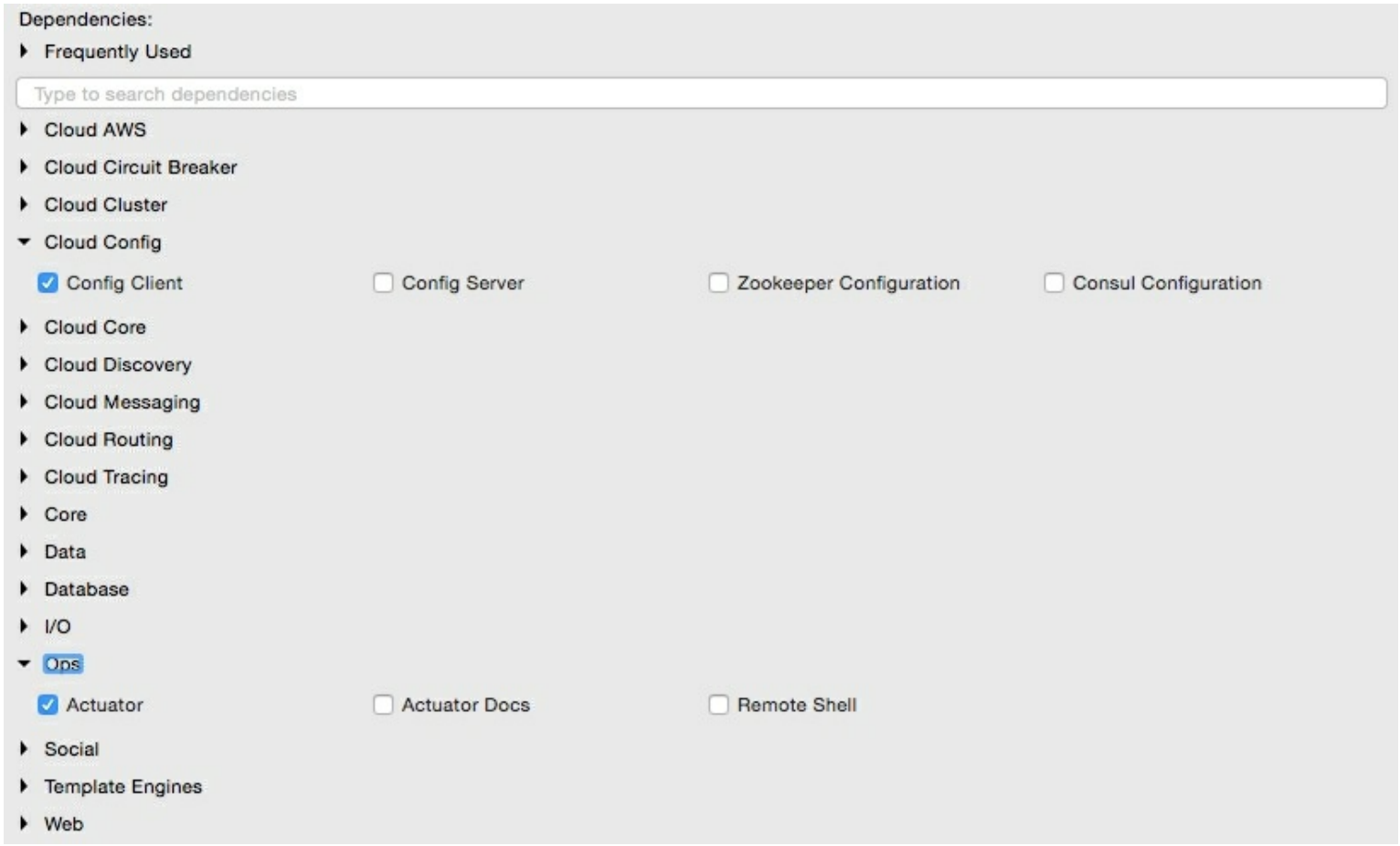
1. Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the `pom.xml` file. The actuator is mandatory for refreshing the configuration properties:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

2. Since we are modifying the Spring Boot Search microservice from the earlier chapter, we will have to add the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

3. The next screenshot shows the Cloud starter library selection screen. If the application is built from the ground up, select the libraries as shown in the following screenshot:



4. Rename `application.properties` to `bootstrap.properties`, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (`8888`) on the local host:

   The new `bootstrap.properties` file will look as follows:

```
spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888

server.port=8090

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

   `search-service` is a logical name given to the Search microservice. This will be treated as service ID. The Config server will look for `search-service.properties` in the repository to resolve the properties.
5. Create a new configuration file for `search-service`. Create a new `search-service.properties` under the `config-repo` folder where the Git repository is created. Note that `search-service` is the service ID given to the Search microservice in the `bootstrap.properties` file. Move service-specific properties from `bootstrap.properties`

to the new `search-service.properties` file. The following properties will be removed from `bootstrap.properties`, and added to `search-service.properties`:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

6. In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the property file. We will add `originairports.shutdown` to temporarily take out an airport from the search. Users will not get any flights when searching for an airport mentioned in the shutdown list:

```
originairports.shutdown=SEA
```

   In this example, we will not return any flights when searching with `SEA` as origin.
7. Commit this new file into the Git repository by executing the following commands:

```
git add -A .
git commit -m "adding new configuration"
```

8. The final `search-service.properties` file should look as follows:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
originairports.shutdown:SEA
```

9. The `chapter5.search` project's `bootstrap.properties` should look like the following:

```
spring.application.name=search-service
server.port=8090
spring.cloud.config.uri=http://localhost:8888
```

10. Modify the Search microservice code to use the configured parameter, `originairports.shutdown`. A `RefreshScope` annotation has to be added at the class level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the `SearchRestController` class:

```
@RefreshScope
```

11. Add the following instance variable as a place holder for the new property that is just added in the Config server. The property names in the `search-service.properties` file must match:

```
  @Value("${originairports.shutdown}")
  private String originAirportShutdownList;
```

12. Change the application code to use this property. This is done by modifying the `search` method as follows:

```
  @RequestMapping(value="/get", method = RequestMethod.POST)
  List<Flight> search(@RequestBody SearchQuery query){
    logger.info("Input : "+ query);
```

```
if(Arrays.asList(originAirportShutdownList.split(",")).contains(query.getOri
gin())){
    logger.info("The origin airport is in shutdown state");
    return new ArrayList<Flight>();
  }
  return searchComponent.search(query);
  }
```

The `search` method is modified to read the parameter `originAirportShutdownList` and see whether the requested origin is in the shutdown list. If there is a match, then instead of proceeding with the actual search, the search method will return an empty flight list.

13. Start the Config server. Then start the Search microservice. Make sure that the RabbitMQ server is running.
14. Modify the `chapter5.website` project to match the `bootstrap.properties` content as follows to utilize the Config server:

```
spring.application.name=test-client
server.port=8001
spring.cloud.config.uri=http://localhost:8888
```

15. Change the `run` method of `CommandLineRunner` in `Application.java` to query SEA as the origin airport:

```
SearchQuery searchQuery = new SearchQuery("SEA","SFO","22-JAN-16");
```

16. Run the `chapter5.website` project. The `CommandLineRunner` will now return an empty flight list. The following message will be printed in the server:

```
The origin airport is in shutdown state
```

Handling configuration changes

This section will demonstrate how to propagate configuration properties when there is a change:

1. Change the property in the `search-service.properties` file to the following:

   ```
   originairports.shutdown:NYC
   ```

   Commit the change in the Git repository. Refresh the Config server URL (`http://localhost:8888/search-service/default`) for this service and see whether the property change is reflected. If everything is fine, we will see the property change. The preceding request will force the Config server to read the property file again from the repository.
2. Rerun the website project again, and observe the `CommandLineRunner` execution. Note that in this case, we are not restarting the Search microservice nor the Config server. The service returns an empty flight list as earlier, and still complains as follows:

   ```
   The origin airport is in shutdown state
   ```

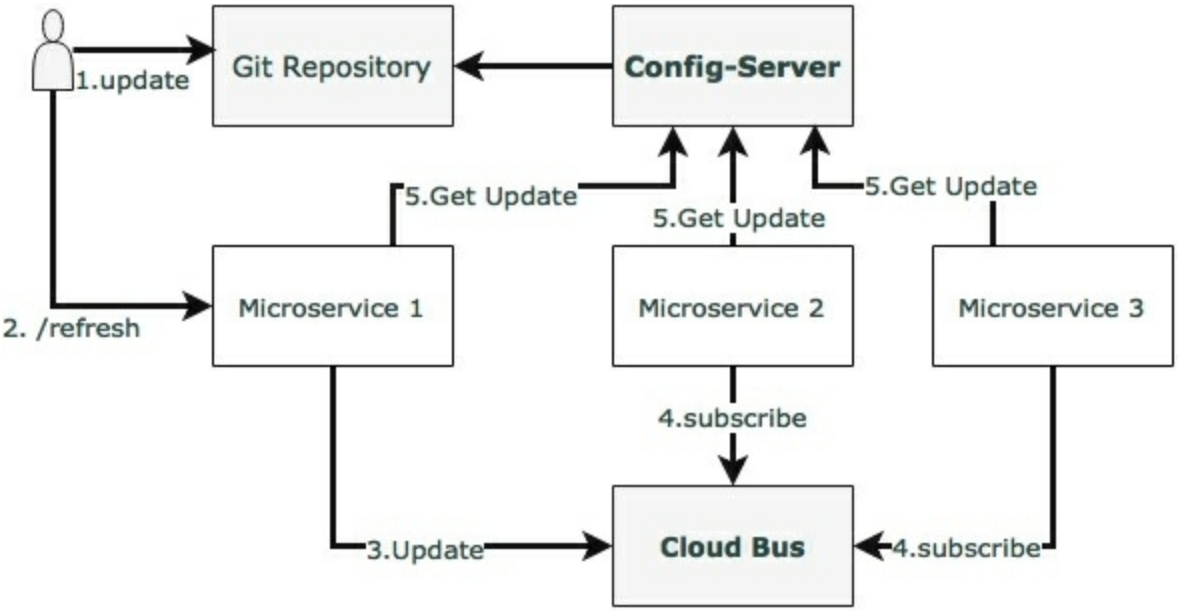
   This means the change is not reflected in the Search service, and the service is still working with an old copy of the configuration properties.
3. In order to force reloading of the configuration properties, call the `/refresh` endpoint of the Search microservice. This is actually the actuator's refresh endpoint. The following command will send an empty POST to the `/refresh` endpoint:

   ```
   curl -d {} localhost:8090/refresh
   ```
4. Rerun the website project, and observe the `CommandLineRunner` execution. This should return the list of flights that we have requested from SEA. Note that the website project may fail if the Booking service is not up and running.

   The `/refresh` endpoint will refresh the locally cached configuration properties, and reload fresh values from the Config server.

Spring Cloud Bus for propagating configuration changes

With the preceding approach, configuration parameters can be changed without restarting the microservices. This is good when there are only one or two instances of the services running. What happens if there are many instances? For example, if there are five instances, then we have to hit `/refresh` against each service instance. This is definitely a cumbersome activity:



The Spring Cloud Bus provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations. This is particularly handy when there are many service instances of a microservice running or when there are many microservices of different types running. This is done by connecting all service instances through a single message broker. Each instance subscribes for change events, and refreshes its local configuration when required. This refresh is triggered by making a call to any one instance by hitting the `/bus/refresh` endpoint, which then propagates the changes through the cloud bus and the common message broker.

In this example, RabbitMQ is used as the AMQP message broker. Implement this by following the steps documented as follows:

1. Add a new dependency in the `chapter5.search` project's `pom.xml` file to introduce the Cloud Bus dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

2. The Search microservice also needs connectivity to the RabbitMQ, but this is already provided in `search-service.properties`.
3. Rebuild and restart the Search microservice. In this case, we will run two instances of the Search microservice from a command line, as follows:

```
java -jar -Dserver.port=8090  search-1.0.jar
java -jar -Dserver.port=8091  search-1.0.jar
```

The two instances of the Search service will be now running, one on port `8090` and another one on `8091`.

4. Rerun the website project. This is just to make sure that everything is working. The Search service should return one flight at this point.
5. Now, update `search-service.properties` with the following value, and commit to Git:

   ```
   originairports.shutdown:SEA
   ```

6. Run the following command to `/bus/refresh`. Note that we are running a new bus endpoint against one of the instances, `8090` in this case:

   ```
   curl -d {} localhost:8090/bus/refresh
   ```

7. Immediately, we will see the following message for both instances:

   ```
   Received remote refresh request. Keys refreshed [originairports.shutdown]
   ```

   The bus endpoint sends a message to the message broker internally, which is eventually consumed by all instances, reloading their property files. Changes can also be applied to a specific application by specifying the application name like so:

   ```
   /bus/refresh?destination=search-service:**
   ```

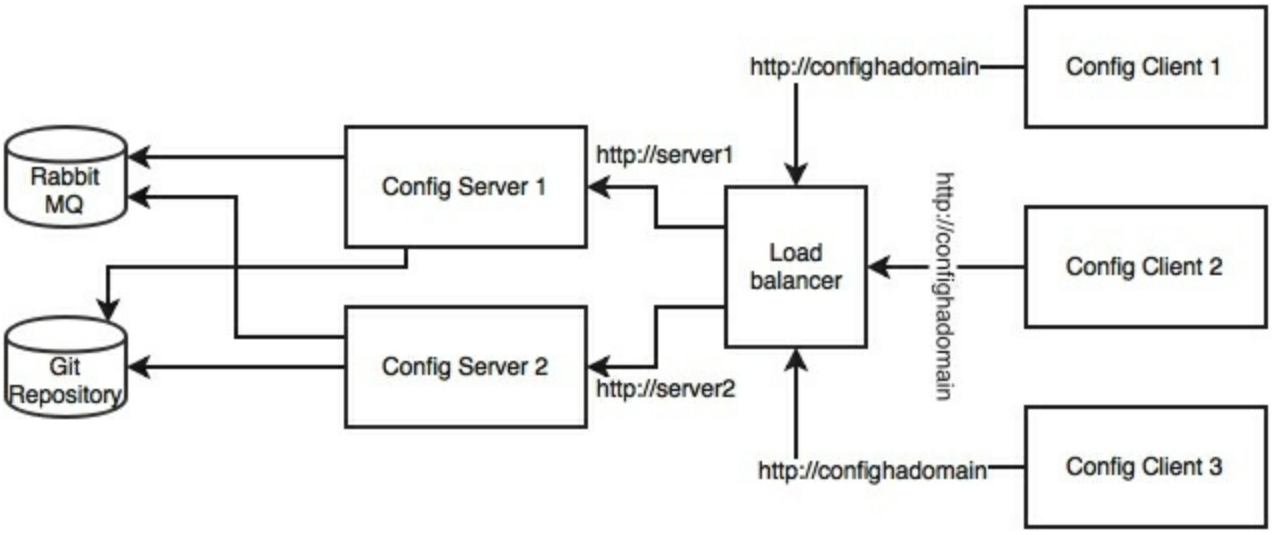
We can also refresh specific properties by setting the property name as a parameter.

Setting up high availability for the Config server

The previous sections explored how to set up the Config server, allowing real-time refresh of configuration properties. However, the Config server is a single point of failure in this architecture.

There are three single points of failure in the default architecture that was established in the previous section. One of them is the availability of the Config server itself, the second one is the Git repository, and the third one is the RabbitMQ server.

The following diagram shows a high availability architecture for the Config server:



The architecture mechanisms and rationale are explained as follows:

The Config server requires high availability, since the services won't be able to bootstrap if the Config server is not available. Hence, redundant Config servers are required for high availability. However, the applications can continue to run if the Config server is unavailable after the services are bootstrapped. In this case, services will run with the last known configuration state. Hence, the Config server availability is not at the same critical level as the microservices availability.

In order to make the Config server highly available, we need multiple instances of the Config servers. Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel. Based on the load on the configuration server, a number of instances have to be adjusted. The `bootstrap.properties` file is not capable of handling more than one server address. Hence, multiple configuration servers should be configured to run behind a load balancer or behind a local DNS with failover and fallback capabilities. The load balancer or DNS server URL will be configured in the microservices' `bootstrap.properties` file. This is with the assumption that the DNS or the load balancer is highly available and capable of handling failovers.

In a production scenario, it is not recommended to use a local file-based Git repository. The configuration server should be typically backed with a highly available Git service. This is possible

by either using an external highly available Git service or a highly available internal Git service. SVN can also be considered.

Having said that, an already bootstrapped Config server is always capable of working with a local copy of the configuration. Hence, we need a highly available Git only when the Config server needs to be scaled. Therefore, this too is not as critical as the microservices availability or the Config server availability.

Note

The GitLab example for setting up high availability is available at https://about.gitlab.com/high-availability/.

RabbitMQ also has to be configured for high availability. The high availability for RabbitMQ is needed only to push configuration changes dynamically to all instances. Since this is more of an offline controlled activity, it does not really require the same high availability as required by the components.

RabbitMQ high availability can be achieved by either using a cloud service or a locally configured highly available RabbitMQ service.

Note

Setting up high availability for Rabbit MQ is documented at https://www.rabbitmq.com/ha.html.

Monitoring the Config server health

The Config server is nothing but a Spring Boot application, and is, by default, configured with an actuator. Hence, all actuator endpoints are applicable for the Config server. The health of the server can be monitored using the following actuator URL: `http://localhost:8888/health`.

Config server for configuration files

We may run into scenarios where we need a complete configuration file such as `logback.xml` to be externalized. The Config server provides a mechanism to configure and store such files. This is achievable by using the URL format as follows: `/{name}/{profile}/{label}/{path}`.

The name, profile, and label have the same meanings as explained earlier. The path indicates the file name such as `logback.xml`.

Completing changes to use the Config server

In order to build this capability to complete BrownField Airline's PSS, we have to make use of the configuration server for all services. All microservices in the examples given in `chapter5.*` need to make similar changes to look to the Config server for getting the configuration parameters.

The following are a few key change considerations:

- The Fare service URL in the booking component will also be externalized:

```
private static final String FareURL = "/fares";

@Value("${fares-service.url}")
private String fareServiceUrl;

Fare = restTemplate.getForObject(fareServiceUrl+FareURL +"/get?
flightNumber="+record.getFlightNumber()+"&flightDate="+record.getFlightDate(
),Fare.class);
```

  As shown in the preceding code snippet, the Fare service URL is fetched through a new property: `fares-service.url`.
- We are not externalizing the queue names used in the Search, Booking, and Check-in services at the moment. Later in this chapter, these will be changed to use Spring Cloud Streams.

Feign as a declarative REST client

In the Booking microservice, there is a synchronous call to Fare. `RestTemplate` is used for making the synchronous call. When using `RestTemplate`, the URL parameter is constructed programmatically, and data is sent across to the other service. In more complex scenarios, we will have to get to the details of the HTTP APIs provided by `RestTemplate` or even to APIs at a much lower level.

Feign is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls. Spring Cloud Feign works on a declarative principle. When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client. The developer need not worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, developers need not get into the details of the HTTP level APIs provided by `RestTemplate`.

The following code snippet is the existing code in the Booking microservice for calling the Fare service:

```
Fare fare = restTemplate.getForObject(FareURL +"/get?
flightNumber="+record.getFlightNumber()+"&flightDate="+record.getFlightDate(),Fa
re.class);
```

In order to use Feign, first we need to change the `pom.xml` file to include the Feign dependency as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

For a new Spring Starter project, **Feign** can be selected from the starter library selection screen, or from http://start.spring.io/. This is available under **Cloud Routing** as shown in the following screenshot:



The next step is to create a new `FareServiceProxy` interface. This will act as a proxy interface of the actual Fare service:

```
@FeignClient(name="fares-proxy", url="localhost:8080/fares")
public interface FareServiceProxy {
  @RequestMapping(value = "/get", method=RequestMethod.GET)
  Fare getFare(@RequestParam(value="flightNumber") String flightNumber,
@RequestParam(value="flightDate") String flightDate);
}
```

The `FareServiceProxy` interface has a `@FeignClient` annotation. This annotation tells Spring to create a REST client based on the interface provided. The value could be a service ID or a logical name. The `url` indicates the actual URL where the target service is running. Either name or value is mandatory. In this case, since we have `url`, the `name` attribute is irrelevant.

Use this service proxy to call the Fare service. In the Booking microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered. This will be done by adding `@EnableFeignClients` at the class level of `BookingComponent`. Optionally, we can also give the package names to scan.

Change `BookingComponent`, and make changes to the calling part. This is as simple as calling another Java interface:

```
Fare = fareServiceProxy.getFare(record.getFlightNumber(),
record.getFlightDate());
```

Rerun the Booking microservice to see the effect.

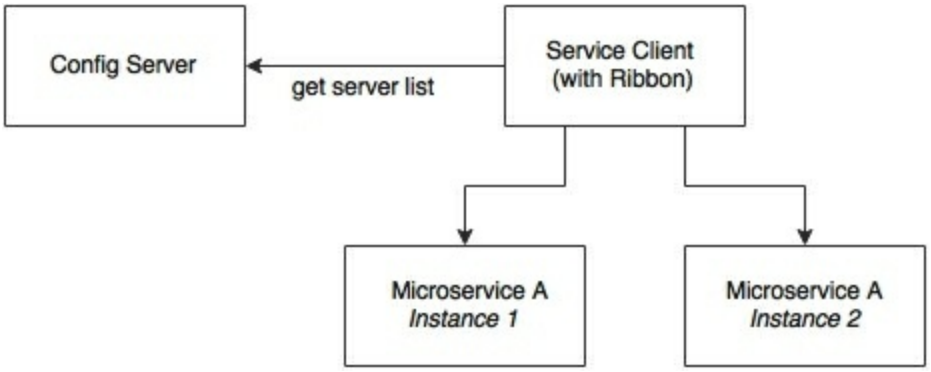
The URL of the Fare service in the `FareServiceProxy` interface is hardcoded: `url="localhost:8080/fares"`.

For the time being, we will keep it like this, but we are going to change this later in this chapter.

Ribbon for load balancing

In the previous setup, we were always running with a single instance of the microservice. The URL is hardcoded both in client as well as in the service-to-service calls. In the real world, this is not a recommended approach, since there could be more than one service instance. If there are multiple instances, then ideally, we should use a load balancer or a local DNS server to abstract the actual instance locations, and configure an alias name or the load balancer address in the clients. The load balancer then receives the alias name, and resolves it with one of the available instances. With this approach, we can configure as many instances behind a load balancer. It also helps us to handle server failures transparent to the client.

This is achievable with Spring Cloud Netflix Ribbon. Ribbon is a client-side load balancer which can do round-robin load balancing across a set of servers. There could be other load balancing algorithms possible with the Ribbon library. Spring Cloud offers a declarative way to configure and use the Ribbon client.



As shown in the preceding diagram, the Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm.

In order to use the Ribbon client, we will have to add the following dependency to the `pom.xml` file:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

In case of development from ground up, this can be selected from the Spring Starter libraries, or from http://start.spring.io/. Ribbon is available under **Cloud Routing**:



Update the Booking microservice configuration file, `booking-service.properties`, to include a new property to keep the list of the Fare microservices:

```
fares-proxy.ribbon.listOfServers=localhost:8080,localhost:8081
```

Going back and editing the `FareServiceProxy` class created in the previous section to use the Ribbon client, we note that the value of the `@RequestMapping` annotations is changed from `/get` to `/fares/get` so that we can move the host name and port to the configuration easily:

```
@FeignClient(name="fares-proxy")
@RibbonClient(name="fares")
public interface FareServiceProxy {
  @RequestMapping(value = "fares/get", method=RequestMethod.GET)
```

We can now run two instances of the Fares microservices. Start one of them on `8080`, and the other one on `8081`:

```
java -jar -Dserver.port=8080 fares-1.0.jar
java -jar -Dserver.port=8081 fares-1.0.jar
```

Run the Booking microservice. When the Booking microservice is bootstrapped, the `CommandLineRunner` automatically inserts one booking record. This will go to the first server.

When running the website project, it calls the Booking service. This request will go to the second server.

On the Booking service, we see the following trace, which says there are two servers enlisted:

```
DynamicServerListLoadBalancer:{NFLoadBalancer:name=fares-proxy,current

list of Servers=[localhost:8080, localhost:8081],Load balancer stats=Zone stats:
{unknown=[Zone:unknown;  Instance count:2;  Active connections count: 0;
Circuit breaker tripped count: 0;  Active connections per server: 0.0;]
```

},

Eureka for registration and discovery

So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances.

Ribbon-based load balancing is sufficient for most of the microservices requirements. However, this approach falls short in a couple of scenarios:

- If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
- When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
- In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the microservices should self-manage their life cycle by dynamically registering service availability, and provision automated discovery for consumers.

Understanding dynamic service registration and discovery

Dynamic registration is primarily from the service provider's point of view. With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.

Dynamic discovery is applicable from the service consumer's point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

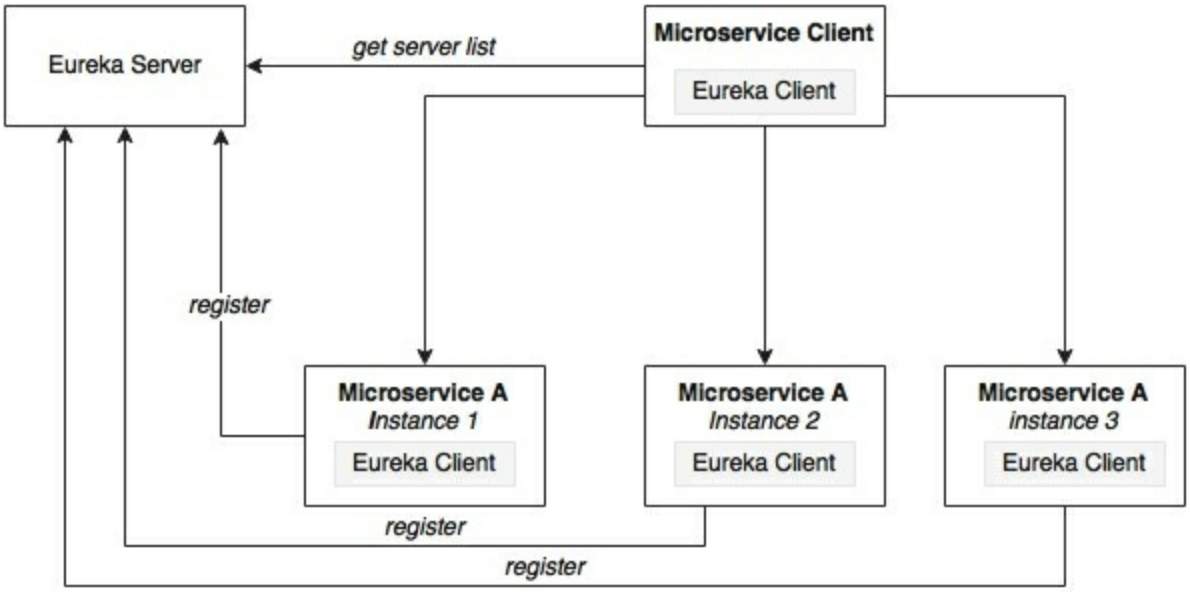
The clients may keep a local cache of the registry data for faster access. Some registry implementations allow clients to keep a watch on the items they are interested in. In this approach, the state changes in the registry server will be propagated to the interested parties to avoid using stale data.

There are a number of options available for dynamic service registration and discovery. Netflix Eureka, ZooKeeper, and Consul are available as part of Spring Cloud, as shown in the http://start.spring.io/ screenshot given next. Etcd is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. In this chapter, we will focus on the Eureka implementation:



Understanding Eureka

Spring Cloud Eureka also comes from Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications. Eureka is primarily used for self-registration, dynamic discovery, and load balancing. Eureka uses Ribbon for load balancing internally:



As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability. The registration typically includes service identity and its URLs. The microservices use the Eureka client for registering their availability. The consuming components will also use the Eureka client for discovering the service instances.

When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease in a few attempts, that service endpoint will be taken out of the service registry. The registry information will be replicated to all Eureka clients so that the clients have to go to the remote Eureka server for each and every request. Eureka clients fetch the registry information from the server, and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

When a client wants to contact a microservice endpoint, the Eureka client provides a list of currently available services based on the requested service ID. The Eureka server is zone aware. Zone information can also be supplied when registering a service. When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the Eureka client. The communication between the Eureka client and the server is done using REST and JSON.

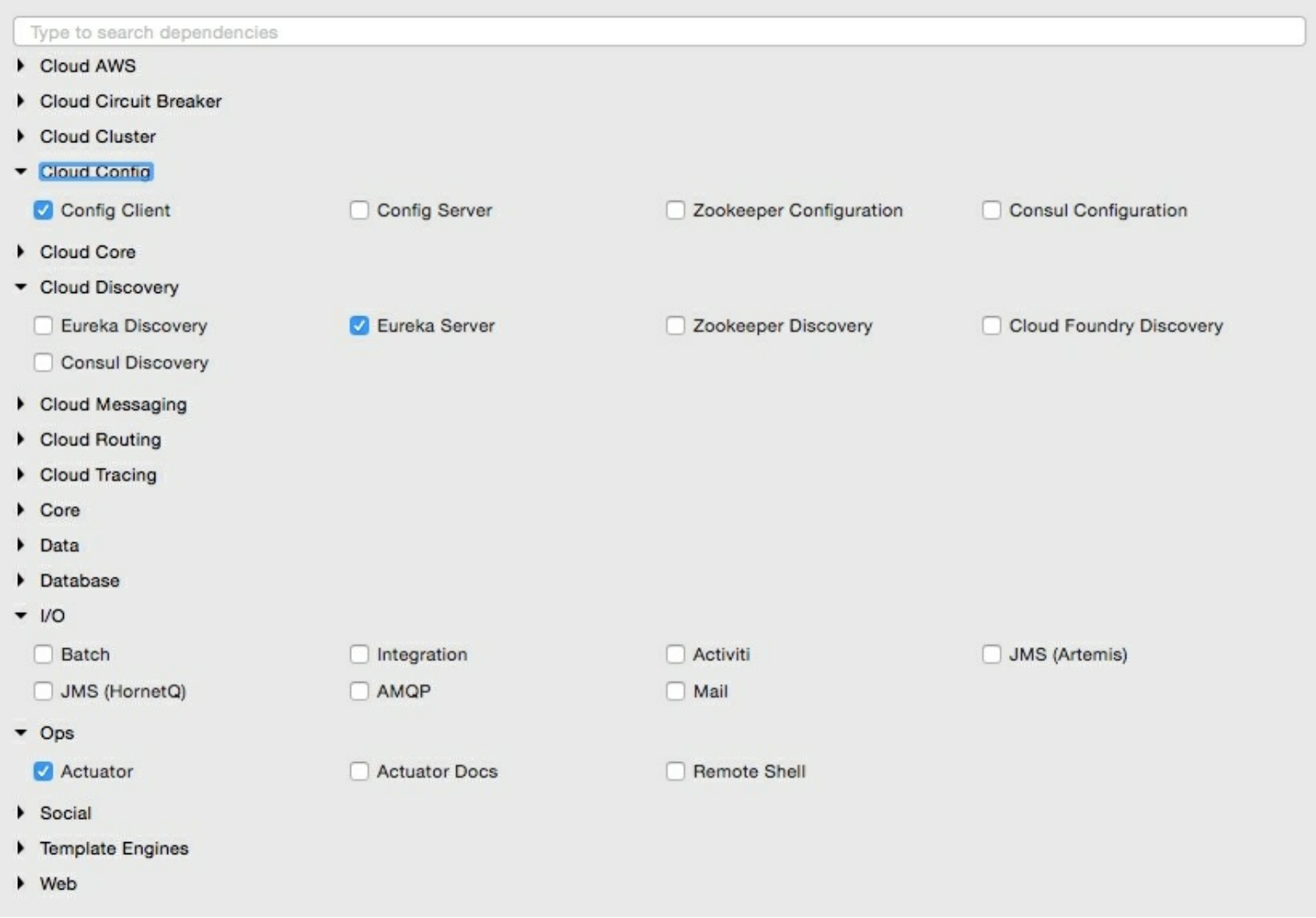
Setting up the Eureka server

In this section, we will run through the steps required for setting up the Eureka server.

Note

The full source code of this section is available under the `chapter5.eurekaserver` project in the code files. Note that the Eureka server registration and refresh cycles take up to 30 seconds. Hence, when running services and clients, wait for 40-50 seconds.

1. Start a new Spring Starter project, and select **Config Client**, **Eureka Server**, and **Actuator**:



The project structure of the Eureka server is shown in the following image:



Note that the main application is named `EurekaserverApplication.java`.

2. Rename `application.properties` to `bootstrap.properties` since this is using the Config server. As we did earlier, configure the details of the Config server in the `bootsratp.properties` file so that it can locate the Config server instance. The `bootstrap.properties` file will look as follows:

```
spring.application.name=eureka-server1
server.port:8761
spring.cloud.config.uri=http://localhost:8888
```

The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the `eureka.client.serviceUrl.defaultZone` property.

In the standalone mode, we point `eureka.client.serviceUrl.defaultZone` back to the same standalone instance. Later we will see how we can run Eureka servers in a clustered mode.

3. Create a `eureka-server1.properties` file, and update it in the Git repository. `eureka-server1` is the name of the application given in the application's `bootstrap.properties` file in the previous step. As shown in the following code, `serviceUrl` points back to the same server. Once the following properties are added, commit the file to the Git repository:

```
spring.application.name=eureka-server1
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

4. Change the default `Application.java`. In this example, the package is also renamed as `com.brownfield.pss.eurekaserver`, and the class name changed to `EurekaserverApplication`. In `EurekaserverApplication`, add `@EnableEurekaServer`:

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaserverApplication {
```

5. We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose **Run As | Spring Boot App**. Once the application is started, open `http://localhost:8761` in a browser to see the Eureka console.
6. In the console, note that there is no instance registered under **Instances currently registered with Eureka**. Since no services have been started with the Eureka client enabled, the list is empty at this point.
7. Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add the Eureka dependencies to the `pom.xml` file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:



8. Since we are modifying our microservices, add the following additional dependency to all microservices in their `pom.xml` files:

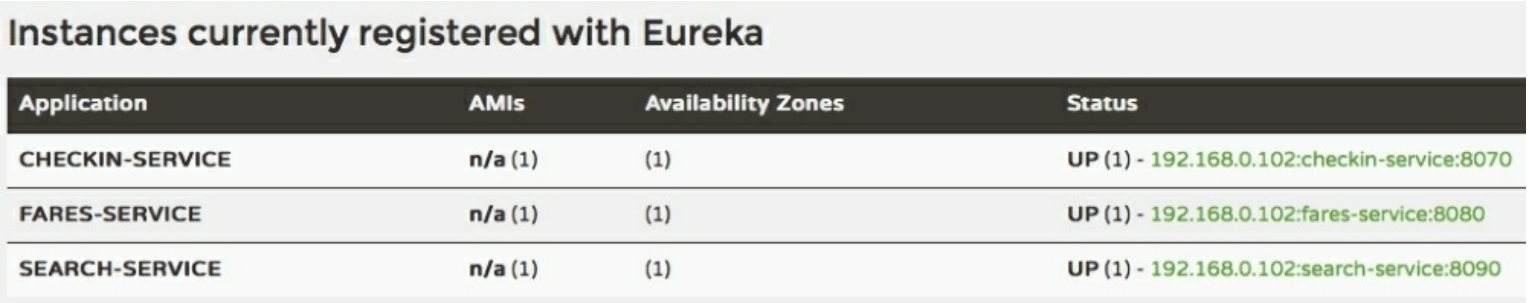
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
```

```
</dependency>
```

9. The following property has to be added to all microservices in their respective configuration files under `config-repo`. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:

```
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
```

10. Add `@EnableDiscoveryClient` to all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability.
11. Start all servers except Booking. Since we are using the Ribbon client on the Booking service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon.
12. Going to the Eureka URL (`http://localhost:8761`), you can see that all three instances are up and running:



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

Time to fix the issue with Booking. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.

13. Remove the following dependency:

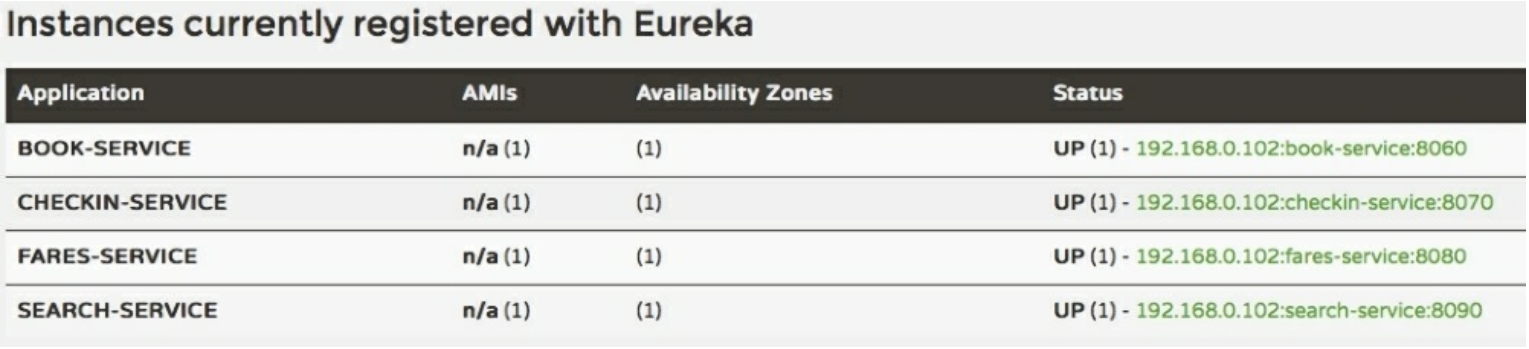
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

14. Also remove the `@RibbonClient(name="fares")` annotation from the `FareServiceProxy` class.
15. Update `@FeignClient(name="fares-service")` to match the actual Fare microservices' service ID. In this case, `fare-service` is the service ID configured in the Fare microservices' `bootstrap.properties`. This is the name that the Eureka discovery client sends to the Eureka server. The service ID will be used as a key for the services registered in the Eureka server.
16. Also remove the list of servers from the `booking-service.properties` file. With Eureka, we are going to dynamically discover this list from the Eureka server:

```
fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081
```

17. Start the Booking service. You will see that `CommandLineRunner` successfully created a booking, which involves calling the Fare services using the Eureka discovery mechanism. Go

back to the URL to see all the registered services:



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOK-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:book-service:8060
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

18. Change the website project's `bootstrap.properties` file to make use of Eureka rather than connecting directly to the service instances. We will not use the Feign client in this case. Instead, for demonstration purposes, we will use the load balanced `RestTemplate`. Commit these changes to the Git repository:

```
spring.application.name=test-client
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
```

19. Add `@EnableDiscoveryClient` to the `Application` class to make the client Eureka-aware.
20. Edit both `Application.java` as well as `BrownFieldSiteController.java`. Add three `RestTemplate` instances. This time, we annotate them with `@Loadbalanced` to ensure that we use the load balancing features using Eureka and Ribbon. `RestTemplate` cannot be automatically injected. Hence, we have to provide a configuration entry as follows:

```
@Configuration
class AppConfiguration {
    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
@Autowired
RestTemplate searchClient;

@Autowired
RestTemplate bookingClient;

@Autowired
RestTemplate checkInClient;
```

21. We use these `RestTemplate` instances to call the microservices. Replace the hardcoded URLs with service IDs that are registered in the Eureka server. In the following code, we use the service names `search-service`, `book-service`, and `checkin-service` instead of explicit host names and ports:

```
Flight[] flights = searchClient.postForObject("http://search-
service/search/get", searchQuery, Flight[].class);
```

```
long bookingId = bookingClient.postForObject("http://book-
service/booking/create", booking, long.class);

long checkinId = checkInClient.postForObject("http://checkin-
service/checkin/create", checkIn, long.class);
```

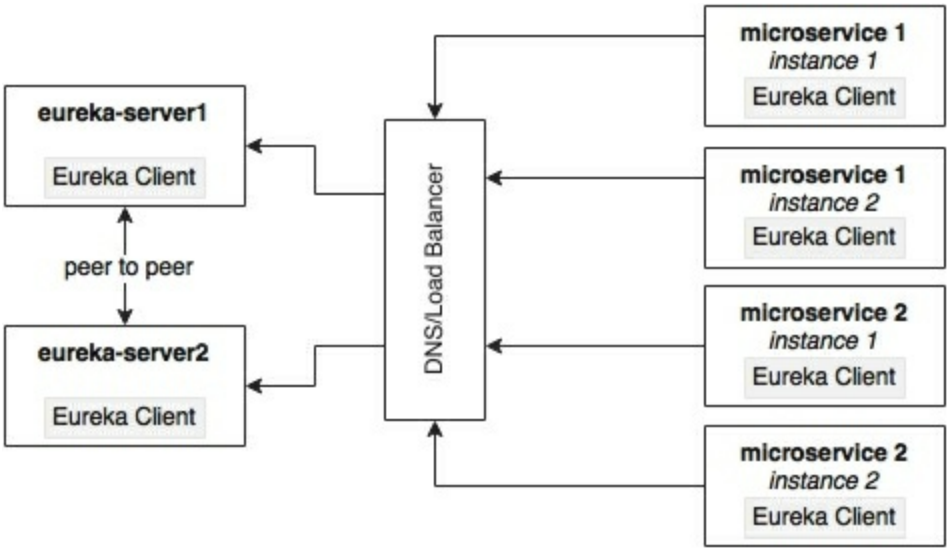
22. We are now ready to run the client. Run the website project. If everything is fine, the website project's `CommandLineRunner` will successfully perform search, booking, and check-in. The same can also be tested using the browser by pointing the browser to `http://localhost:8001`.

High availability for Eureka

In the previous example, there was only one Eureka server in standalone mode. This is not good enough for a real production system.

The Eureka client connects to the server, fetches registry information, and stores it locally in a cache. The client always works with this local cache. The Eureka client checks the server periodically for any state changes. In the case of a state change, it downloads the changes from the server, and updates the cache. If the Eureka server is not reachable, then the Eureka clients can still work with the last-known state of the servers based on the data available in the client cache. However, this could lead to stale state issues quickly.

This section will explore the high availability for the Eureka server. The high availability architecture is shown in the following diagram:



The Eureka server is built with a peer-to-peer data synchronization mechanism. The runtime state information is not stored in a database, but managed using an in-memory cache. The high availability implementation favors availability and partition tolerance in the CAP theorem, leaving out consistency. Since the Eureka server instances are synchronized with each other using an asynchronous mechanism, the states may not always match between server instances. The peer-to-peer synchronization is done by pointing `serviceUrls` to each other. If there is more than one Eureka server, each one has to be connected to at least one of the peer servers. Since the state is replicated across all peers, Eureka clients can connect to any one of the available Eureka servers.

The best way to achieve high availability for Eureka is to cluster multiple Eureka servers, and run them behind a load balancer or a local DNS. The clients always connect to the server using the DNS/load balancer. At runtime, the load balancer takes care of selecting the appropriate servers. This load balancer address will be provided to the Eureka clients.

This section will showcase how to run two Eureka servers in a cluster for high availability. For this, define two property files: `eureka-server1` and `eureka-server2`. These are peer servers; if one fails, the other one will take over. Each of these servers will also act as a client for the other so that they can sync their states. Two property files are defined in the following snippet. Upload and commit these properties to the Git repository.

The client URLs point to each other, forming a peer network as shown in the following configuration:

eureka-server1.properties

```
eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

eureka-server2.properties

```
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

Update the `bootstrap.properties` file of Eureka, and change the application name to `eureka`. Since we are using two profiles, based on the active profile supplied at startup, the Config server will look for either `eureka-server1` or `eureka-server2`:

```
spring.application.name=eureka
spring.cloud.config.uri=http://localhost:8888
```

Start two instances of the Eureka servers, `server1` on `8761` and `server2` on `8762`:

```
java -jar –Dserver.port=8761 -Dspring.profiles.active=server1 demo-0.0.1-SNAPSHOT.jar
java -jar –Dserver.port=8762 -Dspring.profiles.active=server2 demo-0.0.1-SNAPSHOT.jar
```

All our services still point to the first server, `server1`. Open both the browser windows: `http://localhost:8761` and `http://localhost:8762`.

Start all microservices. The one which opened `8761` will immediately reflect the changes, whereas the other one will take 30 seconds for reflecting the states. Since both the servers are in a cluster, the state is synchronized between these two servers. If we keep these servers behind a load balancer/DNS, then the client will always connect to one of the available servers.

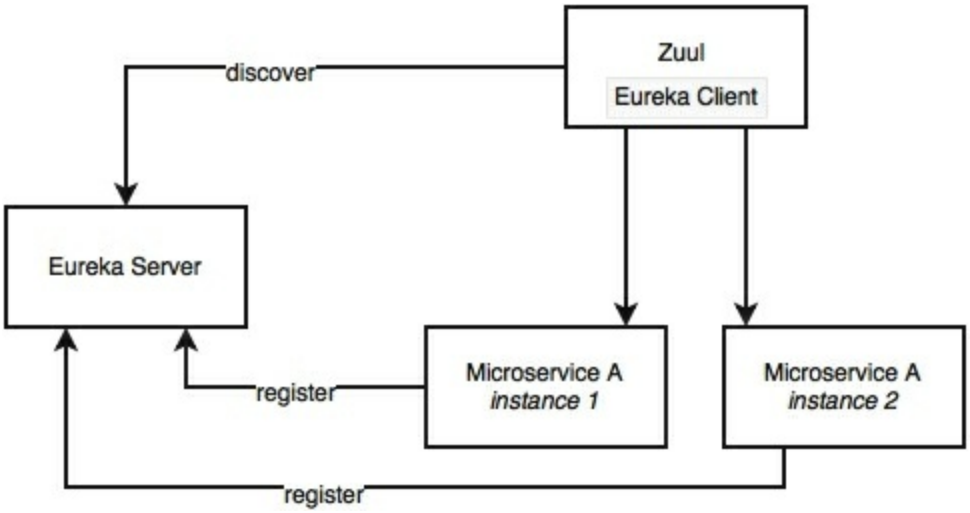
After completing this exercise, switch back to the standalone mode for the remaining exercises.

Zuul proxy as the API gateway

In most microservice implementations, internal microservice endpoints are not exposed outside. They are kept as private services. A set of public services will be exposed to the clients using an API gateway. There are many reasons to do this:

- Only a selected set of microservices are required by the clients.
- If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
- It is hard to implement client-specific transformations at the service endpoint.
- If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Zuul is a simple gateway service or edge service that suits these situations well. Zuul also comes from the Netflix family of microservice products. Unlike many enterprise API gateway products, Zuul provides complete control for the developers to configure or program based on specific requirements:



The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.

The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on. In simple terms, we can consider Zuul a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

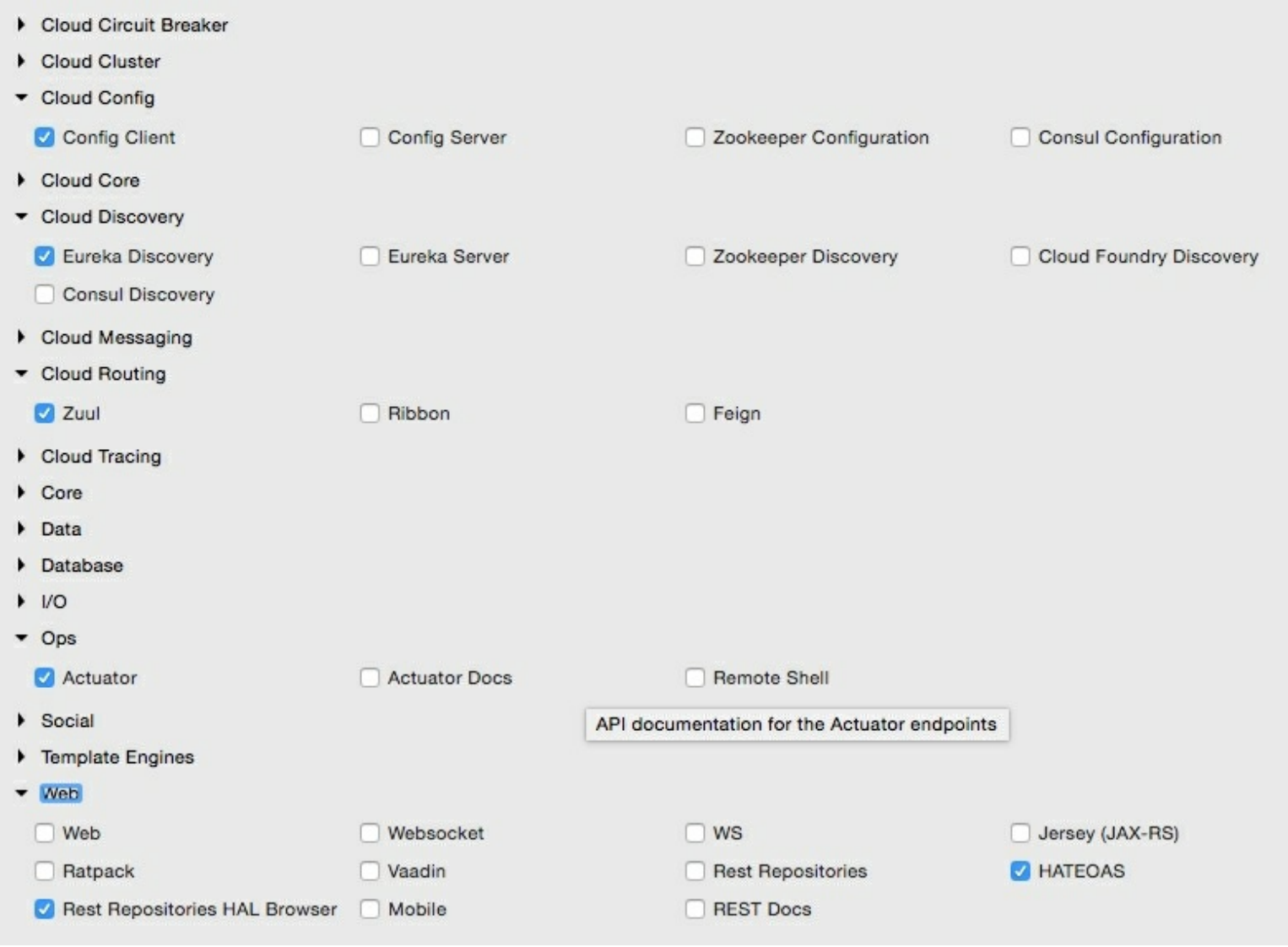
Setting up Zuul

Unlike the Eureka server and the Config server, in typical deployments, Zuul is specific to a microservice. However, there are deployments in which one API gateway covers many microservices. In this case, we are going to add Zuul for each of our microservices: Search, Booking, Fare, and Check-in:

Note

The full source code of this section is available under the `chapter5.*-apigateway` project in the code files.

1. Convert the microservices one by one. Start with Search API Gateway. Create a new Spring Starter project, and select **Zuul**, **Config Client**, **Actuator**, and **Eureka Discovery**:



The project structure for `search-apigateway` is shown in the following diagram:



2. The next step is to integrate the API gateway with Eureka and the Config server. Create a `search-apigateway.property` file with the contents given next, and commit to the Git repository.

   This configuration also sets a rule on how to forward traffic. In this case, any request coming on the `/api` endpoint of the API gateway should be sent to `search-service`:

   ```
   spring.application.name=search-apigateway
   zuul.routes.search-apigateway.serviceId=search-service
   zuul.routes.search-apigateway.path=/api/**
   eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
   ```

   `search-service` is the service ID of the Search service, and it will be resolved using the Eureka server.
3. Update the `bootstrap.properties` file of `search-apigateway` as follows. There is nothing new in this configuration—a name to the service, the port, and the Config server URL:

   ```
   spring.application.name=search-apigateway
   server.port=8095
   spring.cloud.config.uri=http://localhost:8888
   ```

4. Edit `Application.java`. In this case, the package name and the class name are also changed to `com.brownfield.pss.search.apigateway` and `SearchApiGateway` respectively. Also add `@EnableZuulProxy` to tell Spring Boot that this is a Zuul proxy:

   ```
   @EnableZuulProxy
   @EnableDiscoveryClient
   @SpringBootApplication
   public class SearchApiGateway {
   ```

5. Run this as a Spring Boot app. Before that, ensure that the Config server, the Eureka server, and the Search microservice are running.
6. Change the website project's `CommandLineRunner` as well as `BrownFieldSiteController` to make use of the API gateway:

   ```
   Flight[] flights = searchClient.postForObject("http://search-
   ```

```
apigateway/api/search/get", searchQuery, Flight[].class);
```

In this case, the Zuul proxy acts as a reverse proxy which proxies all microservice endpoints to consumers. In the preceding example, the Zuul proxy does not add much value, as we just pass through the incoming requests to the corresponding backend service.

Zuul is particularly useful when we have one or more requirements like the following:

- Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint. The gateway can handle security policies, token handling, and so on before passing the request to the relevant services behind. It can also do basic rejections based on some business policies such as blocking requests coming from certain black-listed users.
- Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data, and push it to an external system for analysis. This will be handy as we can do this at one place rather than applying it across many microservices.
- API gateways are useful in scenarios where dynamic routing is required based on fine-grained controls. For example, send requests to different service instances based on business specific values such as "origin country". Another example is all requests coming from a region to be sent to one group of service instances. Yet another example is all requests requesting for a particular product have to be routed to a group of service instances.
- Handling the load shredding and throttling requirements is another scenario where API gateways are useful. This is when we have to control load based on set thresholds such as number of requests in a day. For example, control requests coming from a low-value third party online channel.
- The Zuul gateway is useful for fine-grained load balancing scenarios. The Zuul, Eureka client, and Ribbon together provide fine-grained controls over the load balancing requirements. Since the Zuul implementation is nothing but another Spring Boot application, the developer has full control over the load balancing.
- The Zuul gateway is also useful in scenarios where data aggregation requirements are in place. If the consumer wants higher level coarse-grained services, then the gateway can internally aggregate data by calling more than one service on behalf of the client. This is particularly applicable when the clients are working in low bandwidth environments.

Zuul also provides a number of filters. These filters are classified as pre filters, routing filters, post filters, and error filters. As the names indicate, these are applied at different stages of the life cycle of a service call. Zuul also provides an option for developers to write custom filters. In order to write a custom filter, extend from the abstract `ZuulFilter`, and implement the following methods:

```
public class CustomZuulFilter extends ZuulFilter{
public Object run(){}
public boolean shouldFilter(){}
public int filterOrder(){}
public String filterType(){}
```

Once a custom filter is implemented, add that class to the main context. In our example case, add this

to the `SearchApiGateway` class as follows:

```
@Bean
public CustomZuulFilter customFilter() {
    return new CustomZuulFilter();
}
```

As mentioned earlier, the Zuul proxy is a Spring Boot service. We can customize the gateway programmatically in the way we want. As shown in the following code, we can add custom endpoints to the gateway, which, in turn, can call the backend services:

```
@RestController
class SearchAPIGatewayController {

  @RequestMapping("/")
  String greet(HttpServletRequest req){
    return "<H1>Search Gateway Powered By Zuul</H1>";
  }
}
```

In the preceding case, it just adds a new endpoint, and returns a value from the gateway. We can further use `@Loadbalanced RestTemplate` to call a backend service. Since we have full control, we can do transformations, data aggregation, and so on. We can also use the Eureka APIs to get the server list, and implement completely independent load-balancing or traffic-shaping mechanisms instead of the out-of-the-box load balancing features provided by Ribbon.

High availability of Zuul

Zuul is just a stateless service with an HTTP endpoint, hence, we can have as many Zuul instances as we need. There is no affinity or stickiness required. However, the availability of Zuul is extremely critical as all traffic from the consumer to the provider flows through the Zuul proxy. However, the elastic scaling requirements are not as critical as the backend microservices where all the heavy lifting happens.

The high availability architecture of Zuul is determined by the scenario in which we are using Zuul. The typical usage scenarios are:

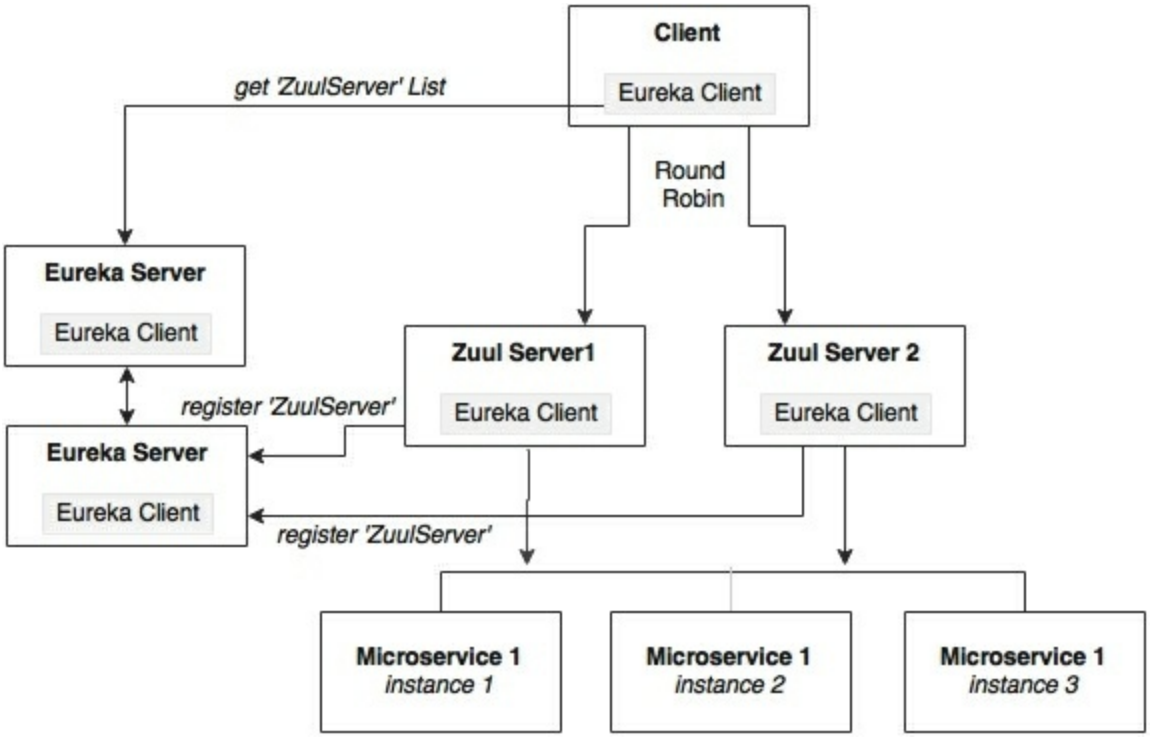
- When a client-side JavaScript MVC such as AngularJS accesses Zuul services from a remote browser.
- Another microservice or non-microservice accesses services via Zuul

In some cases, the client may not have the capabilities to use the Eureka client libraries, for example, a legacy application written on PL/SQL. In some cases, organization policies do not allow Internet clients to handle client-side load balancing. In the case of browser-based clients, there are third-party Eureka JavaScript libraries available.

It all boils down to whether the client is using Eureka client libraries or not. Based on this, there are two ways we can set up Zuul for high availability.

High availability of Zuul when the client is also a Eureka client

In this case, since the client is also another Eureka client, Zuul can be configured just like other microservices. Zuul registers itself to Eureka with a service ID. The clients then use Eureka and the service ID to resolve Zuul instances:



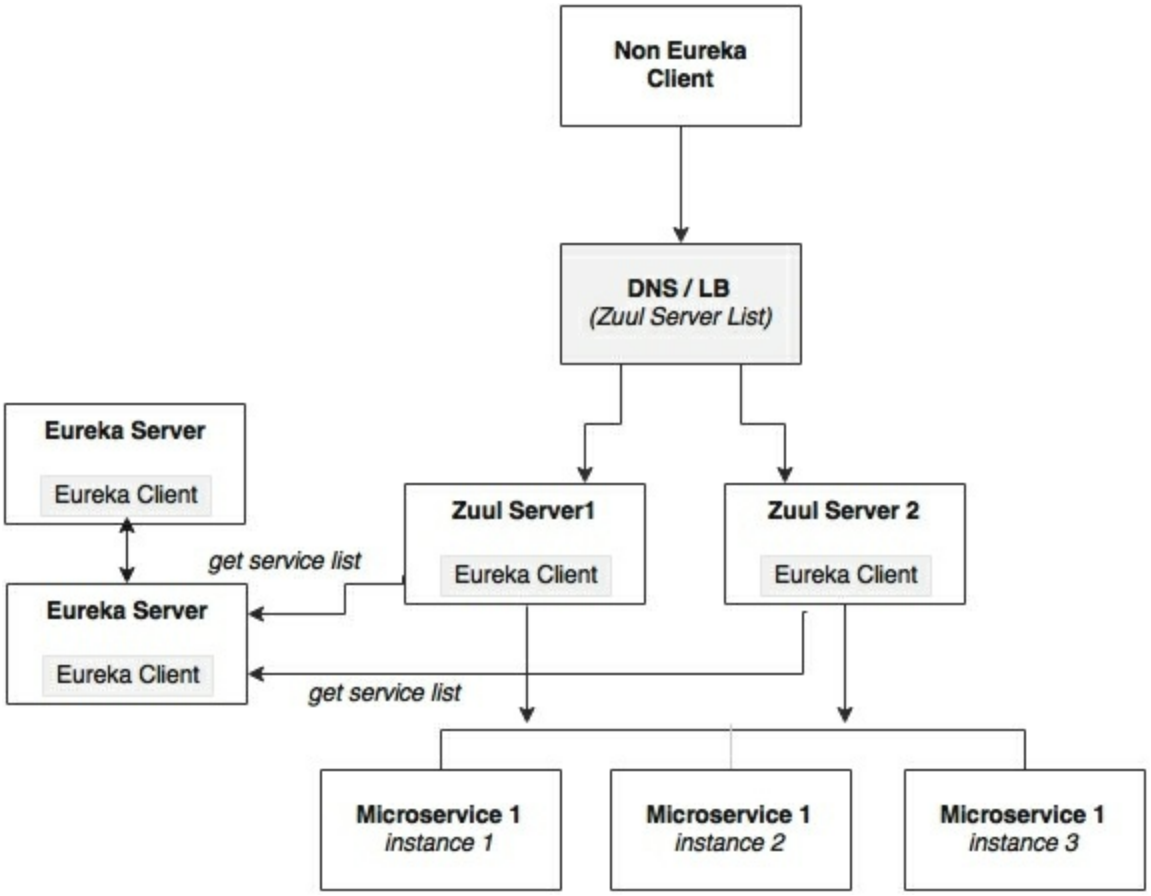
As shown in the preceding diagram, Zuul services register themselves with Eureka with a service ID, `search-apigateway` in our case. The Eureka client asks for the server list with the ID `search-apigateway`. The Eureka server returns the list of servers based on the current Zuul topology. The Eureka client, based on this list picks up one of the servers, and initiates the call.

As we saw earlier, the client uses the service ID to resolve the Zuul instance. In the following case, `search-apigateway` is the Zuul instance ID registered with Eureka:

```
Flight[] flights = searchClient.postForObject("http://search-
apigateway/api/search/get", searchQuery, Flight[].class);
```

High availability when the client is not a Eureka client

In this case, the client is not capable of handling load balancing by using the Eureka server. As shown in the following diagram, the client sends the request to a load balancer, which in turn identifies the right Zuul service instance. The Zuul instance, in this case, will be running behind a load balancer such as HAProxy or a hardware load balancer like NetScaler:



The microservices will still be load balanced by Zuul using the Eureka server.

Completing Zuul for all other services

In order to complete this exercise, add API gateway projects (name them as `*-apigateway`) for all our microservices. The following steps are required to achieve this task:

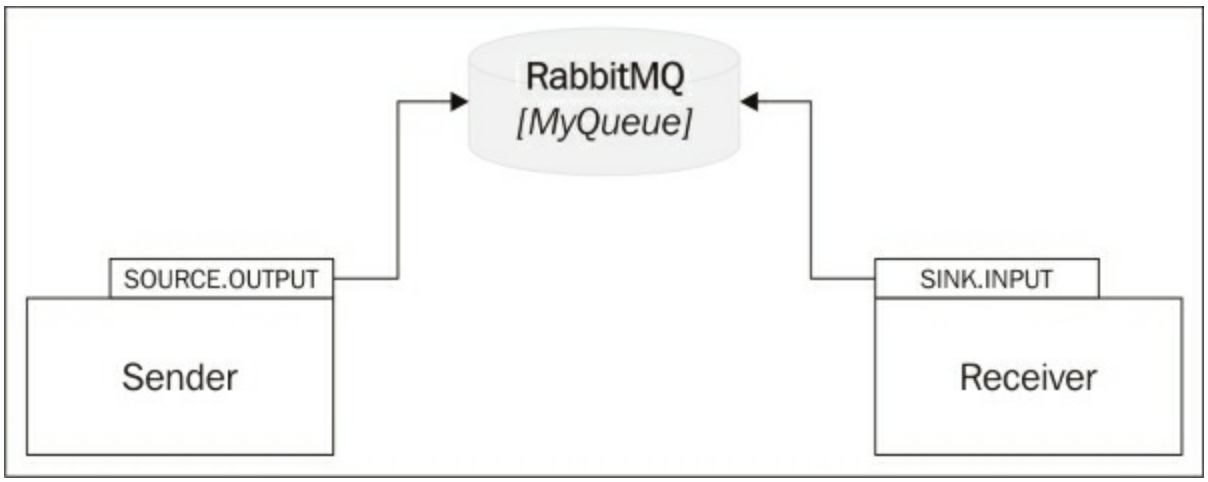
1. Create new property files per service, and check in to the Git repositories.
2. Change `application.properties` to `bootstrap.properties`, and add the required configurations.
3. Add `@EnableZuulProxy` to `Application.java` in each of the `*-apigateway` projects.
4. Add `@EnableDiscoveryClient` in all the `Application.java` files under each of the `*-apigateway` projects.
5. Optionally, change the package names and file names generated by default.

In the end, we will have the following API gateway projects:

- `chapter5.fares-apigateway`
- `chapter5.search-apigateway`
- `chapter5.checkin-apigateway`
- `chapter5.book-apigateway`

Streams for reactive microservices

Spring Cloud Stream provides an abstraction over the messaging infrastructure. The underlying messaging implementation can be RabbitMQ, Redis, or Kafka. Spring Cloud Stream provides a declarative approach for sending and receiving messages:

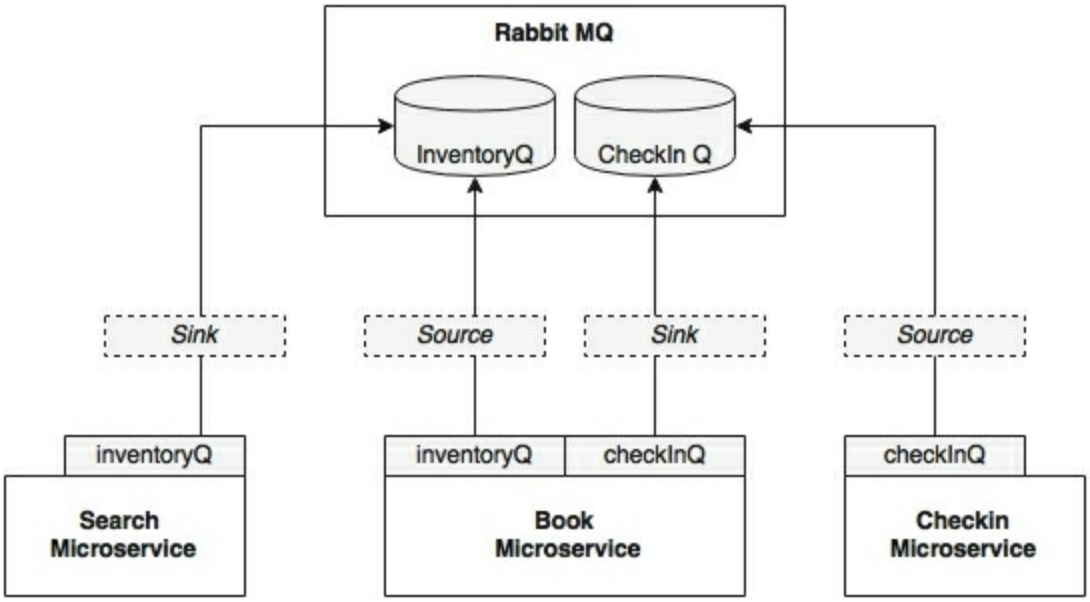


As shown in the preceding diagram, Cloud Stream works on the concept of a **source** and a sink. The source represents the sender perspective of the messaging, and sink represents the receiver perspective of the messaging.

In the example shown in the diagram, the sender defines a logical queue called `Source.OUTPUT` to which the sender sends messages. The receiver defines a logical queue called `Sink.INPUT` from which the receiver retrieves messages. The physical binding of `OUTPUT` to `INPUT` is managed through the configuration. In this case, both link to the same physical queue—`MyQueue` on RabbitMQ. So, while at one end, `Source.OUTPUT` points to `MyQueue`, on the other end, `Sink.INPUT` points to the same `MyQueue`.

Spring Cloud offers the flexibility to use multiple messaging providers in one application such as connecting an input stream from Kafka to a Redis output stream, without managing the complexities. Spring Cloud Stream is the basis for message-based integration. The Cloud Stream Modules subproject is another Spring Cloud library that provides many endpoint implementations.

As the next step, rebuild the inter-microservice messaging communication with the Cloud Streams. As shown in the next diagram, we will define a `SearchSink` connected to `InventoryQ` under the Search microservice. Booking will define a `BookingSource` for sending inventory change messages connected to `InventoryQ`. Similarly, Check-in defines a `CheckinSource` for sending the check-in messages. Booking defines a sink, `BookingSink`, for receiving messages, both bound to the `CheckinQ` queue on the RabbitMQ:



In this example, we will use RabbitMQ as the message broker:

1. Add the following Maven dependency to Booking, Search, and Check-in, as these are the three modules using messaging:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

2. Add the following two properties to `book-service.properties`. These properties bind the logical queue `inventoryQ` to physical `inventoryQ`, and the logical `checkinQ` to the physical `checkinQ`:

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

3. Add the following property to `search-service.properties`. This property binds the logical queue `inventoryQ` to the physical `inventoryQ`:

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
```

4. Add the following property to `checkin-service.properties`. This property binds the logical queue `checkinQ` to the physical `checkinQ`:

```
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

5. Commit all files to the Git repository.
6. The next step is to edit the code. The Search microservice consumes a message from the Booking microservice. In this case, Booking is the source and Search is the sink.

   Add `@EnableBinding` to the `Sender` class of the Booking service. This enables the Cloud Stream to work on autoconfigurations based on the message broker library available in the class

path. In our case, it is RabbitMQ. The parameter `BookingSource` defines the logical channels to be used for this configuration:

```
@EnableBinding(BookingSource.class)
public class Sender {
```

7. In this case, `BookingSource` defines a message channel called `inventoryQ`, which is physically bound to RabbitMQ's `inventoryQ`, as configured in the configuration. `BookingSource` uses an annotation, `@Output`, to indicate that this is of the output type—a message that is outgoing from a module. This information will be used for autoconfiguration of the message channel:

```
interface BookingSource {
    public static String InventoryQ="inventoryQ";
    @Output("inventoryQ")
    public MessageChannel inventoryQ();
}
```

8. Instead of defining a custom class, we can also use the default `Source` class that comes with Spring Cloud Stream if the service has only one source and sink:

```
public interface Source {
  @Output("output")
  MessageChannel output();
}
```

9. Define a message channel in the sender, based on `BookingSource`. The following code will inject an output message channel with the name `inventory`, which is already configured in `BookingSource`:

```
  @Output (BookingSource.InventoryQ)
  @Autowired
  private MessageChannel;
```

10. Reimplement the `send` message method in `BookingSender`:

```
public void send(Object message){
  messageChannel.
    send(MessageBuilder.withPayload(message).
    build());
}
```

11. Now add the following to the `SearchReceiver` class the same way we did for the Booking service:

```
@EnableBinding(SearchSink.class)
public class Receiver {
```

12. In this case, the `SearchSink` interface will look like the following. This will define the logical sink queue it is connected with. The message channel in this case is defined as `@Input` to indicate that this message channel is to accept messages:

```
interface SearchSink {
    public static String INVENTORYQ="inventoryQ";
    @Input("inventoryQ")
```

```
    public MessageChannel inventoryQ();
}
```

13. Amend the Search service to accept this message:

```
@ServiceActivator(inputChannel = SearchSink.INVENTORYQ)
public void accept(Map<String,Object> fare){
        searchComponent.updateInventory((String)fare.
        get("FLIGHT_NUMBER"),(String)fare.
        get("FLIGHT_DATE"),(int)fare.
        get("NEW_INVENTORY"));
}
```

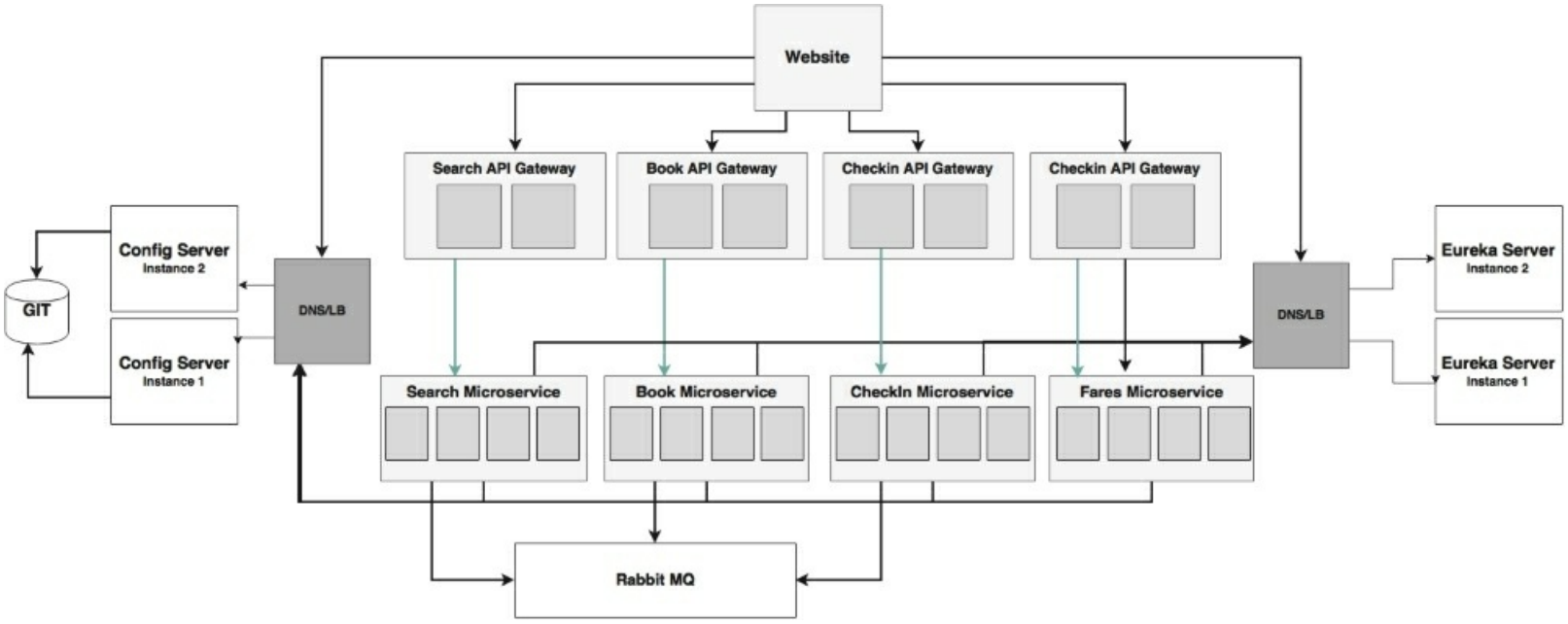
14. We will still need the RabbitMQ configurations that we have in our configuration files to connect to the message broker:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
server.port=8090
```

15. Run all services, and run the website project. If everything is fine, the website project successfully executes the Search, Booking, and Check-in functions. The same can also be tested using the browser by pointing to `http://localhost:8001`.

Summarizing the BrownField PSS architecture

The following diagram shows the overall architecture that we have created with the Config server, Eureka, Feign, Zuul, and Cloud Streams. The architecture also includes the high availability of all components. In this case, we assume that the client uses the Eureka client libraries:



The summary of the projects and the port they are listening on is given in the following table:

Microservice	**Projects**	**Port**
Book microservice	`chapter5.book`	`8060` to `8064`
Check-in microservice	`chapter5.checkin`	`8070` to `8074`
Fare microservice	`chapter5.fares`	`8080` to `8084`
Search microservice	`chapter5.search`	`8090` to `8094`
Website client	`chapter5.website`	`8001`
Spring Cloud Config server	`chapter5.configserver`	`8888/8889`
Spring Cloud Eureka server	`chapter5.eurekaserver`	`8761/8762`
Book API gateway	`chapter5.book-apigateway`	`8095` to `8099`

Check-in API gateway	`chapter5.checkin-apigateway`	`8075` to `8079`
Fares API gateway	`chapter5.fares-apigateway`	`8085` to `8089`
Search API gateway	`chapter5.search-apigateway`	`8065` to `8069`

Follow these steps to do a final run:

1. Run RabbitMQ.
2. Build all projects using `pom.xml` at the root level:

```
mvn -Dmaven.test.skip=true clean install
```

3. Run the following projects from their respective folders. Remember to wait for 40 to 50 seconds before starting the next service. This will ensure that the dependent services are registered and are available before we start a new service:

```
java -jar target/fares-1.0.jar
java -jar target/search-1.0.jar
java -jar target/checkin-1.0.jar
java -jar target/book-1.0.jar
java -jar target/fares-apigateway-1.0.jar
java -jar target/search-apigateway-1.0.jar
java -jar target/checkin-apigateway-1.0.jar
java -jar target/book-apigateway-1.0.jar
java -jar target/website-1.0.jar
```

4. Open the browser window, and point to `http://localhost:8001`. Follow the steps mentioned in the *Running and testing the project* section in Chapter 4, *Microservices Evolution – A Case Study*.

Summary

In this chapter, you learned how to scale a Twelve-Factor Spring Boot microservice using the Spring Cloud project. What you learned was then applied to the BrownField Airline's PSS microservice that we developed in the previous chapter.

We then explored the Spring Config server for externalizing the microservices' configuration, and the way to deploy the Config server for high availability. We also discussed the declarative service calls using Feign, examined the use of Ribbon and Eureka for load balancing, dynamic service registration, and discovery. Implementation of an API gateway was examined by implementing Zuul. Finally, we concluded with a reactive style integration of microservices using Spring Cloud Stream.

BrownField Airline's PSS microservices are now deployable on the Internet scale. Other Spring Cloud components such as Hyterix, Sleuth, and so on will be covered in Chapter 7, *Logging and Monitoring Microservices*. The next chapter will demonstrate autoscaling features, extending the BrownField PSS implementation.

Chapter 6. Autoscaling Microservices

Spring Cloud provides the support essential for the deployment of microservices at scale. In order to get the full power of a cloud-like environment, the microservices instances should also be capable of scaling out and shrinking automatically based on traffic patterns.

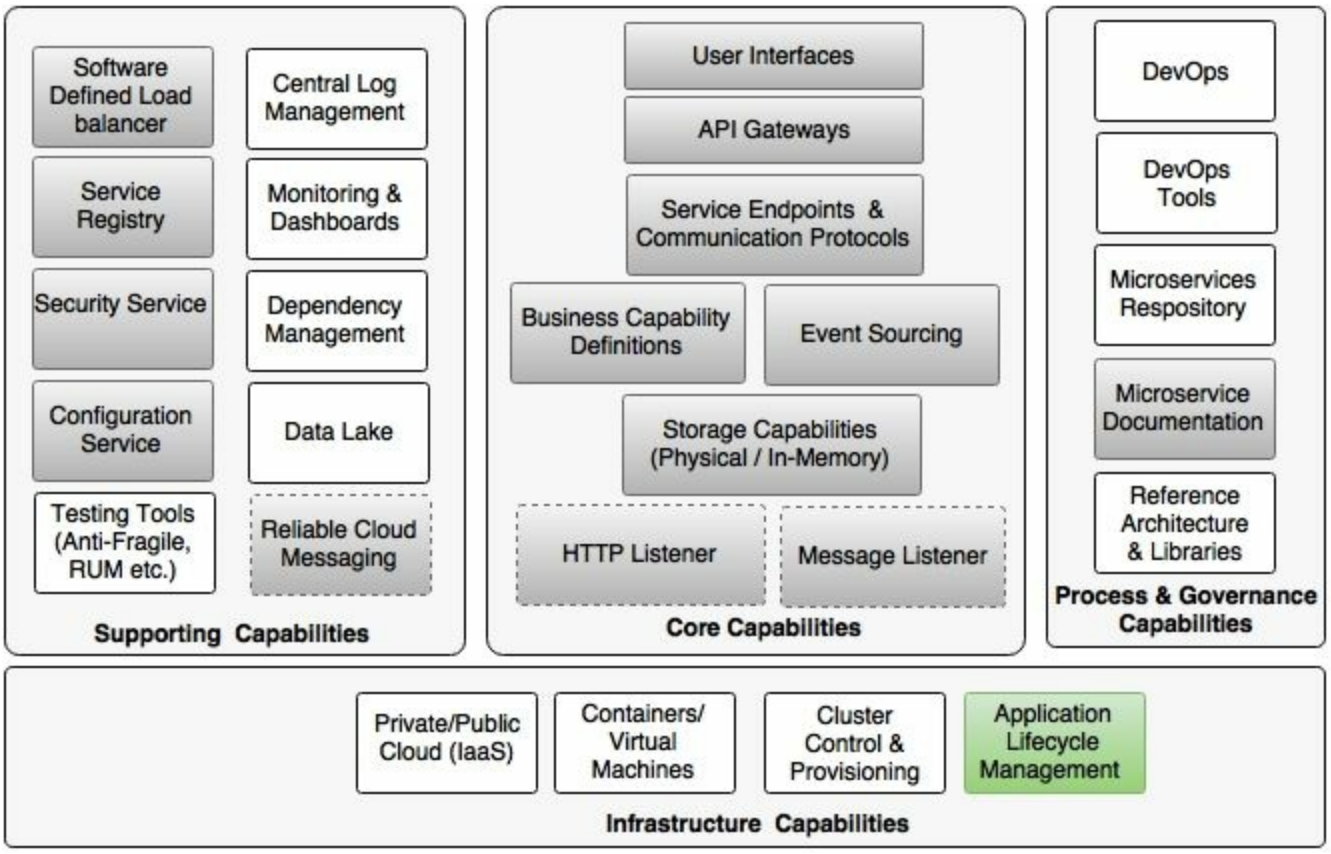
This chapter will detail out how to make microservices elastically grow and shrink by effectively using the actuator data collected from Spring Boot microservices to control the deployment topology by implementing a simple life cycle manager.

By the end of this chapter, you will learn about the following topics:

- The basic concept of autoscaling and different approaches for autoscaling
- The importance and capabilities of a life cycle manager in the context of microservices
- Examining the custom life cycle manager to achieve autoscaling
- Programmatically collecting statistics from the Spring Boot actuator and using it to control and shape incoming traffic

Reviewing the microservice capability model

This chapter will cover the **Application Lifecycle Management** capability in the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*, highlighted in the following diagram:



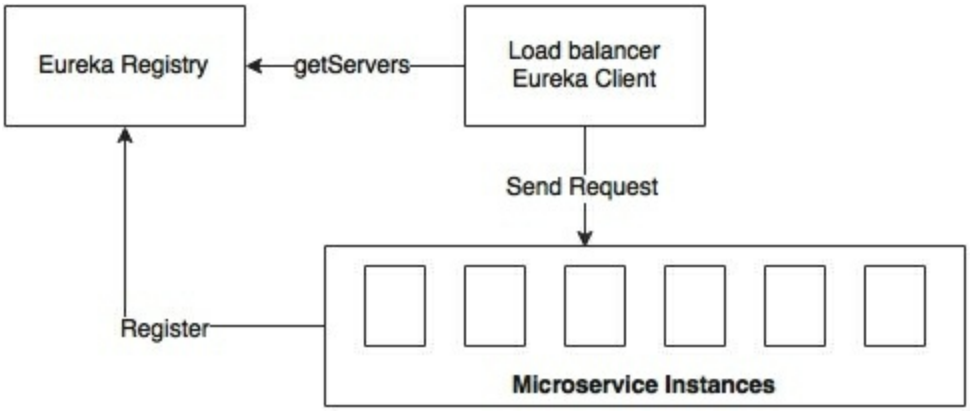
We will see a basic version of the life cycle manager in this chapter, which will be enhanced in later chapters.

Scaling microservices with Spring Cloud

In Chapter 5, *Scaling Microservices with Spring Cloud*, you learned how to scale Spring Boot microservices using Spring Cloud components. The two key concepts of Spring Cloud that we implemented are self-registration and self-discovery. These two capabilities enable automated microservices deployments. With self-registration, microservices can automatically advertise the service availability by registering service metadata to a central service registry as soon as the instances are ready to accept traffic. Once the microservices are registered, consumers can consume the newly registered services from the very next moment by discovering service instances using the registry service. Registry is at the heart of this automation.

This is quite different from the traditional clustering approach employed by the traditional JEE application servers. In the case of JEE application servers, the server instances' IP addresses are more or less statically configured in a load balancer. Therefore, the cluster approach is not the best solution for automatic scaling in Internet-scale deployments. Also, clusters impose other challenges, such as they have to have exactly the same version of binaries on all cluster nodes. It is also possible that the failure of one cluster node can poison other nodes due to the tight dependency between nodes.

The registry approach decouples the service instances. It also eliminates the need to manually maintain service addresses in the load balancer or configure virtual IPs:



As shown in the diagram, there are three key components in our automated microservices deployment topology:

- **Eureka** is the central registry component for microservice registration and discovery. REST APIs are used by both consumers as well as providers to access the registry. The registry also holds the service metadata such as the service identity, host, port, health status, and so on.
- The **Eureka** client, together with the **Ribbon** client, provide client-side dynamic load balancing. Consumers use the Eureka client to look up the Eureka server to identify the available instances of a target service. The Ribbon client uses this server list to load-balance between the available microservice instances. In a similar way, if the service instance goes out of service, these instances will be taken out of the Eureka registry. The load balancer automatically reacts to these

dynamic topology changes.
- The third component is the **microservices** instances developed using Spring Boot with the actuator endpoints enabled.

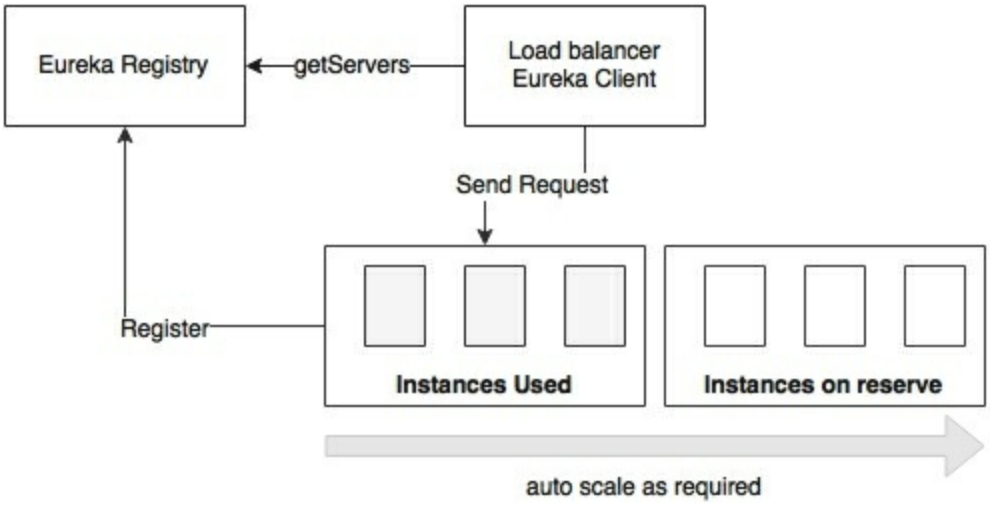
However, there is one gap in this approach. When there is need for an additional microservice instance, a manual task is required to kick off a new instance. In an ideal scenario, the starting and stopping of microservice instances also require automation.

For example, when there is a requirement to add another Search microservice instance to handle the increase in traffic volumes or a load burst scenario, the administrator has to manually bring up a new instance. Also, when the Search instance is idle for some time, it needs to be manually taken out of service to have optimal infrastructure usage. This is especially relevant when services run on a pay-as-per-usage cloud environment.

Understanding the concept of autoscaling

Autoscaling is an approach to automatically scaling out instances based on the resource usage to meet the SLAs by replicating the services to be scaled.

The system automatically detects an increase in traffic, spins up additional instances, and makes them available for traffic handling. Similarly, when the traffic volumes go down, the system automatically detects and reduces the number of instances by taking active instances back from the service:



As shown in the preceding diagram, autoscaling is done, generally, using a set of reserve machines.

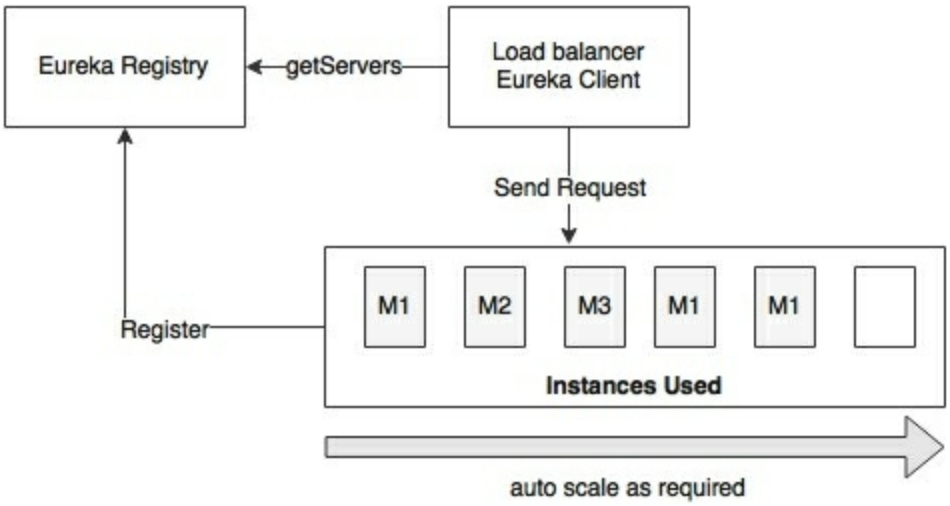
As many of the cloud subscriptions are based on a pay-as-you-go model, this is an essential capability when targeting cloud deployments. This approach is often called **elasticity**. It is also called **dynamic resource provisioning and deprovisioning**. Autoscaling is an effective approach specifically for microservices with varying traffic patterns. For example, an Accounting service would have high traffic during month ends and year ends. There is no point in permanently provisioning instances to handle these seasonal loads.

In the autoscaling approach, there is often a resource pool with a number of spare instances. Based on the demand, instances will be moved from the resource pool to the active state to meet the surplus demand. These instances are not pretagged for any particular microservices or prepackaged with any of the microservice binaries. In advanced deployments, the Spring Boot binaries are downloaded on demand from an artifact repository such as Nexus or Artifactory.

The benefits of autoscaling

There are many benefits in implementing the autoscaling mechanism. In traditional deployments, administrators reserve a set of servers against each application. With autoscaling, this preallocation is no longer required. This prefixed server allocation may result in underutilized servers. In this case, idle servers cannot be utilized even when neighboring services struggle for additional resources.

With hundreds of microservice instances, preallocating a fixed number of servers to each of the microservices is not cost effective. A better approach is to reserve a number of server instances for a group of microservices without preallocating or tagging them against a microservice. Instead, based on the demand, a group of services can share a set of available resources. By doing so, microservices can be dynamically moved across the available server instances by optimally using the resources:



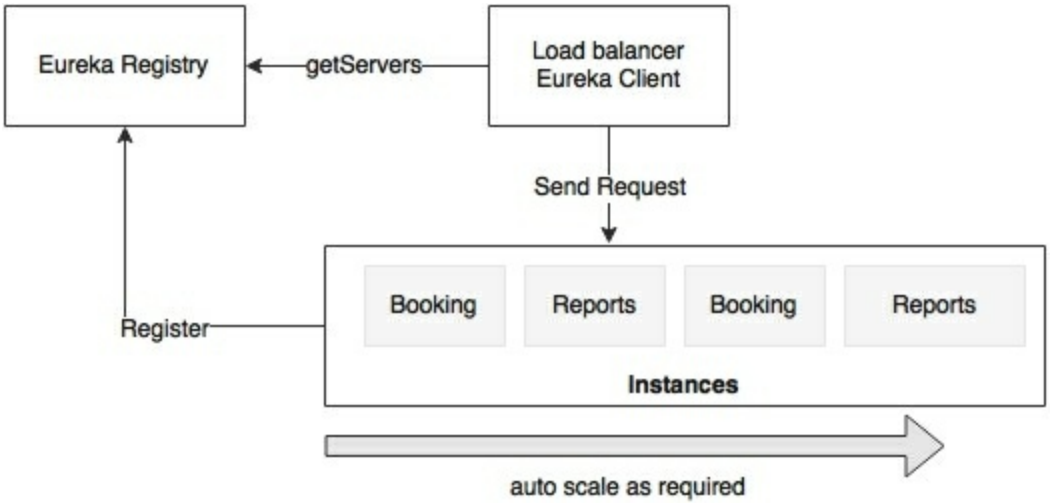
As shown in the preceding diagram, there are three instances of the **M1** microservice, one instance of **M2**, and one instance of **M3** up and running. There is another server kept unallocated. Based on the demand, the unallocated server can be used for any of the microservices: **M1**, **M2**, or **M3**. If **M1** has more service requests, then the unallocated instance will be used for **M1**. When the service usage goes down, the server instance will be freed up and moved back to the pool. Later, if the **M2** demand increases, the same server instance can be activated using **M2**.

Some of the key benefits of autoscaling are:

- **It has high availability and is fault tolerant**: As there are multiple service instances, even if one fails, another instance can take over and continue serving clients. This failover will be transparent to the consumers. If no other instance of this service is available, the autoscaling service will recognize this situation and bring up another server with the service instance. As the whole process of bringing up or bringing down instances is automatic, the overall availability of the services will be higher than the systems implemented without autoscaling. The systems without autoscaling require manual intervention to add or remove service instances, which will be hard to manage in large deployments.

For example, assume that two of instances of the Booking service are running. If there is an increase in the traffic flow, in a normal scenario, the existing instance might become overloaded. In most of the scenarios, the entire set of services will be jammed, resulting in service unavailability. In the case of autoscaling, a new Booking service instance can be brought up quickly. This will balance the load and ensure service availability.

- **It increases scalability**: One of the key benefits of autoscaling is horizontal scalability. Autoscaling allows us to selectively scale up or scale down services automatically based on traffic patterns.
- **It has optimal usage and is cost saving**: In a pay-as-you-go subscription model, billing is based on actual resource utilization. With the autoscaling approach, instances will be started and shut down based on the demand. Hence, resources are optimally utilized, thereby saving cost.
- **It gives priority to certain services or group of services**: With autoscaling, it is possible to give priority to certain critical transactions over low-value transactions. This will be done by removing an instance from a low-value service and reallocating it to a high-value service. This will also eliminate situations where a low-priority transaction heavily utilizes resources when high-value transactions are cramped up for resources.



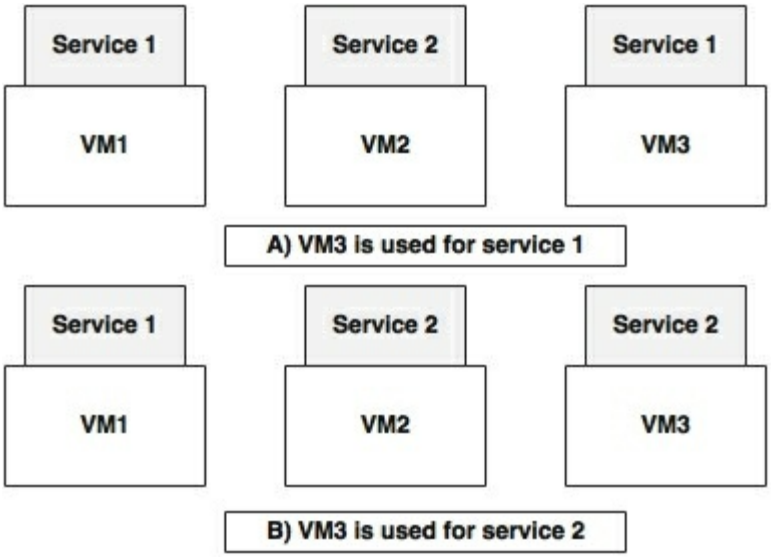
For instance, the **Booking** and **Reports** services run with two instances, as shown in the preceding diagram. Let's assume that the **Booking** service is a revenue generation service and therefore has a higher value than the **Reports** service. If there are more demands for the **Booking** service, then one can set policies to take one **Reports** service out of the service and release this server for the **Booking** service.

Different autoscaling models

Autoscaling can be applied at the application level or at the infrastructure level. In a nutshell, application scaling is scaling by replicating application binaries only, whereas infrastructure scaling is replicating the entire virtual machine, including application binaries.

Autoscaling an application

In this scenario, scaling is done by replicating the microservices, not the underlying infrastructure, such as virtual machines. The assumption is that there is a pool of VMs or physical infrastructures available to scale up microservices. These VMs have the basic image fused with any dependencies, such as JRE. It is also assumed that microservices are homogeneous in nature. This gives flexibility in reusing the same virtual or physical machines for different services:

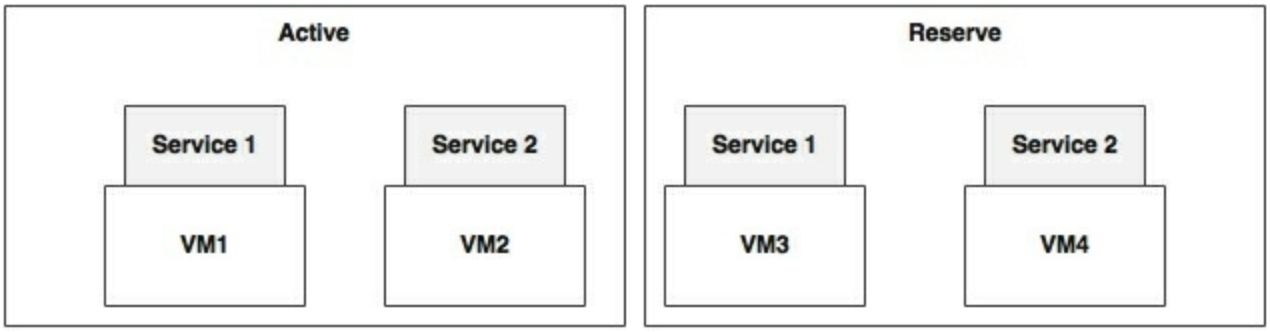


As shown in the preceding diagram, in scenario **A**, **VM3** is used for **Service 1**, whereas in scenario **B**, the same **VM3** is used for **Service 2**. In this case, we only swapped the application library and not the underlying infrastructure.

This approach gives faster instantiation as we are only handling the application binaries and not the underlying VMs. The switching is easier and faster as the binaries are smaller in size and there is no OS boot required either. However, the downside of this approach is that if certain microservices require OS-level tuning or use polyglot technologies, then dynamically swapping microservices will not be effective.

Autoscaling the infrastructure

In contrast to the previous approach, in this case, the infrastructure is also provisioned automatically. In most cases, this will create a new VM on the fly or destroy the VMs based on the demand:



As shown in the preceding diagram, the reserve instances are created as VM images with predefined service instances. When there is demand for **Service 1**, **VM3** is moved to an active state. When there is a demand for **Service 2**, **VM4** is moved to the active state.

This approach is efficient if the applications depend upon the parameters and libraries at the infrastructure level, such as the operating system. Also, this approach is better for polyglot microservices. The downside is the heavy nature of VM images and the time required to spin up a new VM. Lightweight containers such as Dockers are preferred in such cases instead of traditional heavyweight virtual machines.

Autoscaling in the cloud

Elasticity or autoscaling is one of the fundamental features of most cloud providers. Cloud providers use infrastructure scaling patterns, as discussed in the previous section. These are typically based on a set of pooled machines.

For example, in AWS, these are based on introducing new EC2 instances with a predefined AMI. AWS supports autoscaling with the help of autoscaling groups. Each group is set with a minimum and maximum number of instances. AWS ensures that the instances are scaled on demand within these bounds. In case of predictable traffic patterns, provisioning can be configured based on timelines. AWS also provides ability for applications to customize autoscaling policies.

Microsoft Azure also supports autoscaling based on the utilization of resources such as the CPU, message queue length, and so on. IBM Bluemix supports autoscaling based on resources such as CPU usage.

Other PaaS platforms, such as CloudBees and OpenShift, also support autoscaling for Java applications. Pivotal Cloud Foundry supports autoscaling with the help of Pivotal Autoscale. Scaling policies are generally based on resource utilization, such as the CPU and memory thresholds.

There are components that run on top of the cloud and provide fine-grained controls to handle autoscaling. Netflix Fenzo, Eucalyptus, Boxfuse, and Mesosphere are some of the components in this category.

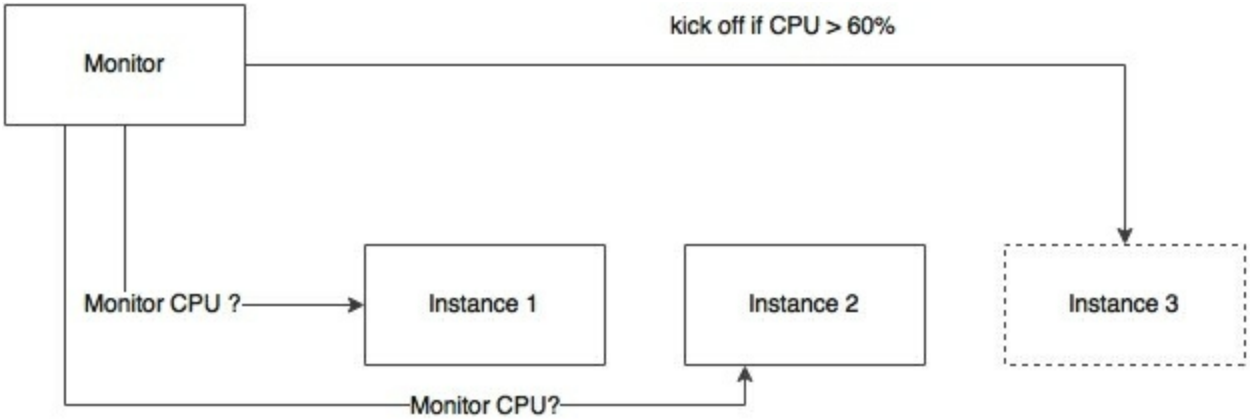
Autoscaling approaches

Autoscaling is handled by considering different parameters and thresholds. In this section, we will discuss the different approaches and policies that are typically applied to take decisions on when to scale up or down.

Scaling with resource constraints

This approach is based on real-time service metrics collected through monitoring mechanisms. Generally, the resource-scaling approach takes decisions based on the CPU, memory, or the disk of machines. This can also be done by looking at the statistics collected on the service instances themselves, such as heap memory usage.

A typical policy may be spinning up another instance when the CPU utilization of the machine goes beyond 60%. Similarly, if the heap size goes beyond a certain threshold, we can add a new instance. The same applies to downsizing the compute capacity when the resource utilization goes below a set threshold. This is done by gradually shutting down servers:



In typical production scenarios, the creation of additional services is not done on the first occurrence of a threshold breach. The most appropriate approach is to define a sliding window or a waiting period.

The following are some of the examples:

- An example of a **response sliding window** is if 60% of the response time of a particular transaction is consistently more than the set threshold value in a 60-second sampling window, increase service instances
- In a **CPU sliding window**, if the CPU utilization is consistently beyond 70% in a 5 minutes sliding window, then a new instance is created
- An example of the **exception sliding window** is if 80% of the transactions in a sliding window of 60 seconds or 10 consecutive executions result in a particular system exception, such as a connection timeout due to exhausting the thread pool, then a new service instance is created

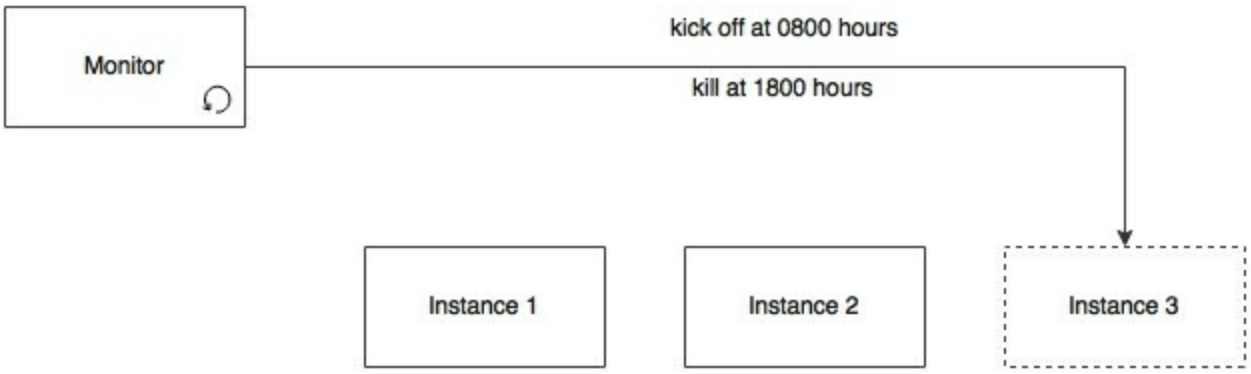
In many cases, we will set a lower threshold than the actual expected thresholds. For example, instead of setting the CPU utilization threshold at 80%, set it at 60% so that the system gets enough time to spin up an instance before it stops responding. Similarly, when scaling down, we use a lower threshold than the actual. For example, we will use 40% CPU utilization to scale down instead of 60%. This allows us to have a cool-down period so that there will not be any resource struggle when

shutting down instances.

Resource-based scaling is also applicable to service-level parameters such as the throughput of the service, latency, applications thread pool, connection pool, and so on. These can also be at the application level, such as the number of **sales orders** processing in a service instance, based on internal benchmarking.

Scaling during specific time periods

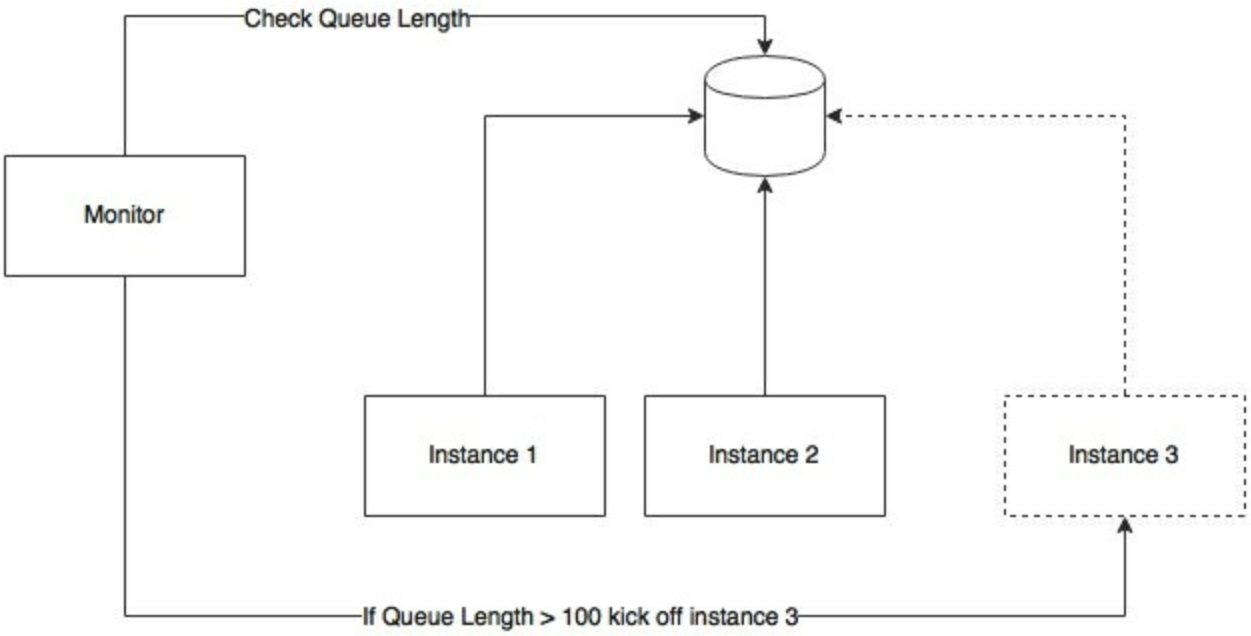
Time-based scaling is an approach to scaling services based on certain periods of the day, month, or year to handle seasonal or business peaks. For example, some services may experience a higher number of transactions during office hours and a considerably low number of transactions outside office hours. In this case, during the day, services autoscale to meet the demand and automatically downsize during the non-office hours:



Many airports worldwide impose restrictions on night-time landing. As a result, the number of passengers checking in at the airports during the night time is less compared to the day time. Hence, it is cost effective to reduce the number of instances during the night time.

Scaling based on the message queue length

This is particularly useful when the microservices are based on asynchronous messaging. In this approach, new consumers are automatically added when the messages in the queue go beyond certain limits:



This approach is based on the competing consumer pattern. In this case, a pool of instances is used to consume messages. Based on the message threshold, new instances are added to consume additional messages.

Scaling based on business parameters

In this case, adding instances is based on certain business parameters—for example, spinning up a new instance just before handling **sales closing** transactions. As soon as the monitoring service receives a preconfigured business event (such as **sales closing minus 1 hour**), a new instance will be brought up in anticipation of large volumes of transactions. This will provide fine-grained control on scaling based on business rules:



Predictive autoscaling

Predictive scaling is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling. A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns. Autoscaling is done based on these predictions. Predictive autoscaling helps avoid hardcoded rules and time windows. Instead, the system can automatically predict such time windows. In more sophisticated deployments, predictive analysis may use cognitive computing mechanisms to predict autoscaling.

In the cases of sudden traffic spikes, traditional autoscaling may not help. Before the autoscaling component can react to the situation, the spike would have hit and damaged the system. The predictive system can understand these scenarios and predict them before their actual occurrence. An example will be handling a flood of requests immediately after a planned outage.

Netflix Scryer is an example of such a system that can predict resource requirements in advance.

Autoscaling BrownField PSS microservices

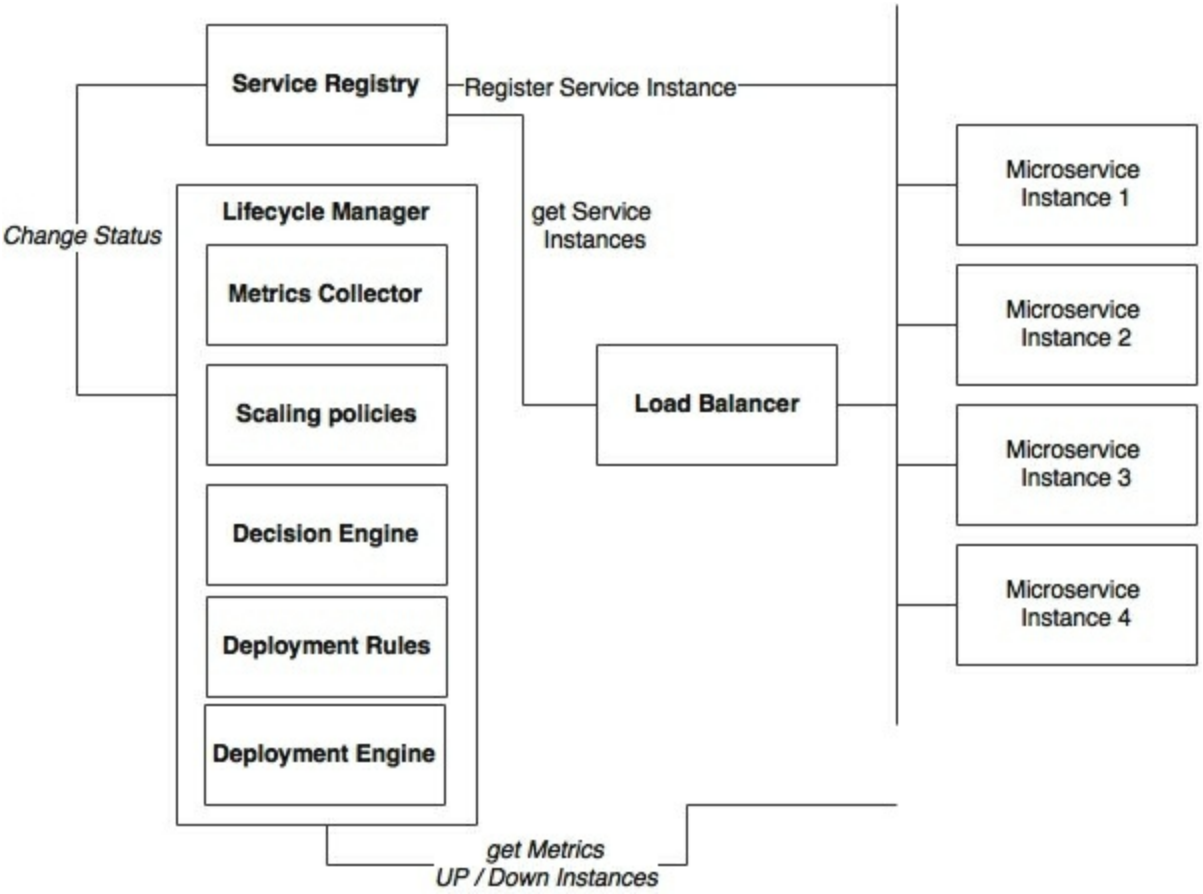
In this section, we will examine how to enhance microservices developed in Chapter 5, *Scaling Microservices with Spring Cloud*, for autoscaling. We need a component to monitor certain performance metrics and trigger autoscaling. We will call this component the **life cycle manager**.

The service life cycle manager, or the application life cycle manager, is responsible for detecting scaling requirements and adjusting the number of instances accordingly. It is responsible for starting and shutting down instances dynamically.

In this section, we will take a look at a primitive autoscaling system to understand the basic concepts, which will be enhanced in later chapters.

The capabilities required for an autoscaling system

A typical autoscaling system has capabilities as shown in the following diagram:



The components involved in the autoscaling ecosystem in the context of microservices are explained as follows:

- **Microservices**: These are sets of the up-and-running microservice instances that keep sending health and metrics information. Alternately, these services expose actuator endpoints for metrics collection. In the preceding diagram, these are represented as **Microservice 1** through **Microservice 4**.
- **Service Registry**: A service registry keeps track of all the services, their health states, their metadata, and their endpoint URI.
- **Load Balancer**: This is a client-side load balancer that looks up the service registry to get up-to-date information about the available service instances.
- **Lifecycle Manager**: The life cycle manger is responsible for autoscaling, which has the following subcomponents:
  - **Metrics Collector**: A metrics collection unit is responsible for collecting metrics from all service instances. The life cycle manager will aggregate the metrics. It may also keep a sliding time window. The metrics could be infrastructure-level metrics, such as CPU usage, or application-level metrics, such as transactions per minute.
  - **Scaling policies**: Scaling policies are nothing but sets of rules indicating when to scale up

and scale down microservices—for example, 90% of CPU usage above 60% in a sliding window of 5 minutes.

- **Decision Engine**: A decision engine is responsible for making decisions to scale up and scale down based on the aggregated metrics and scaling policies.
- **Deployment Rules**: The deployment engine uses deployment rules to decide which parameters to consider when deploying services. For example, a service deployment constraint may say that the instance must be distributed across multiple availability regions or a 4 GB minimum of memory required for the service.
- **Deployment Engine**: The deployment engine, based on the decisions of the decision engine, can start or stop microservice instances or update the registry by altering the health states of services. For example, it sets the health status as "out of service" to take out a service temporarily.

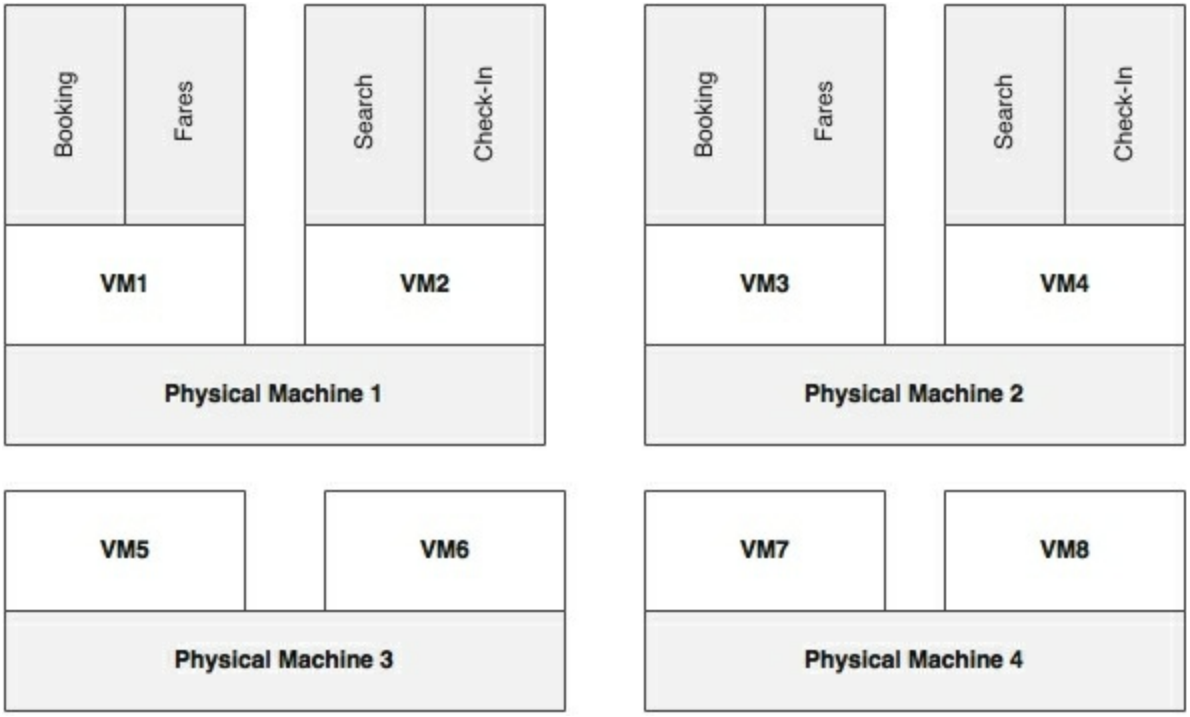
Implementing a custom life cycle manager using Spring Boot

The life cycle manager introduced in this section is a minimal implementation to understand autoscaling capabilities. In later chapters, we will enhance this implementation with containers and cluster management solutions. Ansible, Marathon, and Kubernetes are some of the tools useful in building this capability.

In this section, we will implement an application-level autoscaling component using Spring Boot for the services developed in Chapter 5, *Scaling Microservices with Spring Cloud*.

Understanding the deployment topology

The following diagram shows a sample deployment topology of BrownField PSS microservices:



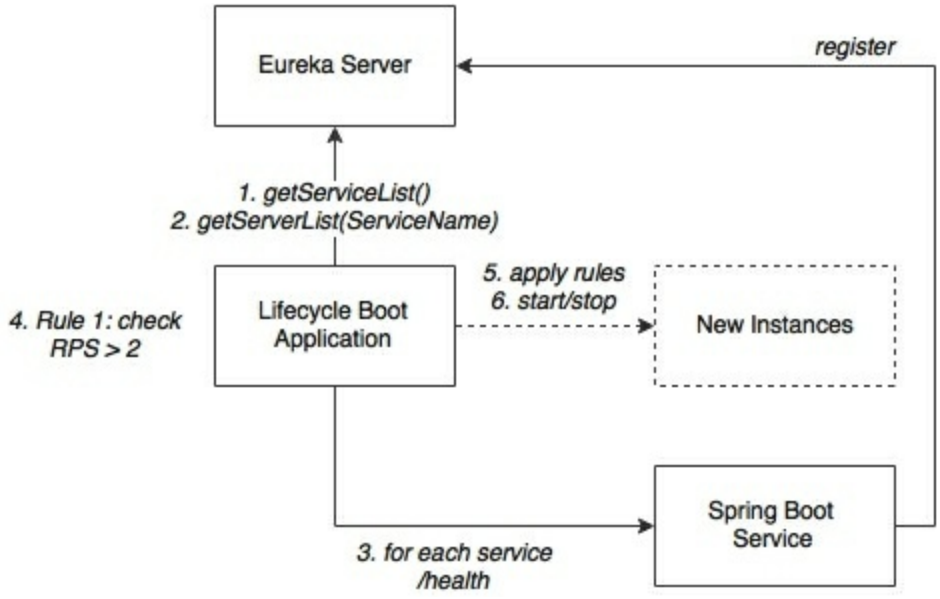
As shown in the diagram, there are four physical machines. Eight VMs are created from four physical machines. Each physical machine is capable of hosting two VMs, and each VM is capable of running two Spring Boot instances, assuming that all services have the same resource requirements.

Four VMs, **VM1** through **VM4**, are active and are used to handle traffic. **VM5** to **VM8** are kept as reserve VMs to handle scalability. **VM5** and **VM6** can be used for any of the microservices and can also be switched between microservices based on scaling demands. Redundant services use VMs created from different physical machines to improve fault tolerance.

Our objective is to scale out any services when there is increase in traffic flow using four VMs, **VM5** through **VM8**, and scale down when there is not enough load. The architecture of our solution is as follows.

Understanding the execution flow

Have a look at the following flowchart:



As shown in the preceding diagram, the following activities are important for us:

- The Spring Boot service represents microservices such as Search, Book, Fares, and Check-in. Services at startup automatically register endpoint details to the Eureka registry. These services are actuator-enabled, so the life cycle manager can collect metrics from the actuator endpoints.
- The life cycle manager service is nothing but another Spring Boot application. The life cycle manager has a metrics collector that runs a background job, periodically polls the Eureka server, and gets details of all the service instances. The metrics collector then invokes the actuator endpoints of each microservice registered in the Eureka registry to get the health and metrics information. In a real production scenario, a subscription approach for data collection is better.
- With the collected metrics information, the life cycle manager executes a list of policies and derives decisions on whether to scale up or scale down instances. These decisions are either to start a new service instance of a particular type on a particular VM or to shut down a particular instance.
- In the case of shutdown, it connects to the server using an actuator endpoint and calls the shutdown service to gracefully shut down an instance.
- In the case of starting a new instance, the deployment engine of the life cycle manager uses the scaling rules and decides where to start the new instance and what parameters are to be used when starting the instance. Then, it connects to the respective VMs using SSH. Once connected, it executes a preinstalled script (or passes this script as a part of the execution) by passing the required constraints as a parameter. This script fetches the application library from a central Nexus repository in which the production binaries are kept and initiates it as a Spring Boot application. The port number is parameterized by the life cycle manager. SSH needs to be enabled on the target machines.

In this example, we will use **TPM** (**Transactions Per Minute**) or **RPM** (**Requests Per Minute**) as sampler metrics for decision making. If the Search service has more than 10 TPM, then it will spin up a new Search service instance. Similarly, if the TPM is below 2, one of the instances will be shut down and released back to the pool.

When starting a new instance, the following policies will be applied:

- The number of service instances at any point should be a minimum of 1 and a maximum of 4. This also means that at least one service instance will always be up and running.
- A scaling group is defined in such a way that a new instance is created on a VM that is on a different physical machine. This will ensure that the services run across different physical machines.

These policies could be further enhanced. The life cycle manager ideally provides options to customize these rules through REST APIs or Groovy scripts.

A walkthrough of the life cycle manager code

We will take a look at how a simple life cycle manager is implemented. This section will be a walkthrough of the code to understand the different components of the life cycle manager.

Tip

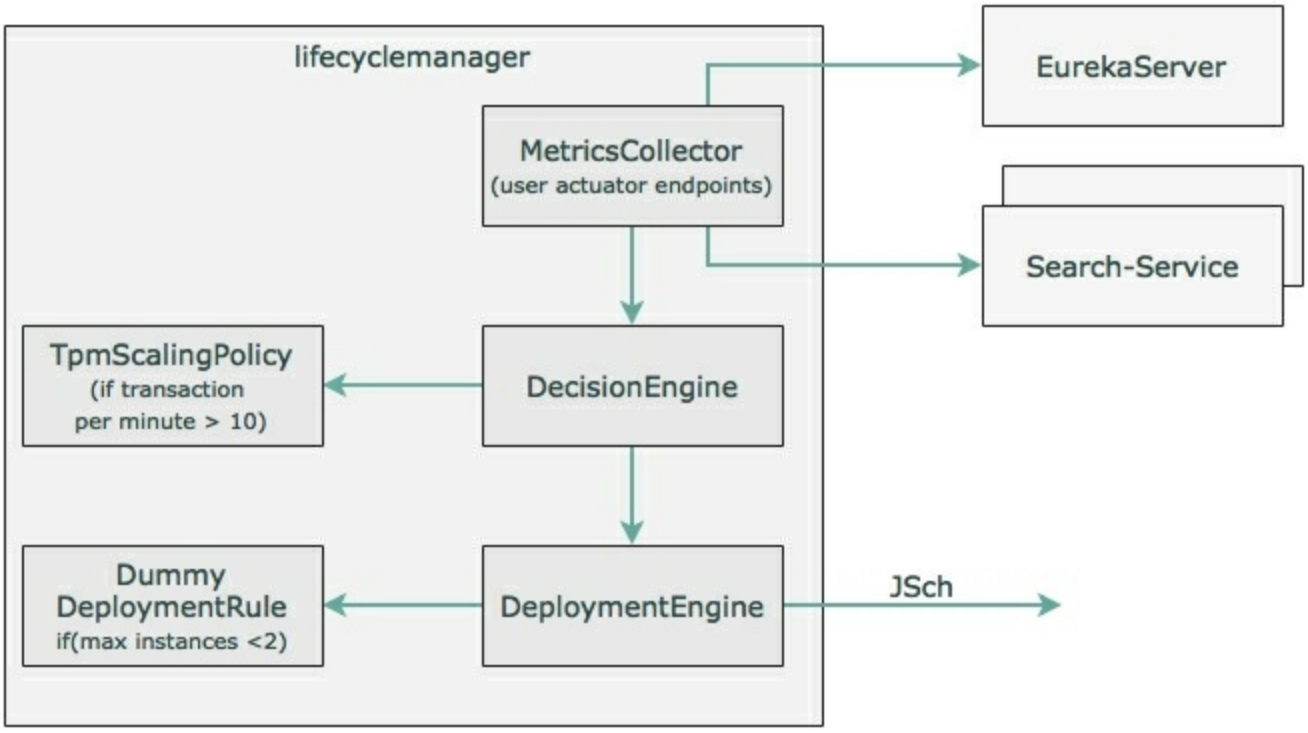
The full source code is available under the `Chapter 6` project in the code files. The `chapter5.configserver`, `chapter5.eurekaserver`, `chapter5.search`, and `chapter5.search-apigateway` are copied and renamed as `chapter6.*`, respectively.

Perform the following steps to implement the custom life cycle manager:

1. Create a new Spring Boot application and name it `chapter6.lifecyclemanager`. The project structure is shown in the following diagram:



   The flowchart for this example is as shown in the following diagram:



The components of this diagram are explained in details here.

2. Create a `MetricsCollector` class with the following method. At the startup of the Spring Boot application, this method will be invoked using `CommandLineRunner`, as follows:

```
public void start(){
  while(true){
    eurekaClient.getServices().forEach(service -> {
System.out.println("discovered service "+ service);
      Map metrics =
restTemplate.getForObject("http://"+service+"/metrics",Map.class);
      decisionEngine.execute(service, metrics);
    });
  }
}
```

The preceding method looks for the services registered in the Eureka server and gets all the instances. In the real world, rather than polling, the instances should publish metrics to a common place, where metrics aggregation will happen.

3. The following `DecisionEngine` code accepts the metric and applies certain scaling policies to determine whether the service requires scaling up or not:

```
  public boolean execute(String serviceId, Map metrics){
  if(scalingPolicies.getPolicy(serviceId).execute(serviceId, metrics)){
      return
deploymentEngine.scaleUp(deploymentRules.getDeploymentRules(serviceId),
serviceId);
    }
    return false;
  }
```

4. Based on the service ID, the policies that are related to the services will be picked up and applied. In this case, a minimal TPM scaling policy is implemented in `TpmScalingPolicy`, as follows:

```
public class TpmScalingPolicy implements ScalingPolicy {
  public boolean execute(String serviceId, Map metrics){
    if(metrics.containsKey("gauge.servo.tpm")){
      Double tpm = (Double) metrics.get("gauge.servo.tpm");
      System.out.println("gauge.servo.tpm " + tpm);
      return (tpm > 10);
    }
    return false;
  }
}
```

5. If the policy returns `true`, `DecisionEngine` then invokes `DeploymentEngine` to spin up another instance. `DeploymentEngine` makes use of `DeploymentRules` to decide how to execute scaling. The rules can enforce the number of min and max instances, in which region or machine the new instance has to be started, the resources required for the new instance, and so on. `DummyDeploymentRule` simply makes sure the max instance is not more than 2.
6. `DeploymentEngine`, in this case, uses the **JSch (Java Secure Channel)** library from JCraft to SSH to the destination server and start the service. This requires the following additional Maven dependency:

```
<dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.53</version>
</dependency>
```

7. The current SSH implementation is kept simple enough as we will change this in future chapters. In this example, `DeploymentEngine` sends the following command over the SSH library on the target machine:

```
 String command ="java -jar -Dserver.port=8091
./work/codebox/chapter6/chapter6.search/target/search-1.0.jar";
```

   Integration with Nexus happens from the target machine using Linux scripts with Nexus CLI or using `curl`. In this example, we will not explore Nexus.
8. The next step is to change the Search microservice to expose a new gauge for TPM. We have to change all the microservices developed earlier to submit this additional metric.

   We will only examine Search in this chapter, but in order to complete it, all the services have to be updated. In order to get the `gauge.servo.tpm` metrics, we have to add `TPMCounter` to all the microservices.

   The following code counts the transactions over a sliding window of 1 minute:

```
class TPMCounter {
  LongAdder count;
```

```
  Calendar expiry = null;
  TPMCounter(){
    reset();
  }
  void reset (){
    count = new LongAdder();
    expiry = Calendar.getInstance();
    expiry.add(Calendar.MINUTE, 1);
  }
  boolean isExpired(){
    return Calendar.getInstance().after(expiry);
  }
  void increment(){
     if(isExpired()){
       reset();
     }
     count.increment();
  }
}
```

9. The following code needs to be added to `SearchController` to set the `tpm` value:

```
class SearchRestController {
  TPMCounter tpm = new TPMCounter();
  @Autowired
  GaugeService gaugeService;
   //other code
```

10. The following code is from the get REST endpoint (the search method) of `SearchRestController`, which submits the `tpm` value as a gauge to the actuator endpoint:

```
tpm.increment();
gaugeService.submit("tpm", tpm.count.intValue());
```

Running the life cycle manager

Perform the following steps to run the life cycle manager developed in the previous section:

1. Edit `DeploymentEngine.java` and update the password to reflect the machine's password, as follows. This is required for the SSH connection:

```
session.setPassword("rajeshrv");
```

2. Build all the projects by running Maven from the root folder (`Chapter 6`) via the following command:

```
mvn -Dmaven.test.skip=true clean install
```

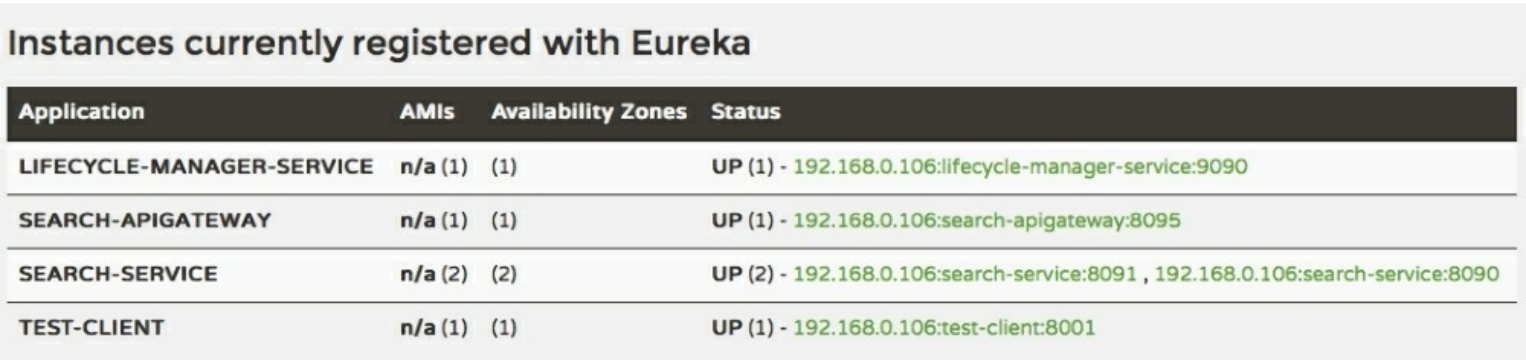
3. Then, run RabbitMQ, as follows:

```
./rabbitmq-server
```

4. Ensure that the Config server is pointing to the right configuration repository. We need to add a property file for the life cycle manager.
5. Run the following commands from the respective project folders:

```
java -jar target/config-server-0.0.1-SNAPSHOT.jar
java -jar target/eureka-server-0.0.1-SNAPSHOT.jar
java -jar target/lifecycle-manager-0.0.1-SNAPSHOT.jar
java -jar target/search-1.0.jar
java -jar target/search-apigateway-1.0.jar
java -jar target/website-1.0.jar
```

6. Once all the services are started, open a browser window and load `http://localhost:8001`.
7. Execute the flight search 11 times, one after the other, within a minute. This will trigger the decision engine to instantiate another instance of the Search microservice.
8. Open the Eureka console (`http://localhost:8761`) and watch for a second **SEARCH-SERVICE**. Once the server is started, the instances will appear as shown here:



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LIFECYCLE-MANAGER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.106:lifecycle-manager-service:9090
SEARCH-APIGATEWAY	n/a (1)	(1)	UP (1) - 192.168.0.106:search-apigateway:8095
SEARCH-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.106:search-service:8091 , 192.168.0.106:search-service:8090
TEST-CLIENT	n/a (1)	(1)	UP (1) - 192.168.0.106:test-client:8001

Summary

In this chapter, you learned the importance of autoscaling when deploying large-scale microservices.

We also explored the concept of autoscaling and the different models of and approaches to autoscaling, such as the time-based, resource-based, queue length-based, and predictive ones. We then reviewed the role of a life cycle manager in the context of microservices and reviewed its capabilities. Finally, we ended this chapter by reviewing a sample implementation of a simple custom life cycle manager in the context of BrownField PSS microservices.

Autoscaling is an important supporting capability required when dealing with large-scale microservices. We will discuss a more mature implementation of the life cycle manager in Chapter 9, *Managing Dockerized Microservices with Mesos and Marathon*.

The next chapter will explore the logging and monitoring capabilities that are indispensable for successful microservice deployments.

Chapter 7. Logging and Monitoring Microservices

One of the biggest challenges due to the very distributed nature of Internet-scale microservices deployment is the logging and monitoring of individual microservices. It is difficult to trace end-to-end transactions by correlating logs emitted by different microservices. As with monolithic applications, there is no single pane of glass to monitor microservices.

This chapter will cover the necessity and importance of logging and monitoring in microservice deployments. This chapter will further examine the challenges and solutions to address logging and monitoring with a number of potential architectures and technologies.

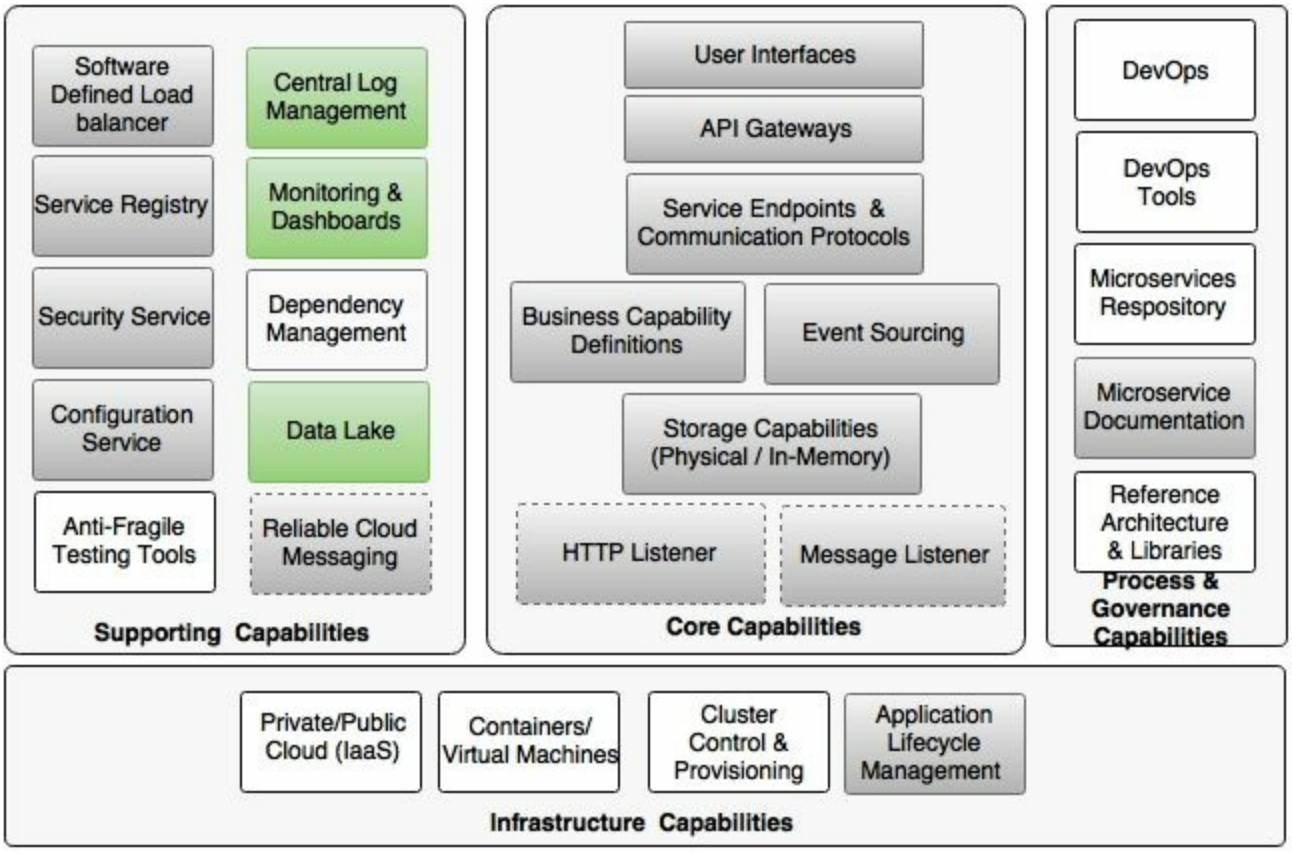
By the end of this chapter, you will learn about:

- The different options, tools, and technologies for log management
- The use of Spring Cloud Sleuth in tracing microservices
- The different tools for end-to-end monitoring of microservices
- The use of Spring Cloud Hystrix and Turbine for circuit monitoring
- The use of data lakes in enabling business data analysis

Reviewing the microservice capability model

In this chapter, we will explore the following microservice capabilities from the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*:

- **Central Log Management**
- **Monitoring and Dashboards**
- **Dependency Management** (part of Monitoring and Dashboards)
- **Data Lake**



Understanding log management challenges

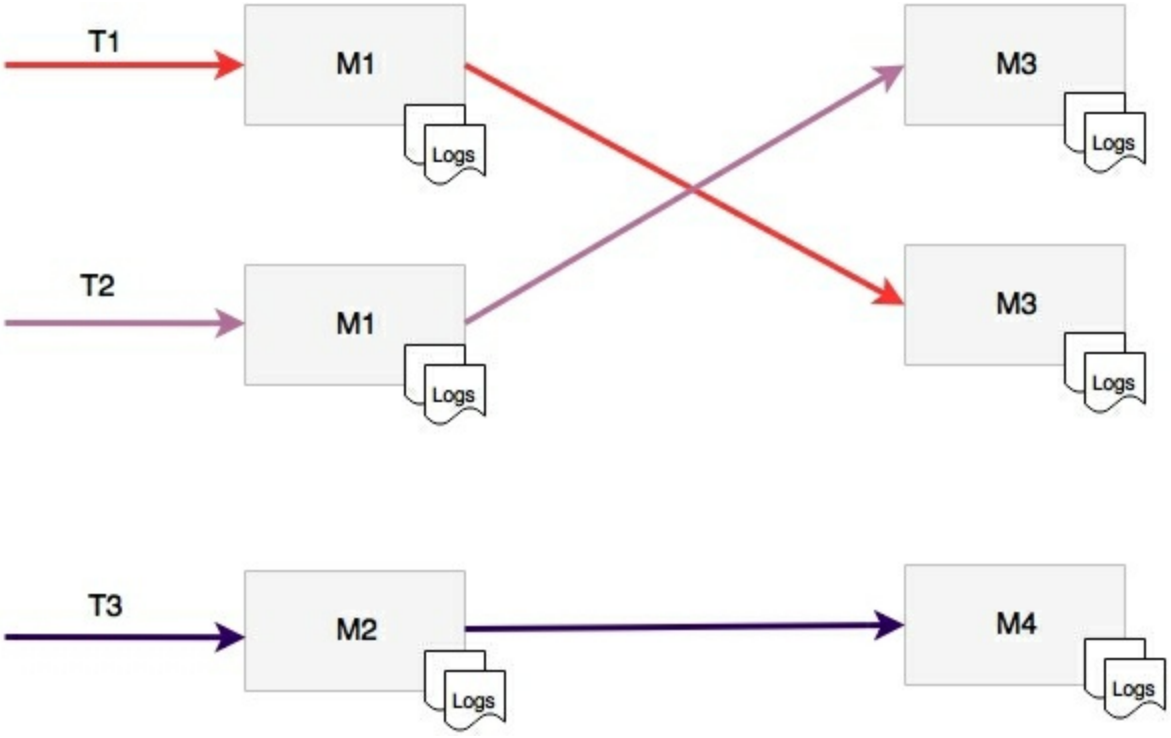
Logs are nothing but streams of events coming from a running process. For traditional JEE applications, a number of frameworks and libraries are available to log. Java Logging (JUL) is an option off the shelf from Java itself. Log4j, Logback, and SLF4J are some of the other popular logging frameworks available. These frameworks support both UDP as well as TCP protocols for logging. Applications send log entries to the console or to the filesystem. File recycling techniques are generally employed to avoid logs filling up all the disk space.

One of the best practices of log handling is to switch off most of the log entries in production due to the high cost of disk IOs. Not only do disk IOs slow down the application, but they can also severely impact scalability. Writing logs into the disk also requires high disk capacity. An out-of-disk-space scenario can bring down the application. Logging frameworks provide options to control logging at runtime to restrict what is to be printed and what not. Most of these frameworks provide fine-grained control over the logging controls. They also provide options to change these configurations at runtime.

On the other hand, logs may contain important information and have high value if properly analyzed. Therefore, restricting log entries essentially limits our ability to understand the application's behavior.

When moved from traditional to cloud deployment, applications are no longer locked to a particular, predefined machine. Virtual machines and containers are not hardwired with an application. The machines used for deployment can change from time to time. Moreover, containers such as Docker are ephemeral. This essentially means that one cannot rely on the persistent state of the disk. Logs written to the disk are lost once the container is stopped and restarted. Therefore, we cannot rely on the local machine's disk to write log files.

As we discussed in Chapter 1, *Demystifying Microservices*, one of the principles of the Twelve-Factor app is to avoid routing or storing log files by the application itself. In the context of microservices, they will run on isolated physical or virtual machines, resulting in fragmented log files. In this case, it is almost impossible to trace end-to-end transactions that span multiple microservices:



As shown in the diagram, each microservice emits logs to a local filesystem. In this case, microservice M1 calls M3. These services write their logs to their own local filesystems. This makes it harder to correlate and understand the end-to-end transaction flow. Also, as shown in the diagram, there are two instances of M1 and two instances of M3 running on two different machines. In this case, log aggregation at the service level is hard to achieve.

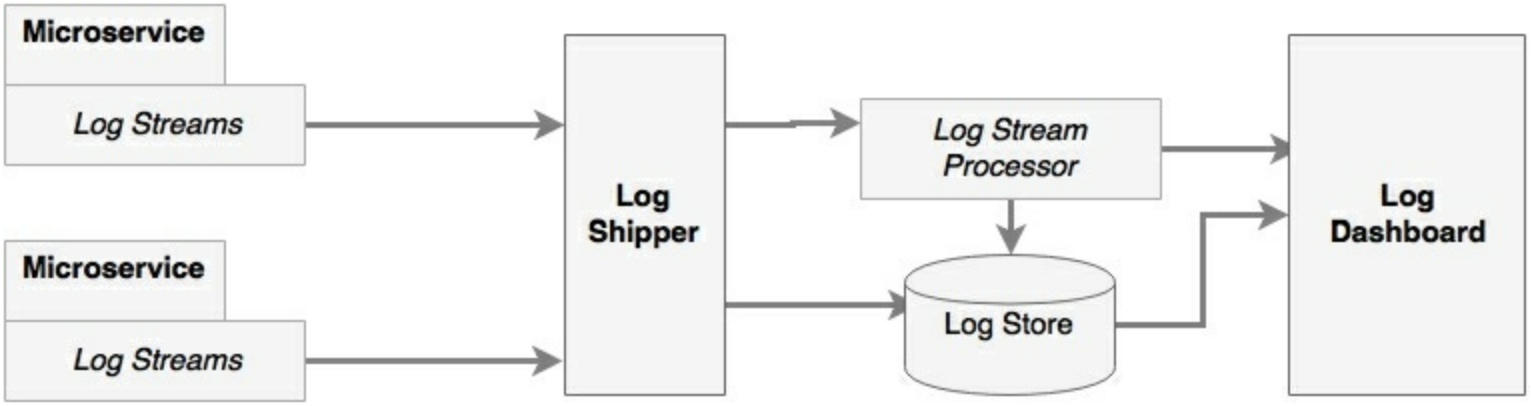
A centralized logging solution

In order to address the challenges stated earlier, traditional logging solutions require serious rethinking. The new logging solution, in addition to addressing the preceding challenges, is also expected to support the capabilities summarized here:

- The ability to collect all log messages and run analytics on top of the log messages
- The ability to correlate and track transactions end to end
- The ability to keep log information for longer time periods for trending and forecasting
- The ability to eliminate dependency on the local disk system
- The ability to aggregate log information coming from multiple sources such as network devices, operating system, microservices, and so on

The solution to these problems is to centrally store and analyze all log messages, irrespective of the source of log. The fundamental principle employed in the new logging solution is to detach log storage and processing from service execution environments. Big data solutions are better suited to storing and processing large numbers of log messages more effectively than storing and processing them in microservice execution environments.

In the centralized logging solution, log messages will be shipped from the execution environment to a central big data store. Log analysis and processing will be handled using big data solutions:



As shown in the preceding logical diagram, there are a number of components in the centralized logging solution, as follows:

- **Log streams**: These are streams of log messages coming out of source systems. The source system can be microservices, other applications, or even network devices. In typical Java-based systems, these are equivalent to streaming Log4j log messages.
- **Log shippers**: Log shippers are responsible for collecting the log messages coming from different sources or endpoints. The log shippers then send these messages to another set of endpoints, such as writing to a database, pushing to a dashboard, or sending it to stream-processing endpoint for further real-time processing.
- **Log store**: A log store is the place where all log messages are stored for real-time analysis,

trending, and so on. Typically, a log store is a NoSQL database, such as HDFS, capable of handling large data volumes.

- **Log stream processor**: The log stream processor is capable of analyzing real-time log events for quick decision making. A stream processor takes actions such as sending information to a dashboard, sending alerts, and so on. In the case of self-healing systems, stream processors can even take actions to correct the problems.
- **Log dashboard**: A dashboard is a single pane of glass used to display log analysis results such as graphs and charts. These dashboards are meant for the operational and management staff.

The benefit of this centralized approach is that there is no local I/O or blocking disk writes. It also does not use the local machine's disk space. This architecture is fundamentally similar to the lambda architecture for big data processing.

Note

To read more on the Lambda architecture, go to http://lambda-architecture.net.

It is important to have in each log message a context, message, and correlation ID. The context typically has the timestamp, IP address, user information, process details (such as service, class, and functions), log type, classification, and so on. The message will be plain and simple free text information. The correlation ID is used to establish the link between service calls so that calls spanning microservices can be traced.

The selection of logging solutions

There are a number of options available to implement a centralized logging solution. These solutions use different approaches, architectures, and technologies. It is important to understand the capabilities required and select the right solution that meets the needs.

Cloud services

There are a number of cloud logging services available, such as the SaaS solution.

Loggly is one of the most popular cloud-based logging services. Spring Boot microservices can use Loggly's Log4j and Logback appenders to directly stream log messages into the Loggly service.

If the application or service is deployed in AWS, AWS CloudTrail can be integrated with Loggly for log analysis.

Papertrial, Logsene, Sumo Logic, Google Cloud Logging, and Logentries are examples of other cloud-based logging solutions.

The cloud logging services take away the overhead of managing complex infrastructures and large storage solutions by providing them as simple-to-integrate services. However, latency is one of the key factors to be considered when selecting cloud logging as a service.

Off-the-shelf solutions

There are many purpose-built tools to provide end-to-end log management capabilities that are installable locally in an on-premises data center or in the cloud.

Graylog is one of the popular open source log management solutions. Graylog uses Elasticsearch for log storage and MongoDB as a metadata store. Graylog also uses GELF libraries for Log4j log streaming.

Splunk is one of the popular commercial tools available for log management and analysis. Splunk uses the log file shipping approach, compared to log streaming used by other solutions to collect logs.

Best-of-breed integration

The last approach is to pick and choose best-of-breed components and build a custom logging solution.

Log shippers

There are log shippers that can be combined with other tools to build an end-to-end log management solution. The capabilities differ between different log shipping tools.

Logstash is a powerful data pipeline tool that can be used to collect and ship log files. Logstash acts as a broker that provides a mechanism to accept streaming data from different sources and sync them to different destinations. Log4j and Logback appenders can also be used to send log messages directly from Spring Boot microservices to Logstash. The other end of Logstash is connected to Elasticsearch, HDFS, or any other database.

Fluentd is another tool that is very similar to Logstash, as is Logspout, but the latter is more appropriate in a Docker container-based environment.

Log stream processors

Stream-processing technologies are optionally used to process log streams on the fly. For example, if a 404 error is continuously occurring as a response to a particular service call, it means there is something wrong with the service. Such situations have to be handled as soon as possible. Stream processors are pretty handy in such cases as they are capable of reacting to certain streams of events that a traditional reactive analysis can't.

A typical architecture used for stream processing is a combination of Flume and Kafka together with either Storm or Spark Streaming. Log4j has Flume appenders, which are useful to collect log messages. These messages are pushed into distributed Kafka message queues. The stream processors collect data from Kafka and process them on the fly before sending it to Elasticsearch and other log stores.

Spring Cloud Stream, Spring Cloud Stream Modules, and Spring Cloud Data Flow can also be used to build the log stream processing.

Log storage

Real-time log messages are typically stored in Elasticsearch. Elasticsearch allows clients to query based on text-based indexes. Apart from Elasticsearch, HDFS is also commonly used to store archived log messages. MongoDB or Cassandra is used to store summary data, such as monthly aggregated transaction counts. Offline log processing can be done using Hadoop's MapReduce programs.

Dashboards

The last piece required in the central logging solution is a dashboard. The most commonly used dashboard for log analysis is Kibana on top of an Elasticsearch data store. Graphite and Grafana are also used to display log analysis reports.

A custom logging implementation

The tools mentioned before can be leveraged to build a custom end-to-end logging solution. The most commonly used architecture for custom log management is a combination of Logstash, Elasticsearch, and Kibana, also known as the ELK stack.

Note

The full source code of this chapter is available under the `Chapter 7` project in the code files. Copy `chapter5.configserver`, `chapter5.eurekaserver`, `chapter5.search`, `chapter5.search-apigateway`, and `chapter5.website` into a new STS workspace and rename them `chapter7.*`.

The following diagram shows the log monitoring flow:



In this section, a simple implementation of a custom logging solution using the ELK stack will be examined.

Follow these steps to implement the ELK stack for logging:

1. Download and install Elasticsearch, Kibana, and Logstash from https://www.elastic.co.
2. Update the Search microservice (`chapter7.search`). Review and ensure that there are some log statements in the Search microservice. The log statements are nothing special but simple log statements using `slf4j`, as follows:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
  //other code goes here
  private static final Logger logger =
LoggerFactory.getLogger(SearchRestController.class);
//other code goes here

logger.info("Looking to load flights...");
for (Flight flight :
flightRepository.findByOriginAndDestinationAndFlightDate("NYC", "SFO", "22-
JAN-16")) {
      logger.info(flight.toString());
}
```

3. Add the `logstash` dependency to integrate `logback` to Logstash in the Search service's

`pom.xml` file, as follows:

```
<dependency>
  <groupId>net.logstash.logback</groupId>
  <artifactId>logstash-logback-encoder</artifactId>
  <version>4.6</version>
</dependency>
```

4. Also, downgrade the `logback` version to be compatible with Spring 1.3.5.RELEASE via the following line:

```
<logback.version>1.1.6</logback.version>
```

5. Override the default Logback configuration. This can be done by adding a new `logback.xml` file under `src/main/resources`, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include
resource="org/springframework/boot/logging/logback/defaults.xml"/>
  <include resource="org/springframework/boot/logging/logback/console-
appender.xml" />
    <appender name="stash"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>localhost:4560</destination>
        <!-- encoder is required -->
        <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
    </appender>
  <root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="stash" />
  </root>
</configuration>
```

The preceding configuration overrides the default Logback configuration by adding a new TCP socket `appender`, which streams all the log messages to a Logstash service, which is listening on port `4560`. It is important to add an encoder, as mentioned in the previous configuration.

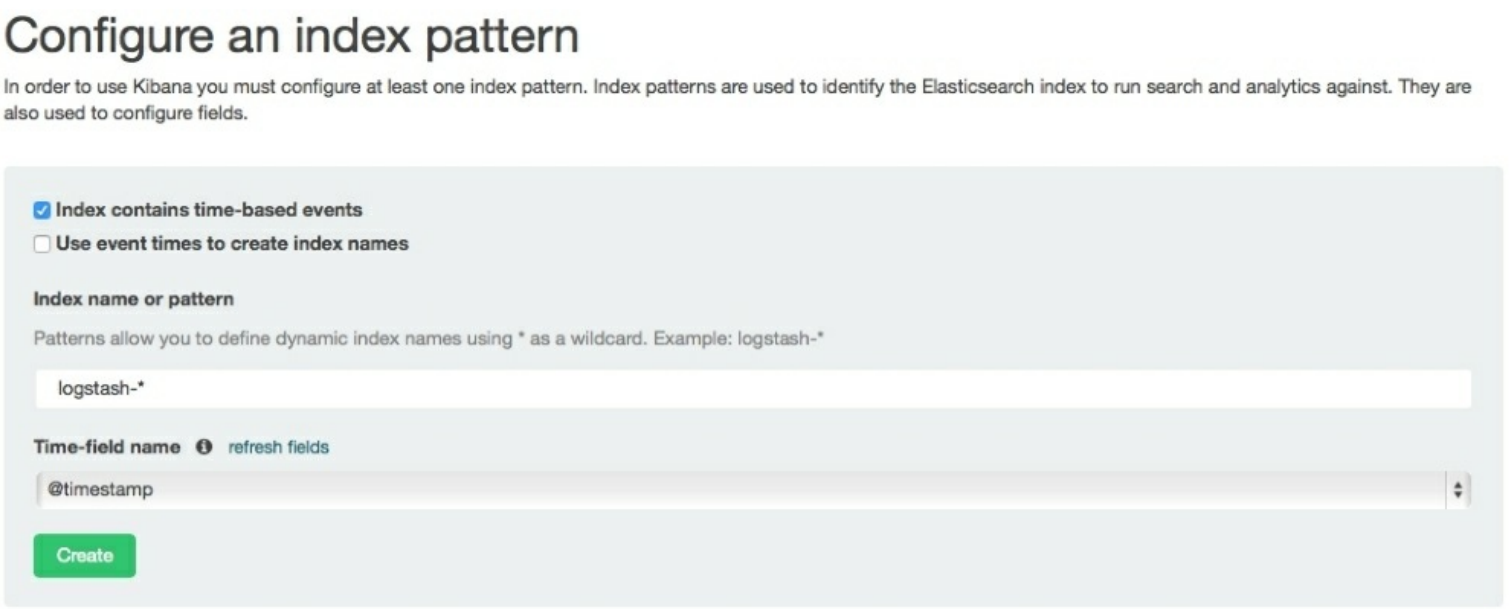
6. Create a configuration as shown in the following code and store it in a `logstash.conf` file. The location of this file is irrelevant as it will be passed as an argument when starting Logstash. This configuration will take input from the socket listening on `4560` and send the output to Elasticsearch running on `9200`. The `stdout` is optional and is set to debug:

```
input {
  tcp {
     port => 4560
     host => localhost
  }
}
output {
elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

7. Run Logstash, Elasticsearch, and Kibana from their respective installation folders, as follows:

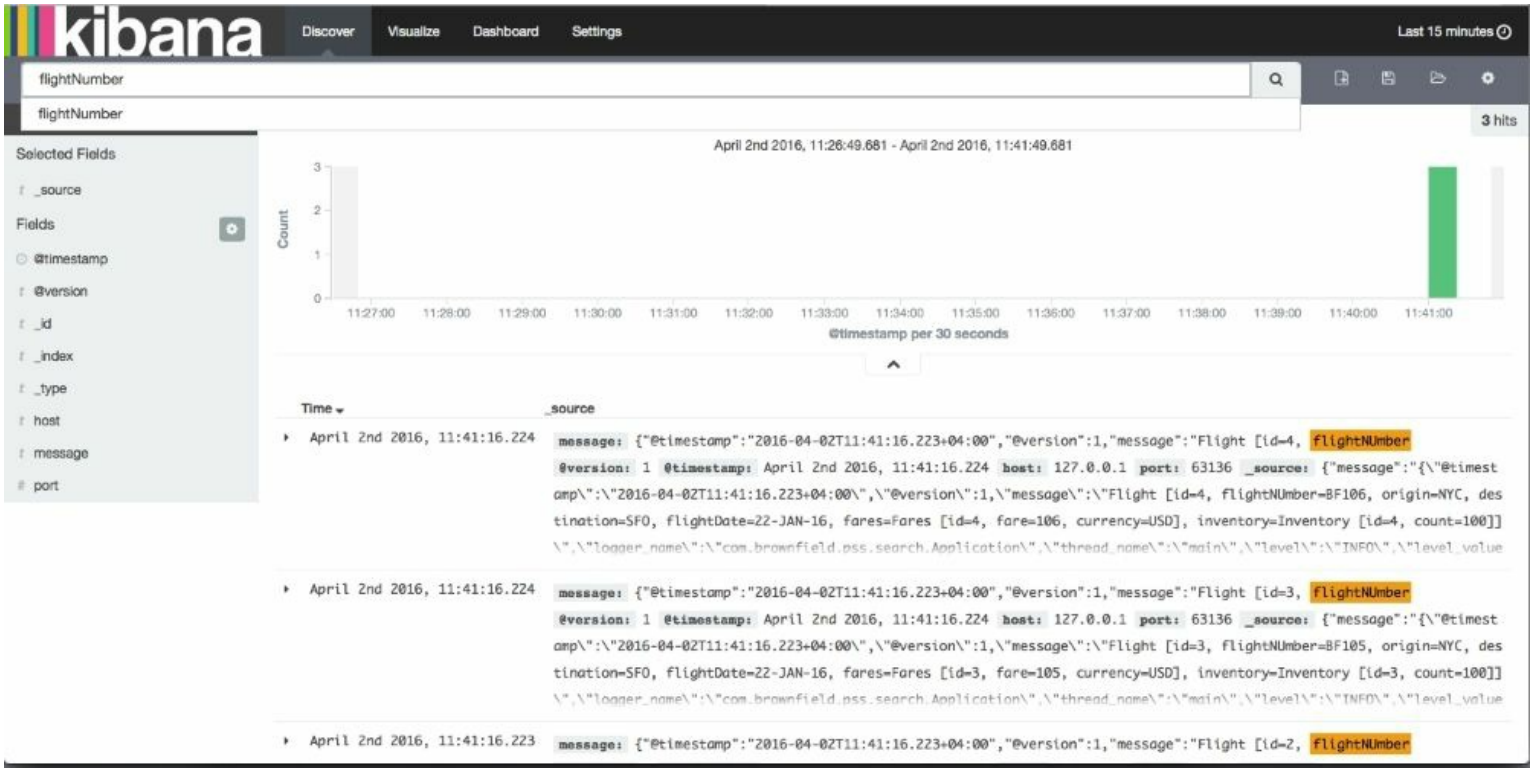
```
./bin/logstash -f logstash.conf
./bin/elasticsearch
./bin/kibana
```

8. Run the Search microservice. This will invoke the unit test cases and result in printing the log statements mentioned before.
9. Go to a browser and access Kibana, at `http://localhost:5601`.
10. Go to **Settings** | **Configure an index pattern**, as shown here:



11. Go to the **Discover** menu to see the logs. If everything is successful, we will see the Kibana screenshot as follows. Note that the log messages are displayed in the Kibana screen.

    Kibana provides out-of-the-box features to build summary charts and graphs using log messages:


kibana
Discover
Visualize
Dashboard
Settings
Last 15 minutes
flightNumber
flightNumber
3 hits
Selected Fields
_source
Fields
@timestamp
@version
_id
_index
_type
host
message
port
April 2nd 2016, 11:26:49.681 - April 2nd 2016, 11:41:49.681
Count
3
2
1
0
11:27:00
11:28:00
11:29:00
11:30:00
11:31:00
11:32:00
11:33:00
11:34:00
11:35:00
11:36:00
11:37:00
11:38:00
11:39:00
11:40:00
11:41:00
@timestamp per 30 seconds
Time
_source
April 2nd 2016, 11:41:16.224
message: {"@timestamp":"2016-04-0ZT11:41:16.223+04:00","@version":1,"message":"Flight [id=4, flightNUmber
@version: 1 @timestamp: April 2nd 2016, 11:41:16.224 host: 127.0.0.1 port: 63136 _source: {"message":"{\"@timest
amp\":\"2016-04-0ZT11:41:16.223+04:00\",\"@version\":1,\"message\":\"Flight [id=4, flightNUmber=BF106, origin=NYC, des
tination=SFO, flightDate=22-JAN-16, fares=Fares [id=4, fare=106, currency=USD], inventory=Inventory [id=4, count=100]]
\",\"logger_name\":\"com.brownfield.pss.search.Application\",\"thread_name\":\"main\",\"level\":\"INFO\",\"level_value
April 2nd 2016, 11:41:16.224
message: {"@timestamp":"2016-04-0ZT11:41:16.223+04:00","@version":1,"message":"Flight [id=3, flightNUmber
@version: 1 @timestamp: April 2nd 2016, 11:41:16.224 host: 127.0.0.1 port: 63136 _source: {"message":"{\"@timest
amp\":\"2016-04-0ZT11:41:16.223+04:00\",\"@version\":1,\"message\":\"Flight [id=3, flightNUmber=BF105, origin=NYC, des
tination=SFO, flightDate=22-JAN-16, fares=Fares [id=3, fare=105, currency=USD], inventory=Inventory [id=3, count=100]]
\",\"logger_name\":\"com.brownfield.pss.search.Application\",\"thread_name\":\"main\",\"level\":\"INFO\",\"level_value
April 2nd 2016, 11:41:16.223
message: {"@timestamp":"2016-04-0ZT11:41:16.223+04:00","@version":1,"message":"Flight [id=2, flightNUmber

Distributed tracing with Spring Cloud Sleuth

The previous section addressed microservices' distributed and fragmented logging issue by centralizing the log data. With the central logging solution, we can have all the logs in a central storage. However, it is still almost impossible to trace end-to-end transactions. In order to do end-to-end tracking, transactions spanning microservices need to have a correlation ID.

Twitter's Zipkin, Cloudera's HTrace, and Google's Dapper systems are examples of distributed tracing systems. Spring Cloud provides a wrapper component on top of these using the Spring Cloud Sleuth library.

Distributed tracing works with the concepts of **span** and **trace**. The span is a unit of work; for example, calling a service is identified by a 64-bit span ID. A set of spans form a tree-like structure is called a trace. Using the trace ID, the call can be tracked end to end:



As shown in the diagram, **Microservice 1** calls **Microservice 2**, and **Microservice 2** calls **Microservice 3**. In this case, as shown in the diagram, the same trace ID is passed across all microservices, which can be used to track transactions end to end.

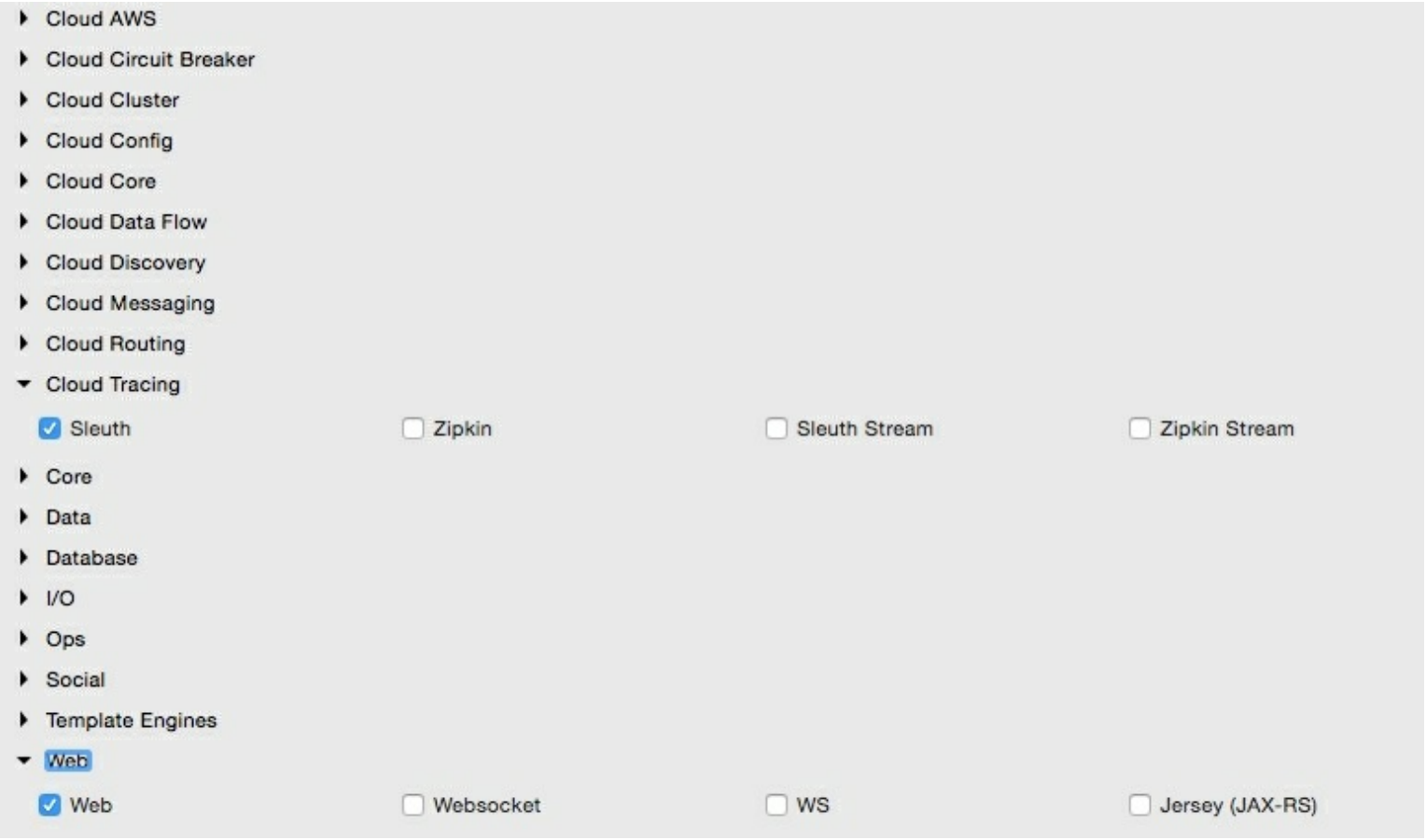
In order to demonstrate this, we will use the Search API Gateway and Search microservices. A new endpoint has to be added in Search API Gateway (`chapter7.search-apigateway`) that internally calls the Search service to return data. Without the trace ID, it is almost impossible to trace or link calls coming from the Website to Search API Gateway to Search microservice. In this case, it only involves two to three services, whereas in a complex environment, there could be many interdependent services.

Follow these steps to create the example using Sleuth:

1. Update Search and Search API Gateway. Before this, the Sleuth dependency needs to be added to the respective POM files, which can be done via the following code:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

2. In the case of building a new service, select **Sleuth** and **Web**, as shown here:



3. Add the Logstash dependency to the Search service as well as the Logback configuration, as in the previous example.
4. The next step is to add two more properties in the Logback configuration, as follows:

```
<property name="spring.application.name" value="search-service"/>
<property name="CONSOLE_LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS}
[${spring.application.name}] [trace=%X{X-Trace-Id:-},span=%X{X-Span-Id:-}]
[%15.15t] %-40.40logger{39}: %m%n"/>
```

   The first property is the name of the application. The names given in this are the service IDs: `search-service` and `search-apigateway` in Search and Search API Gateway, respectively. The second property is an optional pattern used to print the console log messages with a trace ID and span ID. The preceding change needs to be applied to both the services.
5. Add the following piece of code to advise Sleuth when to start a new span ID in the Spring Boot Application class. In this case, `AlwaysSampler` is used to indicate that the span ID has to be created every time a call hits the service. This change needs to be applied in both the services:

```
@Bean
  public AlwaysSampler defaultSampler() {
    return new AlwaysSampler();
  }
```

6. Add a new endpoint to Search API Gateway, which will call the Search service as follows. This is to demonstrate the propagation of the trace ID across multiple microservices. This new method in the gateway returns the operating hub of the airport by calling the Search service, as

follows:

```
  @RequestMapping("/hubongw")
  String getHub(HttpServletRequest req){
    logger.info("Search Request in API gateway for getting Hub, forwarding
to search-service ");
    String hub = restTemplate.getForObject("http://search-
service/search/hub", String.class);
    logger.info("Response for hub received,  Hub "+ hub);
    return hub;
  }
```

7. Add another endpoint in the Search service, as follows:

```
  @RequestMapping("/hub")
  String getHub(){
    logger.info("Searching for Hub, received from search-apigateway ");
    return "SFO";
  }
```

8. Once added, run both the services. Hit the gateway's new hub on the gateway (`/hubongw`) endpoint using a browser (`http://localhost:8095/hubongw`).

   As mentioned earlier, the Search API Gateway service is running on `8095` and the Search service is running on `8090`.
9. Look at the console logs to see the trace ID and span IDs printed. The first print is from Search API Gateway, and the second one came from the Search service. Note that the trace IDs are the same in both the cases, as follows:

```
2016-04-02 17:24:37.624 [search-apigateway] [trace=8a7e278f-7b2b-43e3-a45c-
69d3ca66d663,span=8a7e278f-7b2b-43e3-a45c-69d3ca66d663] [io-8095-exec-10]
c.b.p.s.a.SearchAPIGatewayController    : Response for hub received,  Hub
DXB


2016-04-02 17:24:37.612 [search-service] [trace=8a7e278f-7b2b-43e3-a45c-
69d3ca66d663,span=fd309bba-5b4d-447f-a5e1-7faaab90cfb1] [nio-8090-exec-1]
c.b.p.search.component.SearchComponent  : Searching for Hub, received from
search-apigateway
```

10. Open the Kibana console and search for the trace ID using this trace ID printed in the console. In this case, it is `8a7e278f-7b2b-43e3-a45c-69d3ca66d663`. As shown in the following screenshot, with a trace ID, one can trace service calls that span multiple services:


April 2nd 2016, 17:24:37.613
message: ":"fd309bba-5b4d-447f-a5e1-7faaab90cfb1","X-Trace-Id":"8a7e278f-7b2b-43e3-a45c-69d3ca66d663"}
@version: 1 @timestamp: April 2nd 2016, 17:24:37.613 host: 127.0.0.1 port: 49496 _source: {"message":"{\"@timestamp\":\"2016-04-02T17:24:37.612+04:00\",\"@version\":1,\"message\":\"Searching for Hub, recieved from search-apigateway \",\"logger_name\":\"com.brownfield.pss.search.component.SearchComponent\",\"thread_name\":\"http-nio-8090-exec-1\",
April 2nd 2016, 17:24:37.610
message: ["(begin=1459603477608, end=0, name=http/search/hub, traceId=8a7e278f-7b2b-43e3-a45c-69d3ca66d663, parents","=[8a7e278f-7b2b-43e3-a45c-69d3ca66d663], spanId=fd309bba-5b4d-447f-a5e1-7faaab90cfb1, remote=false","\",\"level_value\":20000,\"HOSTNAME\":\"rvslab.local\",\"X-Span-Id\":\"fd309bba-5b4d-447f-a5e1-7faaab90cfb1\",\"X-Trace-Id\":\"8a7e278f-7b2b-43e3-a45c-69d3ca66d663\"}"] @version: 1 @timestamp: April 2nd 2016, 17:24:37.610 host: 127.0.0.1
April 2nd 2016, 17:24:37.609
message: ["(begin=1459603477608, end=0, name=http/search/hub, traceId=8a7e278f-7b2b-43e3-a45c-69d3ca66d663, parents","=[8a7e278f-7b2b-43e3-a45c-69d3ca66d663], spanId=fd309bba-5b4d-447f-a5e1-7faaab90cfb1, remote=false","\",\"level_value\":20000,\"HOSTNAME\":\"rvslab.local\",\"X-Span-Id\":\"fd309bba-5b4d-447f-a5e1-7faaab90cfb1\",\"X-Trace-Id\":\"8a7e278f-7b2b-43e3-a45c-69d3ca66d663\"}"] @version: 1 @timestamp: April 2nd 2016, 17:24:37.609 host: 127.0.0.1
April 2nd 2016, 17:24:37.609
message: ["=1459603477608, end=0, name=http/hubongw, traceId=8a7e278f-7b2b-43e3-a45c-69d3ca66d663, parents","=[], spanId=8a7e278f-7b2b-43e3-a45c-69d3ca66d663, remote=true, exportable=true, annotations={}, processId","\",\"X-Span-Id\":\"fd309bba-5b4d-447f-a5e1-7faaab90cfb1\",\"X-Trace-Id\":\"8a7e278f-7b2b-43e3-a45c-69d3ca66d663\"}"] @version: 1 @timestamp: April 2nd 2016, 17:24:37.609 host: 127.0.0.1 port: 49496 _source: {"message":"{\"@timestamp\":\"2016-04-02T17:24:37.608+04:00\",\"@version\":1,\"message\":\"With parent: MilliSpan(begin=1459603477608, end=0, name=http
April 2nd 2016, 17:24:37.597
message: ":20000,"HOSTNAME":"rvslab.local","X-Span-Id":"8a7e278f-7b2b-43e3-a45c-69d3ca66d663","X-Trace-Id":"8a7e278f-7b2b-43e3-a45c-69d3ca66d663"} @version: 1 @timestamp: April 2nd 2016, 17:24:37.597 host: 127.0.0.1 port: 49538 _source: {"message":"{\"@timestamp\":\"2016-04-02T17:24:37.596+04:00\",\"@version\":1,\"message\":\"Search Request in API gateway for getting Hub, forwarding to search-service \",\"logger_name\":\"com.brownfield.pss.search.apigateway.SearchAPIGatewayController\",\"thread_name\":\"http-nio-8095-exec-10\",\"level\":\"INFO\",\"level_value\":20000,\

Monitoring microservices

Microservices are truly distributed systems with a fluid deployment topology. Without sophisticated monitoring in place, operations teams may run into trouble managing large-scale microservices. Traditional monolithic application deployments are limited to a number of known services, instances, machines, and so on. This is easier to manage compared to the large number of microservices instances potentially running across different machines. To add more complication, these services dynamically change their topologies. A centralized logging capability only addresses part of the issue. It is important for operations teams to understand the runtime deployment topology and also the behavior of the systems. This demands more than a centralized logging can offer.

In general application, monitoring is more a collection of metrics, aggregation, and their validation against certain baseline values. If there is a service-level breach, then monitoring tools generate alerts and send them to administrators. With hundreds and thousands of interconnected microservices, traditional monitoring does not really offer true value. The one-size-fits-all approach to monitoring or monitoring everything with a single pane of glass is not easy to achieve in large-scale microservices.

One of the main objectives of microservice monitoring is to understand the behavior of the system from a user experience point of view. This will ensure that the end-to-end behavior is consistent and is in line with what is expected by the users.

Monitoring challenges

Similar to the fragmented logging issue, the key challenge in monitoring microservices is that there are many moving parts in a microservice ecosystem.

The typical issues are summarized here:

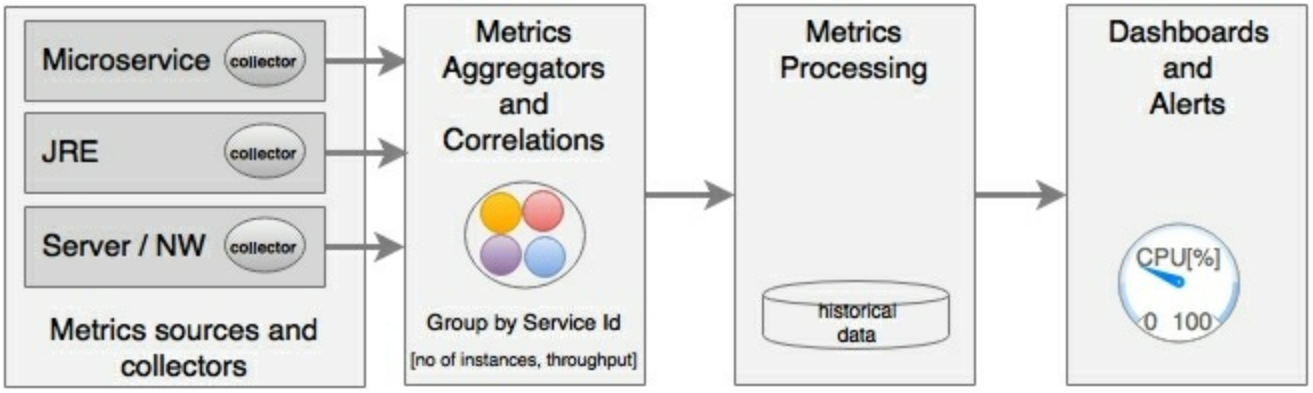
- The statistics and metrics are fragmented across many services, instances, and machines.
- Heterogeneous technologies may be used to implement microservices, which makes things even more complex. A single monitoring tool may not give all the required monitoring options.
- Microservices deployment topologies are dynamic, making it impossible to preconfigure servers, instances, and monitoring parameters.

Many of the traditional monitoring tools are good to monitor monolithic applications but fall short in monitoring large-scale, distributed, interlinked microservice systems. Many of the traditional monitoring systems are agent-based preinstall agents on the target machines or application instances. This poses two challenges:

- If the agents require deep integration with the services or operating systems, then this will be hard to manage in a dynamic environment
- If these tools impose overheads when monitoring or instrumenting the application, it may lead to performance issues

Many traditional tools need baseline metrics. Such systems work with preset rules, such as if the CPU utilization goes above 60% and remains at this level for 2 minutes, then an alert should be sent to the administrator. It is extremely hard to preconfigure these values in large, Internet-scale deployments.

New-generation monitoring applications learn the application's behavior by themselves and set automatic threshold values. This frees up administrators from doing this mundane task. Automated baselines are sometimes more accurate than human forecasts:



As shown in the diagram, the key areas of microservices monitoring are:

- **Metrics sources and data collectors**: Metrics collection at the source is done either by the

server pushing metrics information to a central collector or by embedding lightweight agents to collect information. Data collectors collect monitoring metrics from different sources, such as network, physical machines, containers, software components, applications, and so on. The challenge is to collect this data using autodiscovery mechanisms instead of static configurations.

This is done by either running agents on the source machines, streaming data from the sources, or polling at regular intervals.

- **Aggregation and correlation of metrics**: Aggregation capability is required for aggregating metrics collected from different sources, such as user transaction, service, infrastructure, network, and so on. Aggregation can be challenging as it requires some level of understanding of the application's behavior, such as service dependencies, service grouping, and so on. In many cases, these are automatically formulated based on the metadata provided by the sources.

  Generally, this is done by an intermediary that accept the metrics.
- **Processing metrics and actionable insights**: Once data is aggregated, the next step is to do the measurement. Measurements are typically done using set thresholds. In the new-generation monitoring systems, these thresholds are automatically discovered. Monitoring tools then analyze the data and provide actionable insights.

  These tools may use big data and stream analytics solutions.
- **Alerting, actions, and dashboards**: As soon as issues are detected, they have to be notified to the relevant people or systems. Unlike traditional systems, the microservices monitoring systems should be capable of taking actions on a real-time basis. Proactive monitoring is essential to achieving self-healing. Dashboards are used to display SLAs, KPIs, and so on.

  Dashboards and alerting tools are capable of handling these requirements.

Microservice monitoring is typically done with three approaches. A combination of these is really required for effective monitoring:

- **Application performance monitoring** (**APM**): This is more of a traditional approach to system metrics collection, processing, alerting, and dashboard rendering. These are more from the system's point of view. Application topology discovery and visualization are new capabilities implemented by many of the APM tools. The capabilities vary between different APM providers.
- **Synthetic monitoring**: This is a technique that is used to monitor the system's behavior using end-to-end transactions with a number of test scenarios in a production or production-like environment. Data is collected to validate the system's behavior and potential hotspots. Synthetic monitoring helps understand the system dependencies as well.
- **Real user monitoring** (**RUM**) or **user experience monitoring**: This is typically a browser-based software that records real user statistics, such as response time, availability, and service levels. With microservices, with more frequent release cycle and dynamic topology, user experience monitoring is more important.

Monitoring tools

There are many tools available to monitor microservices. There are also overlaps between many of these tools. The selection of monitoring tools really depends upon the ecosystem that needs to be monitored. In most cases, more than one tool is required to monitor the overall microservice ecosystem.

The objective of this section is to familiarize ourselves with a number of common microservices-friendly monitoring tools:

- AppDynamics, Dynatrace, and New Relic are top commercial vendors in the APM space, as per Gartner Magic Quadrant 2015. These tools are microservice friendly and support microservice monitoring effectively in a single console. Ruxit, Datadog, and Dataloop are other commercial offerings that are purpose-built for distributed systems that are essentially microservices friendly. Multiple monitoring tools can feed data to Datadog using plugins.
- Cloud vendors come with their own monitoring tools, but in many cases, these monitoring tools alone may not be sufficient for large-scale microservice monitoring. For instance, AWS uses CloudWatch and Google Cloud Platform uses Cloud Monitoring to collect information from various sources.
- Some of the data collecting libraries, such as Zabbix, statd, collectd, jmxtrans, and so on operate at a lower level in collecting runtime statistics, metrics, gauges, and counters. Typically, this information is fed into data collectors and processors such as Riemann, Datadog, and Librato, or dashboards such as Graphite.
- Spring Boot Actuator is one of the good vehicles to collect microservices metrics, gauges, and counters, as we discussed in Chapter 2, *Building Microservices with Spring Boot*. Netflix Servo, a metric collector similar to Actuator, and the QBit and Dropwizard metrics also fall in the same category of metric collectors. All these metrics collectors need an aggregator and dashboard to facilitate full-sized monitoring.
- Monitoring through logging is popular but a less effective approach in microservices monitoring. In this approach, as discussed in the previous section, log messages are shipped from various sources, such as microservices, containers, networks, and so on to a central location. Then, we can use the logs files to trace transactions, identify hotspots, and so on. Loggly, ELK, Splunk, and Trace are candidates in this space.
- Sensu is a popular choice for microservice monitoring from the open source community. Weave Scope is another tool, primarily targeting containerized deployments. Spigo is one of the purpose-built microservices monitoring systems closely aligned with the Netflix stack.
- Pingdom, New Relic Synthetics, Runscope, Catchpoint, and so on provide options for synthetic transaction monitoring and user experience monitoring on live systems.
- Circonus is classified more as a DevOps monitoring tool but can also do microservices monitoring. Nagios is a popular open source monitoring tool but falls more into the traditional monitoring system.
- Prometheus provides a time series database and visualization GUI useful in building custom monitoring tools.

Monitoring microservice dependencies

When there are a large number of microservices with dependencies, it is important to have a monitoring tool that can show the dependencies among microservices. It is not a scalable approach to statically configure and manage these dependencies. There are many tools that are useful in monitoring microservice dependencies, as follows:

- Mentoring tools such as AppDynamics, Dynatrace, and New Relic can draw dependencies among microservices. End-to-end transaction monitoring can also trace transaction dependencies. Other monitoring tools, such as Spigo, are also useful for microservices dependency management.
- CMDB tools such as Device42 or purpose-built tools such as Accordance are useful in managing the dependency of microservices. **Veritas Risk Advisor** (**VRA**) is also useful for infrastructure discovery.
- A custom implementation with a Graph database, such as Neo4j, is also useful. In this case, a microservice has to preconfigure its direct and indirect dependencies. At the startup of the service, it publishes and cross-checks its dependencies with a Neo4j database.

Spring Cloud Hystrix for fault-tolerant microservices

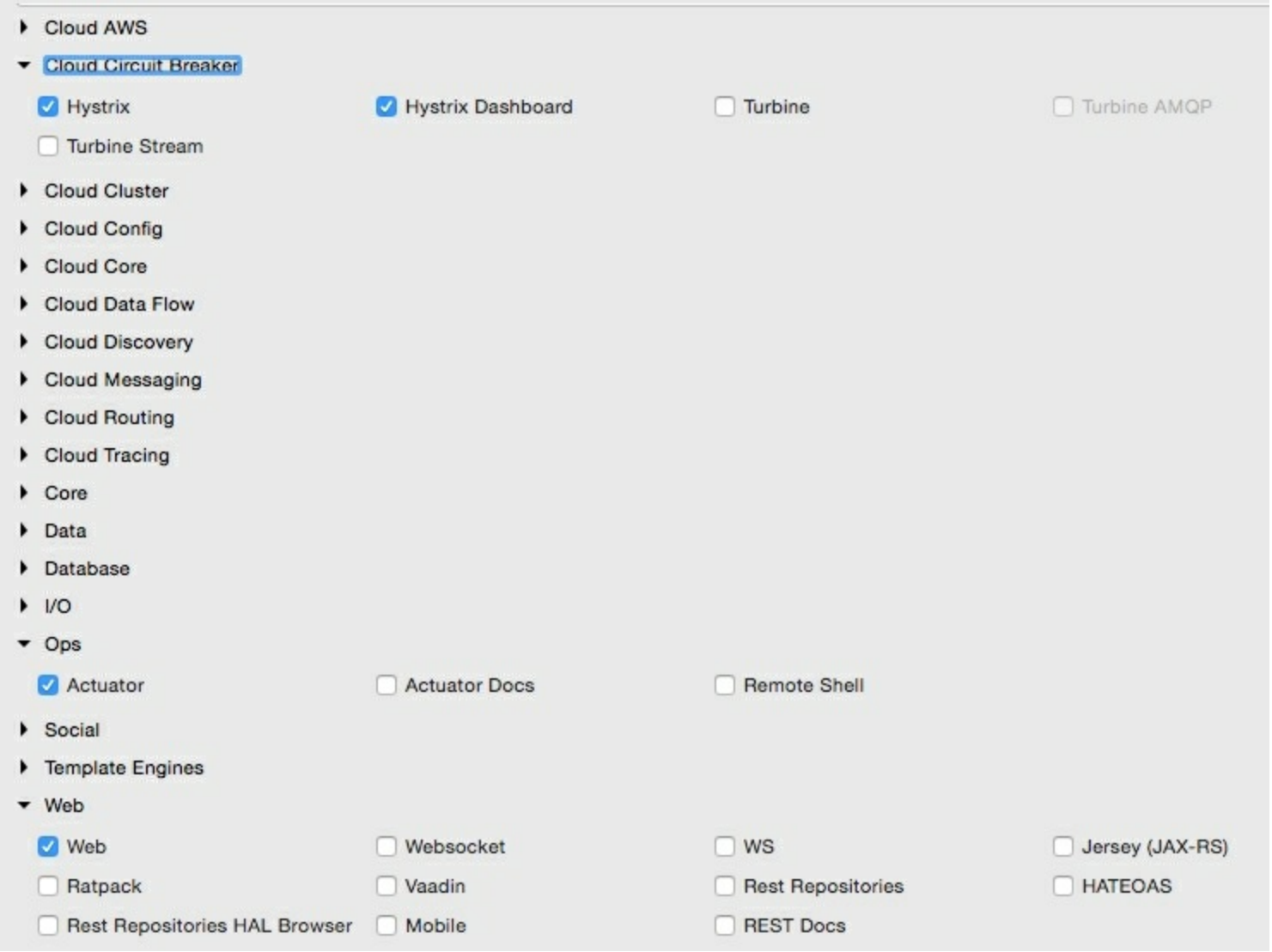
This section will explore Spring Cloud Hystrix as a library for a fault-tolerant and latency-tolerant microservice implementation. Hystrix is based on the *fail fast* and *rapid recovery* principles. If there is an issue with a service, Hystrix helps isolate it. It helps to recover quickly by falling back to another preconfigured fallback service. Hystrix is another battle-tested library from Netflix. Hystrix is based on the circuit breaker pattern.

Note

Read more about the circuit breaker pattern at https://msdn.microsoft.com/en-us/library/dn589784.aspx.

In this section, we will build a circuit breaker with Spring Cloud Hystrix. Perform the following steps to change the Search API Gateway service to integrate it with Hystrix:

1. Update the Search API Gateway service. Add the Hystrix dependency to the service. If developing from scratch, select the following libraries:



2. In the Spring Boot Application class, add `@EnableCircuitBreaker`. This command will tell Spring Cloud Hystrix to enable a circuit breaker for this application. It also exposes the `/hystrix.stream` endpoint for metrics collection.
3. Add a component class to the Search API Gateway service with a method; in this case, this is `getHub` annotated with `@HystrixCommand`. This tells Spring that this method is prone to failure. Spring Cloud libraries wrap these methods to handle fault tolerance and latency tolerance by enabling circuit breaker. The Hystrix command typically follows with a fallback method. In case of failure, Hystrix automatically enables the fallback method mentioned and diverts traffic to the fallback method. As shown in the following code, in this case, `getHub` will fall back to `getDefaultHub`:

```
@Component
class SearchAPIGatewayComponent {
  @LoadBalanced
  @Autowired
  RestTemplate restTemplate;

  @HystrixCommand(fallbackMethod = "getDefaultHub")
  public String getHub(){
    String hub = restTemplate.getForObject("http://search-
service/search/hub", String.class);
    return hub;
  }
  public String getDefaultHub(){
    return "Possibily SFO";
  }
}
```

4. The `getHub` method of `SearchAPIGatewayController` calls the `getHub` method of `SearchAPIGatewayComponent`, as follows:

```
@RequestMapping("/hubongw")
String getHub(){
  logger.info("Search Request in API gateway for getting Hub, forwarding to
search-service ");
  return component.getHub();
}
```

5. The last part of this exercise is to build a Hystrix Dashboard. For this, build another Spring Boot application. Include Hystrix, Hystrix Dashboard, and Actuator when building this application.
6. In the Spring Boot Application class, add the `@EnableHystrixDashboard` annotation.
7. Start the Search service, Search API Gateway, and Hystrix Dashboard applications. Point the browser to the Hystrix Dashboard application's URL. In this example, the Hystrix Dashboard is started on port `9999`. So, open the URL `http://localhost:9999/hystrix`.
8. A screen similar to the following screenshot will be displayed. In the Hystrix Dashboard, enter the URL of the service to be monitored.

   In this case, Search API Gateway is running on port `8095`. Hence, the `hystrix.stream` URL will be `http://localhost:8095/hytrix.stream`, as shown:



Hystrix Dashboard

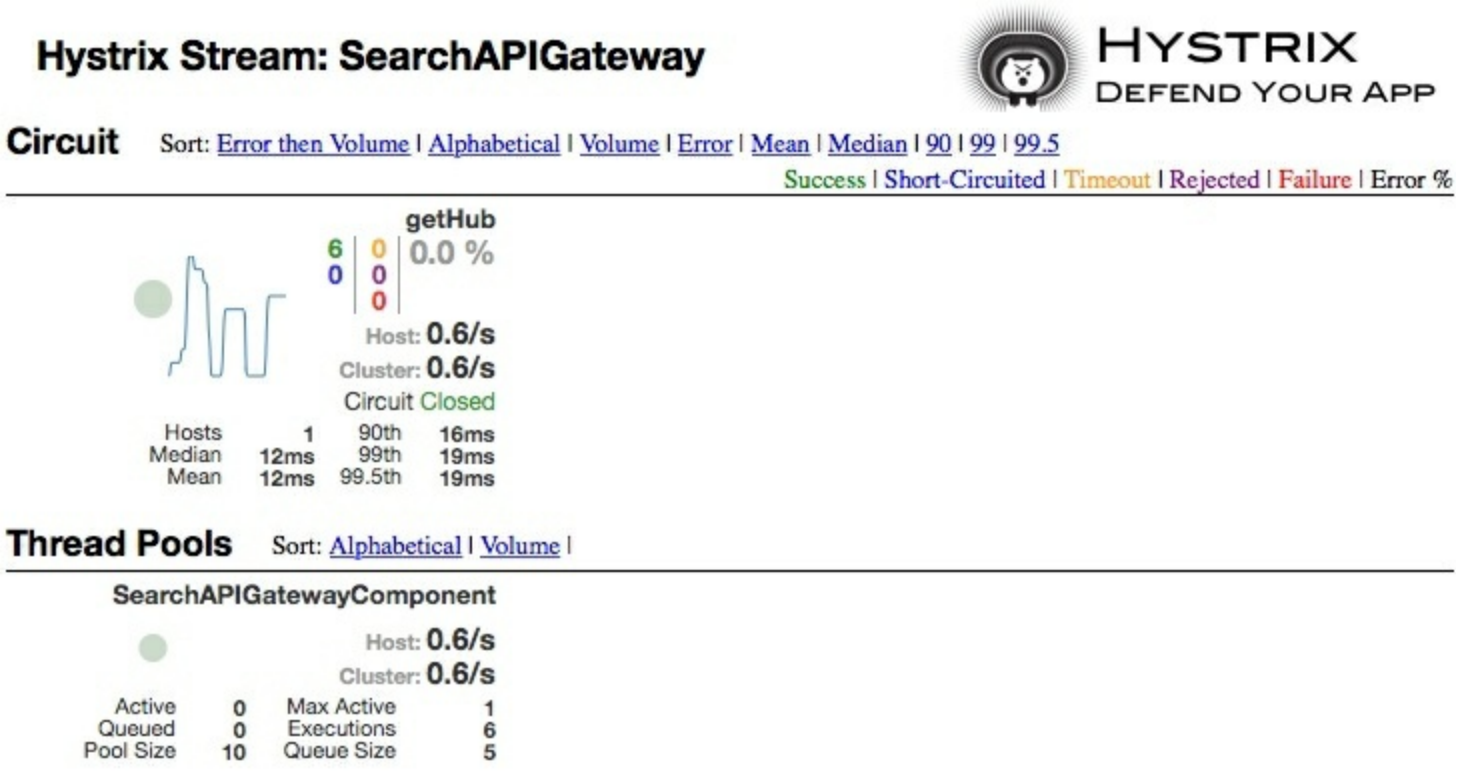
http://localhost:8095/hystrix.stream

Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: http://hystrix-app:port/hystrix.stream

Delay: 2000 ms Title: SearchAPIGateway

Monitor Stream

9. The Hystrix Dashboard will be displayed as follows:



Tip

Note that at least one transaction has to be executed to see the display. This can be done by hitting `http://localhost:8095/hubongw`.

10. Create a failure scenario by shutting down the Search service. Note that the fallback method will be called when hitting the URL `http://localhost:8095/hubongw`.
11. If there are continuous failures, then the circuit status will be changed to open. This can be done by hitting the preceding URL a number of times. In the open state, the original service will no longer be checked. The Hystrix Dashboard will show the status of the circuit as **Open**, as shown in the following screenshot. Once a circuit is opened, periodically, the system will check for the original service status for recovery. When the original service is back, the circuit breaker will fall back to the original service and the status will be set to **Closed**:



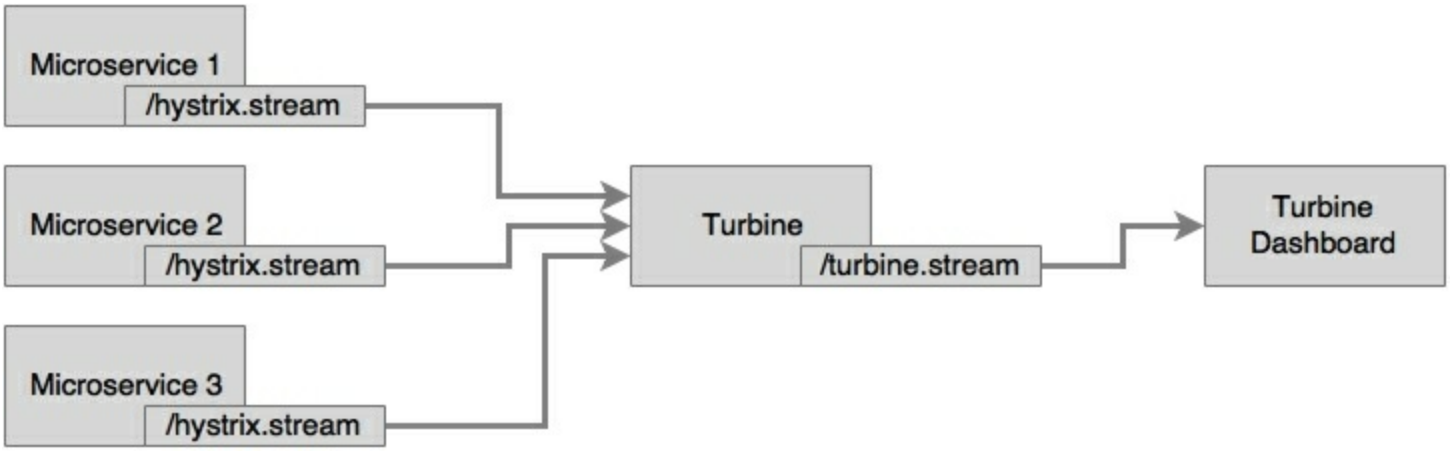
Note

To know the meaning of each of these parameters, visit the Hystrix wiki at https://github.com/Netflix/Hystrix/wiki/Dashboard.

Aggregating Hystrix streams with Turbine

In the previous example, the `/hystrix.stream` endpoint of our microservice was given in the Hystrix Dashboard. The Hystrix Dashboard can only monitor one microservice at a time. If there are many microservices, then the Hystrix Dashboard pointing to the service has to be changed every time we switch the microservices to monitor. Looking into one instance at a time is tedious, especially when there are many instances of a microservice or multiple microservices.

We have to have a mechanism to aggregate data coming from multiple `/hystrix.stream` instances and consolidate it into a single dashboard view. Turbine does exactly the same thing. Turbine is another server that collects Hystrix streams from multiple instances and consolidates them into one `/turbine.stream` instance. Now, the Hystrix Dashboard can point to `/turbine.stream` to get the consolidated information:



Tip

Turbine currently works only with different hostnames. Each instance has to be run on separate hosts. If you are testing multiple services locally on the same host, then update the host file (`/etc/hosts`) to simulate multiple hosts. Once done, `bootstrap.properties` has to be configured as follows:

```
eureka.instance.hostname: localdomain2
```

This example showcases how to use Turbine to monitor circuit breakers across multiple instances and services. We will use the Search service and Search API Gateway in this example. Turbine internally uses Eureka to resolve service IDs that are configured for monitoring.

Perform the following steps to build and execute this example:

1. The Turbine server can be created as just another Spring Boot application using Spring Boot Starter. Select Turbine to include the Turbine libraries.
2. Once the application is created, add `@EnableTurbine` to the main Spring Boot Application class. In this example, both Turbine and Hystrix Dashboard are configured to be run on the same Spring Boot application. This is possible by adding the following annotations to the newly

created Turbine application:

```
@EnableTurbine
@EnableHystrixDashboard
@SpringBootApplication
public class TurbineServerApplication {
```

3. Add the following configuration to the `.yaml` or property file to point to the instances that we are interested in monitoring:

```
spring:
   application:
     name : turbineserver
turbine:
   clusterNameExpression: new String('default')
   appConfig : search-service,search-apigateway
server:
  port: 9090
eureka:
  client:
    serviceUrl:
       defaultZone: http://localhost:8761/eureka/
```

4. The preceding configuration instructs the Turbine server to look up the Eureka server to resolve the `search-service` and `search-apigateway` services. The `search-service` and `search-apigateways` service IDs are used to register services with Eureka. Turbine uses these names to resolve the actual service host and port by checking with the Eureka server. It will then use this information to read `/hystrix.stream` from each of these instances. Turbine will then read all the individual Hystrix streams, aggregate all of them, and expose them under the Turbine server's `/turbine.stream` URL.
5. The cluster name expression is pointing to the default cluster as there is no explicit cluster configuration done in this example. If the clusters are manually configured, then the following configuration has to be used:

```
turbine:
  aggregator:
    clusterConfig: [comma separated clusternames]
```

6. Change the Search service's `SearchComponent` to add another circuit breaker, as follows:

```
  @HystrixCommand(fallbackMethod = "searchFallback")
  public List<Flight> search(SearchQuery query){
```

7. Also, add `@EnableCircuitBreaker` to the main Application class in the Search service.
8. Add the following configuration to `bootstrap.properties` of the Search service. This is required because all the services are running on the same host:

```
Eureka.instance.hostname: localdomain1
```

9. Similarly, add the following in `bootstrap.properties` of the Search API Gateway service. This is to make sure that both the services use different hostnames:

```
eureka.instance.hostname: localdomain2
```

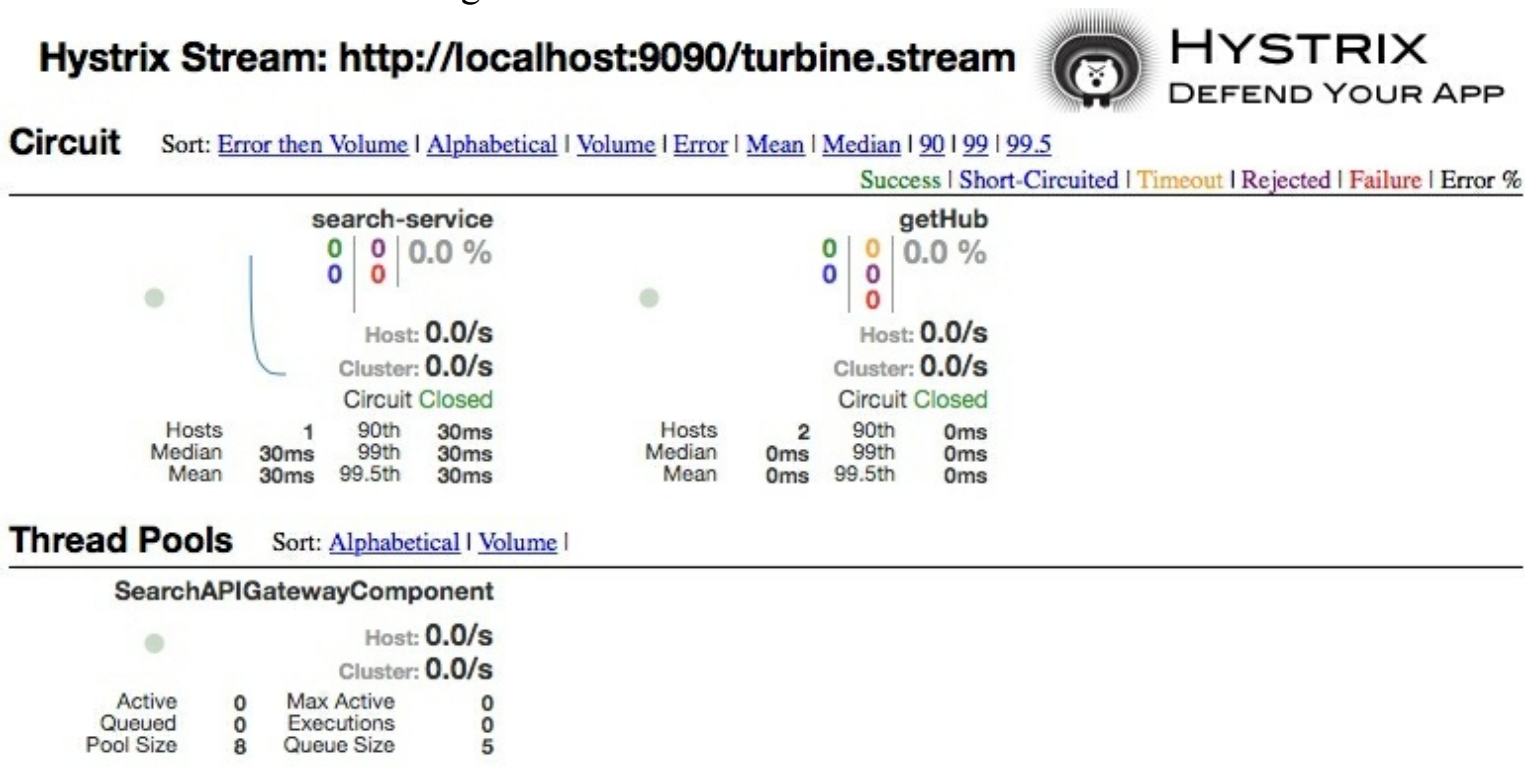
10. In this example, we will run two instances of `search-apigateway`: one on `localdomain1:8095` and another one on `localdomain2:8096`. We will also run one instance of `search-service` on `localdomain1:8090`.
11. Run the microservices with command-line overrides to manage different host addresses, as follows:

```
java -jar -Dserver.port=8096 -Deureka.instance.hostname=localdomain2 -
Dserver.address=localdomain2 target/chapter7.search-apigateway-1.0.jar
java -jar -Dserver.port=8095 -Deureka.instance.hostname=localdomain1 -
Dserver.address=localdomain1 target/chapter7.search-apigateway-1.0.jar
java -jar -Dserver.port=8090 -Deureka.instance.hostname=localdomain1 -
Dserver.address=localdomain1 target/chapter7.search-1.0.jar
```

12. Open Hystrix Dashboard by pointing the browser to `http://localhost:9090/hystrix`.
13. Instead of giving `/hystrix.stream`, this time, we will point to `/turbine.stream`. In this example, the Turbine stream is running on `9090`. Hence, the URL to be given in the Hystrix Dashboard is `http://localhost:9090/turbine.stream`.
14. Fire a few transactions by opening a browser window and hitting the following two URLs: `http://localhost:8095/hubongw` and `http://localhost:8096/hubongw`.

    Once this is done, the dashboard page will show the **getHub** service.
15. Run `chapter7.website`. Execute the search transaction using the website `http://localhost:8001`.

    After executing the preceding search, the dashboard page will show **search-service** as well. This is shown in the following screenshot:



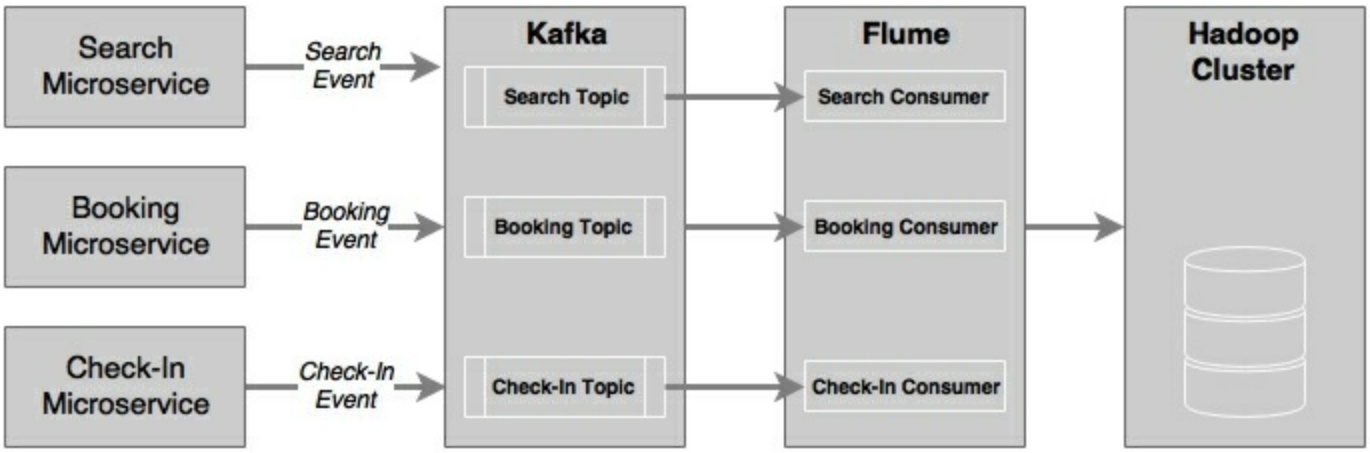
As we can see in the dashboard, **search-service** is coming from the Search microservice, and **getHub** is coming from Search API Gateway. As we have two instances of Search API Gateway, **getHub** is coming from two hosts, indicated by **Hosts 2**.

Data analysis using data lakes

Similarly to the scenario of fragmented logs and monitoring, fragmented data is another challenge in the microservice architecture. Fragmented data poses challenges in data analytics. This data may be used for simple business event monitoring, data auditing, or even deriving business intelligence out of the data.

A data lake or data hub is an ideal solution to handling such scenarios. An event-sourced architecture pattern is generally used to share the state and state changes as events with an external data store. When there is a state change, microservices publish the state change as events. Interested parties may subscribe to these events and process them based on their requirements. A central event store may also subscribe to these events and store them in a big data store for further analysis.

One of the commonly followed architectures for such data handling is shown in the following diagram:



State change events generated from the microservice—in our case, the **Search**, **Booking**, and **Check-In** events—are pushed to a distributed high-performance messaging system, such as Kafka. A data ingestion service, such as Flume, can subscribe to these events and update them to an HDFS cluster. In some cases, these messages will be processed in real time by Spark Streaming. To handle heterogeneous sources of events, Flume can also be used between event sources and Kafka.

Spring Cloud Streams, Spring Cloud Streams modules, and Spring Data Flow are also useful as alternatives for high-velocity data ingestion.

Summary

In this chapter, you learned about the challenges around logging and monitoring when dealing with Internet-scale microservices.

We explored the various solutions for centralized logging. You also learned about how to implement a custom centralized logging using Elasticsearch, Logstash, and Kibana (ELK). In order to understand distributed tracing, we upgraded BrownField microservices using Spring Cloud Sleuth.

In the second half of this chapter, we went deeper into the capabilities required for microservices monitoring solutions and different approaches to monitoring. Subsequently, we examined a number of tools available for microservices monitoring.

The BrownField microservices are further enhanced with Spring Cloud Hystrix and Turbine to monitor latencies and failures in inter-service communications. The examples also demonstrated how to use the circuit breaker pattern to fall back to another service in case of failures.

Finally, we also touched upon the importance of data lakes and how to integrate a data lake architecture in a microservice context.

Microservice management is another important challenge we need to tackle when dealing with large-scale microservice deployments. The next chapter will explore how containers can help in simplifying microservice management.

Chapter 8. Containerizing Microservices with Docker

In the context of microservices, containerized deployment is the icing on the cake. It helps microservices be more autonomous by self-containing the underlying infrastructure, thereby making the microservices cloud neutral.

This chapter will introduce the concepts and relevance of virtual machine images and the containerized deployment of microservices. Then, this chapter will further familiarize readers with building Docker images for the BrownField PSS microservices developed with Spring Boot and Spring Cloud. Finally, this chapter will also touch base on how to manage, maintain, and deploy Docker images in a production-like environment.

By the end of this chapter, you will learn about:

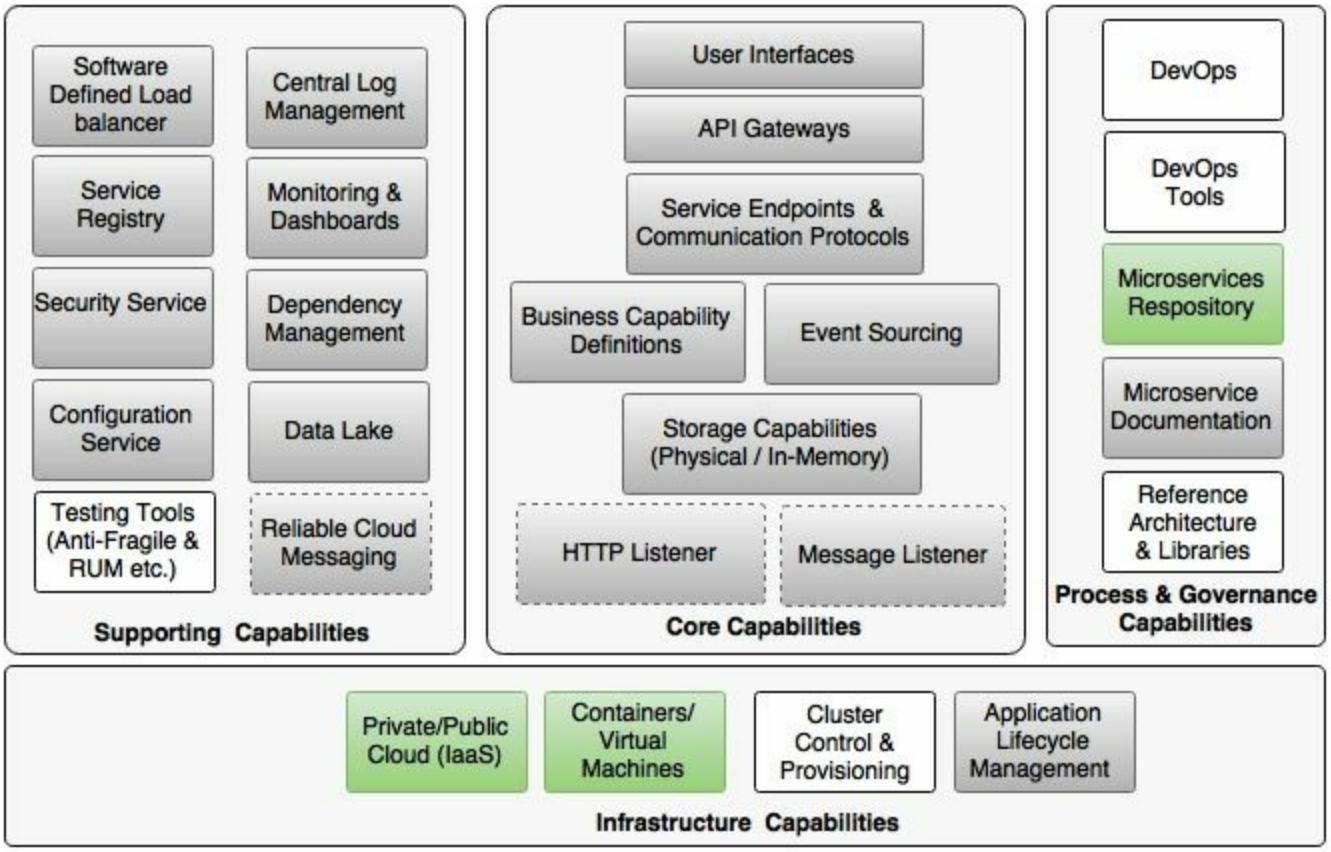
- The concept of containerization and its relevance in the context of microservices
- Building and deploying microservices as Docker images and containers
- Using AWS as an example of cloud-based Docker deployments

Reviewing the microservice capability model

In this chapter, we will explore the following microservice capabilities from the microservice capability model discussed in Chapter 3, *Applying Microservices Concepts*:

- Containers and virtual machines
- The private/public cloud
- The microservices repository

The model is shown in the following diagram:



Understanding the gaps in BrownField PSS microservices

In Chapter 5, *Scaling Microservices with Spring Cloud,* BrownField PSS microservices were developed using Spring Boot and Spring Cloud. These microservices are deployed as versioned fat JAR files on bare metals, specifically on a local development machine.

In Chapter 6, *Autoscaling Microservices*, the autoscaling capability was added with the help of a custom life cycle manager. In Chapter 7, *Logging and Monitoring Microservices*, challenges around logging and monitoring were addressed using centralized logging and monitoring solutions.

There are still a few gaps in our BrownField PSS implementation. So far, the implementation has not used any cloud infrastructure. Dedicated machines, as in traditional monolithic application deployments, are not the best solution for deploying microservices. Automation such as automatic provisioning, the ability to scale on demand, self-service, and payment based on usage are essential capabilities required to manage large-scale microservice deployments efficiently. In general, a cloud infrastructure provides all these essential capabilities. Therefore, a private or public cloud with the capabilities mentioned earlier is better suited to deploying Internet-scale microservices.

Also, running one microservice instance per bare metal is not cost effective. Therefore, in most cases, enterprises end up deploying multiple microservices on a single bare metal server. Running multiple microservices on a single bare metal could lead to a "noisy neighbor" problem. There is no isolation between the microservice instances running on the same machine. As a result, services deployed on a single machine may eat up others' space, thus impacting their performance.

An alternate approach is to run the microservices on VMs. However, VMs are heavyweight in nature. Therefore, running many smaller VMs on a physical machine is not resource efficient. This generally results in resource wastage. In the case of sharing a VM to deploy multiple services, we would end up facing the same issues of sharing the bare metal, as explained earlier.

In the case of Java-based microservices, sharing a VM or bare metal to deploy multiple microservices also results in sharing JRE among microservices. This is because the fat JARs created in our BrownField PSS abstract only application code and its dependencies but not JREs. Any update on JRE installed on the machine will have an impact on all the microservices deployed on this machine. Similarly, if there are OS-level parameters, libraries, or tunings that are required for specific microservices, then it will be hard to manage them on a shared environment.

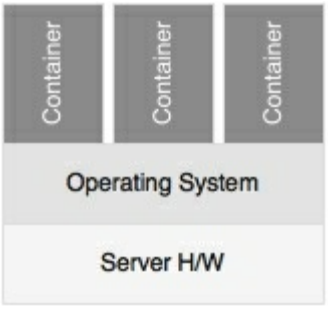
One microservice principle insists that it should be self-contained and autonomous by fully encapsulating its end-to-end runtime environment. In order to align with this principle, all components, such as the OS, JRE, and microservice binaries, have to be self-contained and isolated. The only option to achieve this is to follow the approach of deploying one microservice per VM. However, this will result in underutilized virtual machines, and in many cases, extra cost due to this

can nullify benefits of microservices.

What are containers?

Containers are not revolutionary, ground-breaking concepts. They have been in action for quite a while. However, the world is witnessing the re-entry of containers, mainly due to the wide adoption of cloud computing. The shortcomings of traditional virtual machines in the cloud computing space also accelerated the use of containers. Container providers such as **Docker** simplified container technologies to a great extent, which also enabled a large adoption of container technologies in today's world. The recent popularity of DevOps and microservices also acted as a catalyst for the rebirth of container technologies.

So, what are containers? Containers provide private spaces on top of the operating system. This technique is also called operating system virtualization. In this approach, the kernel of the operating system provides isolated virtual spaces. Each of these virtual spaces is called a container or **virtual engine** (**VE**). Containers allow processes to run on an isolated environment on top of the host operating system. A representation of multiple containers running on the same host is shown as follows:



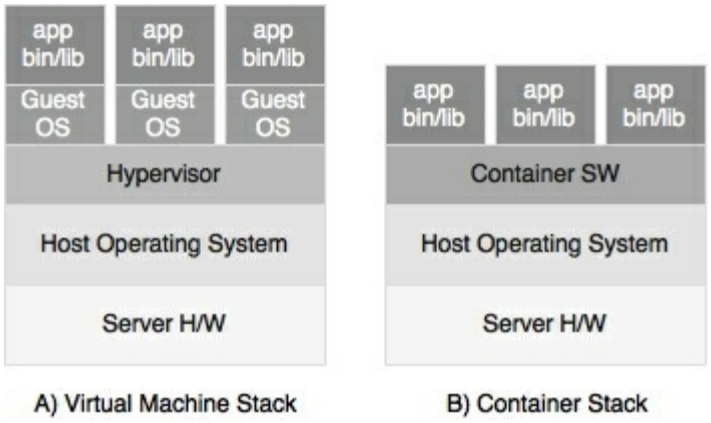
Containers are easy mechanisms to build, ship, and run compartmentalized software components. Generally, containers package all the binaries and libraries that are essential for running an application. Containers reserve their own filesystem, IP address, network interfaces, internal processes, namespaces, OS libraries, application binaries, dependencies, and other application configurations.

There are billions of containers used by organizations. Moreover, there are many large organizations heavily investing in container technologies. Docker is far ahead of the competition, supported by many large operating system vendors and cloud providers. **Lmctfy**, **SystemdNspawn**, **Rocket**, **Drawbridge**, **LXD**, **Kurma**, and **Calico** are some of the other containerization solutions. Open container specification is also under development.

The difference between VMs and containers

VMs such as **Hyper-V**, **VMWare**, and **Zen** were popular choices for data center virtualization a few years ago. Enterprises experienced a cost saving by implementing virtualization over the traditional bare metal usage. It has also helped many enterprises utilize their existing infrastructure in a much more optimized manner. As VMs support automation, many enterprises experienced that they had to make lesser management effort with virtual machines. Virtual machines also helped organizations get isolated environments for applications to run in.

Prima facie, both virtualization and containerization exhibit exactly the same characteristics. However, in a nutshell, containers and virtual machines are not the same. Therefore, it is unfair to make an apple-to-apple comparison between VMs and containers. Virtual machines and containers are two different techniques and address different problems of virtualization. This difference is evident from the following diagram:



Virtual machines operate at a much lower level compared to containers. VMs provide hardware virtualization, such as that of CPUs, motherboards, memory, and so on. A VM is an isolated unit with an embedded operating system, generally called a **Guest OS**. VMs replicate the whole operating system and run it within the VM with no dependency on the host operating system environment. As VMs embed the full operating system environment, these are heavyweight in nature. This is an advantage as well as a disadvantage. The advantage is that VMs offer complete isolation to the processes running on VMs. The disadvantage is that it limits the number of VMs one can spin up in a bare metal due to the resource requirements of VMs.

The size of a VM has a direct impact on the time to start and stop it. As starting a VM in turn boots the OS, the start time for VMs is generally high. VMs are more friendly with infrastructure teams as it requires a low level of infrastructure competency to manage VMs.

In the container world, containers do not emulate the entire hardware or operating system. Unlike VMs, containers share certain parts of the host kernel and operating system. There is no concept of guest OS in the case of containers. Containers provide an isolated execution environment directly on

top of the host operating system. This is its advantage as well as disadvantage. The advantage is that it is lighter as well as faster. As containers on the same machine share the host operating system, the overall resource utilization of containers is fairly small. As a result, many smaller containers can be run on the same machine, as compared to heavyweight VMs. As containers on the same host share the host operating system, there are limitations as well. For example, it is not possible to set iptables firewall rules inside a container. Processes inside the container are completely independent from the processes on different containers running on the same host.

Unlike VMs, container images are publically available on community portals. This makes developers' lives much easier as they don't have to build the images from scratch; instead, they can now take a base image from certified sources and add additional layers of software components on top of the downloaded base image.

The lightweight nature of the containers is also opening up a plethora of opportunities, such as automated build, publishing, downloading, copying, and so on. The ability to download, build, ship, and run containers with a few commands or to use REST APIs makes containers more developer friendly. Building a new container does not take more than a few seconds. Containers are now part and parcel of continuous delivery pipelines as well.

In summary, containers have many advantages over VMs, but VMs have their own exclusive strengths. Many organizations use both containers and VMs, such as by running containers on top of VMs.

The benefits of containers

We have already considered the many benefits of containers over VMs. This section will explain the overall benefits of containers beyond the benefits of VMs:

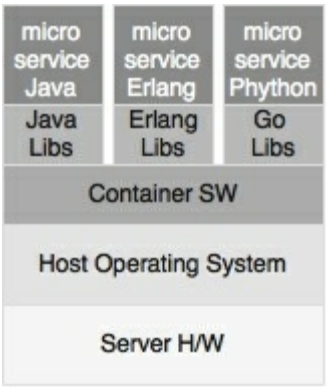
- **Self-contained**: Containers package the essential application binaries and their dependencies together to make sure that there is no disparity between different environments such as development, testing, or production. This promotes the concept of Twelve-Factor applications and that of immutable containers. Spring Boot microservices bundle all the required application dependencies. Containers stretch this boundary further by embedding JRE and other operating system-level libraries, configurations, and so on, if there are any.
- **Lightweight**: Containers, in general, are smaller in size with a lighter footprint. The smallest container, Alpine, has a size of less than 5 MB. The simplest Spring Boot microservice packaged with an Alpine container with Java 8 would only come to around 170 MB in size. Though the size is still on the higher side, it is much less than the VM image size, which is generally in GBs. The smaller footprint of containers not only helps spin new containers quickly but also makes building, shipping, and storing easier.
- **Scalable**: As container images are smaller in size and there is no OS booting at startup, containers are generally faster to spin up and shut down. This makes containers the popular choice for cloud-friendly elastic applications.
- **Portable**: Containers provide portability across machines and cloud providers. Once the containers are built with all the dependencies, they can be ported across multiple machines or across multiple cloud providers without relying on the underlying machines. Containers are portable from desktops to different cloud environments.
- **Lower license cost**: Many software license terms are based on the physical core. As containers share the operating system and are not virtualized at the physical resources level, there is an advantage in terms of the license cost.
- **DevOps**: The lightweight footprint of containers makes it easy to automate builds and publish and download containers from remote repositories. This makes it easy to use in Agile and DevOps environments by integrating with automated delivery pipelines. Containers also support the concept of *build once* by creating immutable containers at build time and moving them across multiple environments. As containers are not deep into the infrastructure, multidisciplinary DevOps teams can manage containers as part of their day-to-day life.
- **Version controlled**: Containers support versions by default. This helps build versioned artifacts, just as with versioned archive files.
- **Reusable**: Container images are reusable artifacts. If an image is built by assembling a number of libraries for a purpose, it can be reused in similar situations.
- **Immutable containers**: In this concept, containers are created and disposed of after usage. They are never updated or patched. Immutable containers are used in many environments to avoid complexities in patching deployment units. Patching results in a lack of traceability and an inability to recreate environments consistently.

Microservices and containers

There is no direct relationship between microservices and containers. Microservices can run without containers, and containers can run monolithic applications. However, there is a sweet spot between microservices and containers.

Containers are good for monolithic applications, but the complexities and the size of the monolith application may kill some of the benefits of the containers. For example, spinning new containers quickly may not be easy with monolithic applications. In addition to this, monolithic applications generally have local environment dependencies, such as the local disk, stovepipe dependencies with other systems, and so on. Such applications are difficult to manage with container technologies. This is where microservices go hand in hand with containers.

The following diagram shows three polyglot microservices running on the same host machine and sharing the same operating system but abstracting the runtime environment:



The real advantage of containers can be seen when managing many polyglot microservices—for instance, one microservice in Java and another one in Erlang or some other language. Containers help developers package microservices written in any language or technology in a platform- and technology-agnostic fashion and uniformly distribute them across multiple environments. Containers eliminate the need to have different deployment management tools to handle polyglot microservices. Containers not only abstract the execution environment but also how to access the services. Irrespective of the technologies used, containerized microservices expose REST APIs. Once the container is up and running, it binds to certain ports and exposes its APIs. As containers are self-contained and provide full stack isolation among services, in a single VM or bare metal, one can run multiple heterogeneous microservices and handle them in a uniform way.

Introduction to Docker

The previous sections talked about containers and their benefits. Containers have been in the business for years, but the popularity of Docker has given containers a new outlook. As a result, many container definitions and perspectives emerged from the Docker architecture. Docker is so popular that even containerization is referred to as **dockerization**.

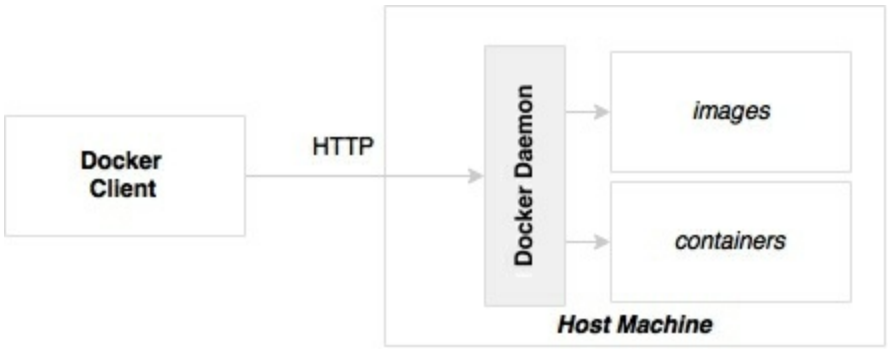
Docker is a platform to build, ship, and run lightweight containers based on Linux kernels. Docker has default support for Linux platforms. It also has support for Mac and Windows using **Boot2Docker**, which runs on top of Virtual Box.

Amazon **EC2 Container Service** (**ECS**) has out-of-the-box support for Docker on AWS EC2 instances. Docker can be installed on bare metals and also on traditional virtual machines such as VMWare or Hyper-V.

The key components of Docker

A Docker installation has two key components: a **Docker daemon** and a **Docker client**. Both the Docker daemon and Docker client are distributed as a single binary.

The following diagram shows the key components of a Docker installation:



The Docker daemon

The Docker daemon is a server-side component that runs on the host machine responsible for building, running, and distributing Docker containers. The Docker daemon exposes APIs for the Docker client to interact with the daemon. These APIs are primarily REST-based endpoints. One can imagine that the Docker daemon as a controller service running on the host machine. Developers can programmatically use these APIs to build custom clients as well.

The Docker client

The Docker client is a remote command-line program that interacts with the Docker daemon through either a socket or REST APIs. The CLI can run on the same host as the daemon is running on or it can run on a completely different host and connect to the daemon remotely. Docker users use the CLI to build, ship, and run Docker containers.

Docker concepts

The Docker architecture is built around a few concepts: images, containers, the registry, and the Dockerfile.

Docker images

One of the key concepts of Docker is the image. A Docker image is the read-only copy of the operating system libraries, the application, and its libraries. Once an image is created, it is guaranteed to run on any Docker platform without alterations.

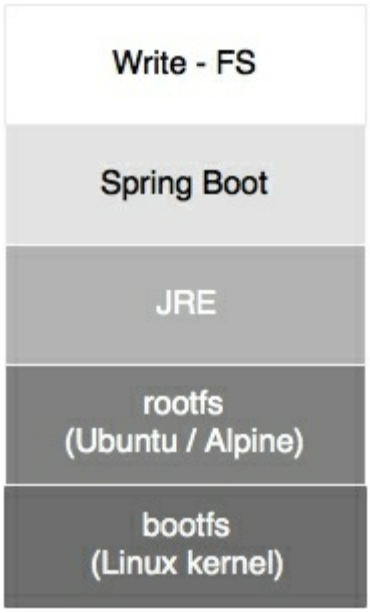
In Spring Boot microservices, a Docker image packages operating systems such as Ubuntu, Alpine, JRE, and the Spring Boot fat application JAR file. It also includes instructions to run the application and expose the services:



As shown in the diagram, Docker images are based on a layered architecture in which the base image is one of the flavors of Linux. Each layer, as shown in the preceding diagram, gets added to the base image layer with the previous image as the parent layer. Docker uses the concept of a union filesystem to combine all these layers into a single image, forming a single filesystem.

In typical cases, developers do not build Docker images from scratch. Images of an operating system, or other common libraries, such as Java 8 images, are publicly available from trusted sources. Developers can start building on top of these base images. The base image in Spring microservices can be JRE 8 rather than starting from a Linux distribution image such as Ubuntu.

Every time we rebuild the application, only the changed layer gets rebuilt, and the remaining layers are kept intact. All the intermediate layers are cached, and hence, if there is no change, Docker uses the previously cached layer and builds it on top. Multiple containers running on the same machine with the same type of base images would reuse the base image, thus reducing the size of the deployment. For instance, in a host, if there are multiple containers running with Ubuntu as the base image, they all reuse the same base image. This is applicable when publishing or downloading images as well:



As shown in the diagram, the first layer in the image is a boot filesystem called `bootfs`, which is similar to the Linux kernel and the boot loader. The boot filesystem acts as a virtual filesystem for all images.

On top of the boot filesystem, the operating system filesystem is placed, which is called `rootfs`. The root filesystem adds the typical operating system directory structure to the container. Unlike in the Linux systems, `rootfs`, in the case of Docker, is on a read-only mode.

On top of `rootfs`, other required images are placed as per the requirements. In our case, these are JRE and the Spring Boot microservice JARs. When a container is initiated, a writable filesystem is placed on top of all the other filesystems for the processes to run. Any changes made by the process to the underlying filesystem are not reflected in the actual container. Instead, these are written to the writable filesystem. This writable filesystem is volatile. Hence, the data is lost once the container is stopped. Due to this reason, Docker containers are ephemeral in nature.

The base operating system packaged inside Docker is generally a minimal copy of just the OS filesystem. In reality the process running on top may not use the entire OS services. In a Spring Boot microservice, in many cases, the container just initiates a CMD and JVM and then invokes the Spring Boot fat JAR.

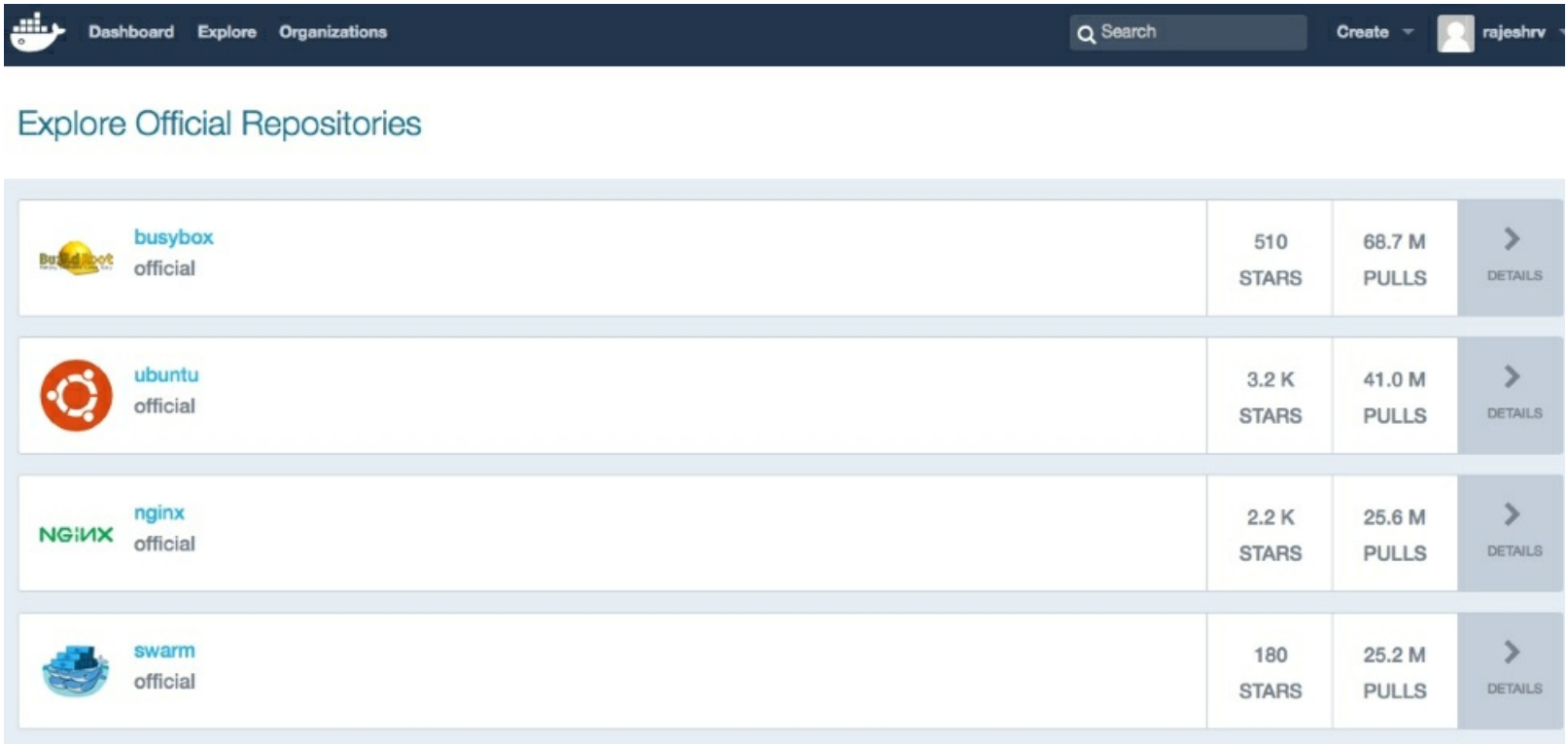
Docker containers

Docker containers are the running instances of a Docker image. Containers use the kernel of the host operating system when running. Hence, they share the host kernel with other containers running on the same host. The Docker runtime ensures that the container processes are allocated with their own isolated process space using kernel features such as **cgroups** and the kernel **namespace** of the operating system. In addition to the resource fencing, containers get their own filesystem and network configurations as well.

The containers, when instantiated, can have specific resource allocations, such as the memory and CPU. Containers, when initiated from the same image, can have different resource allocations. The Docker container, by default, gets an isolated **subnet** and **gateway** to the network. The network has three modes.

The Docker registry

The Docker registry is a central place where Docker images are published and downloaded from. The URL https://hub.docker.com is the central registry provided by Docker. The Docker registry has public images that one can download and use as the base registry. Docker also has private images that are specific to the accounts created in the Docker registry. The Docker registry screenshot is shown as follows:



Docker also offers **Docker Trusted Registry**, which can be used to set up registries locally on premises.

Dockerfile

A Dockerfile is a build or scripting file that contains instructions to build a Docker image. There can be multiple steps documented in the Dockerfile, starting from getting a base image. A Dockerfile is a text file that is generally named Dockerfile. The `docker build` command looks up Dockerfile for instructions to build. One can compare a Dockerfile to a `pom.xml` file used in a Maven build.

Deploying microservices in Docker

This section will operationalize our learning by showcasing how to build containers for our BrownField PSS microservices.

Note

The full source code of this chapter is available under the `Chapter 8` project in the code files. Copy `chapter7.configserver`, `chapter7.eurekaserver`, `chapter7.search`, `chapter7.search-apigateway`, and `chapter7.website` into a new STS workspace and rename them `chapter8.*`.

Perform the following steps to build Docker containers for BrownField PSS microservices:

1. Install Docker from the official Docker site at https://www.docker.com.

   Follow the **Get Started** link for the download and installation instructions based on the operating system of choice. Once installed, use the following command to verify the installation:

   ```
   $docker –version
   Docker version 1.10.1, build 9e83765
   ```

2. In this section, we will take a look at how to dockerize the **Search** (`chapter8.search`) microservice, the **Search API Gateway** (`chapter8.search-apigateway`) microservice, and the **Website** (`chapter8.website`) Spring Boot application.
3. Before we make any changes, we need to edit `bootstrap.properties` to change the config server URL from localhost to the IP address as localhost is not resolvable from within the Docker containers. In the real world, this will point to a DNS or load balancer, as follows:

   ```
   spring.cloud.config.uri=http://192.168.0.105:8888
   ```

   ### Note

   Replace the IP address with the IP address of your machine.

4. Similarly, edit `search-service.properties` on the Git repository and change localhost to the IP address. This is applicable for the Eureka URL as well as the RabbitMQ URL. Commit back to Git after updating. You can do this via the following code:

   ```
   spring.application.name=search-service
   spring.rabbitmq.host=192.168.0.105
   spring.rabbitmq.port=5672
   spring.rabbitmq.username=guest
   spring.rabbitmq.password=guest
   orginairports.shutdown:JFK
   eureka.client.serviceUrl.defaultZone: http://192.168.0.105:8761/eureka/
   spring.cloud.stream.bindings.inventoryQ=inventoryQ
   ```

5. Change the RabbitMQ configuration file `rabbitmq.config` by uncommenting the following line to provide access to guest. By default, guest is restricted to be accessed from localhost only:

```
{loopback_users, []}
```

The location of `rabbitmq.config` will be different for different operating systems.

6. Create a Dockerfile under the root directory of the Search microservice, as follows:

```
FROM frolvlad/alpine-oraclejdk8
VOLUME /tmp
ADD  target/search-1.0.jar search.jar
EXPOSE 8090
ENTRYPOINT ["java","-jar","/search.jar"]
```

The following is a quick examination of the contents of the Dockerfile:

- `FROM frolvlad/alpine-oraclejdk8`: This tells the Docker build to use a specific `alpine-oraclejdk8` version as the basic image for this build. The `frolvlad` indicates the repository to locate the `alpine-oraclejdk8` image. In this case, it is an image built with Alpine Linux and Oracle JDK 8. This will help layer our application on top of the base image without setting up Java libraries ourselves. In this case, as this image is not available on our local image store, the Docker build will go ahead and download this image from the remote Docker Hub registry.
- `VOLUME /tmp`: This enables access from the container to the directory specified in the host machine. In our case, this points to the `tmp` directory in which the Spring Boot application creates working directories for Tomcat. The `tmp` directory is a logical one for the container, which indirectly points to one of the local directories of the host.
- `ADD target/search-1.0.jar search.jar`: This adds the application binary file to the container with the destination filename specified. In this case, the Docker build copies `target/search-1.0.jar` to the container as `search.jar`.
- `EXPOSE 8090`: This is to tell the container how to do port mapping. This associates `8090` with external port binding for the internal Spring Boot service.
- `ENTRYPOINT ["java","-jar", "/search.jar"]`: This tells the container which default application to run when a container is started. In this case, we are pointing to the Java process and the Spring Boot fat JAR file to initiate the service.

7. The next step is to run `docker build` from the folder in which the Dockerfile is stored. This will download the base image and run the entries in the Dockerfile one after the other, as follows:

```
docker build -t search:1.0 .
```

The output of this command will be as follows:

```
rvslab:chapter8.search rajeshrv$ docker build -t search:1.0 .
Sending build context to Docker daemon 48.34 MB
Step 1 : FROM frolvlad/alpine-oraclejdk8
 ---> 5b8d90632c89
Step 2 : VOLUME /tmp
 ---> Using cache
 ---> c79a1b3275d4
Step 3 : ADD target/search-1.0.jar app.jar
 ---> 7766e630f139
Removing intermediate container f2ac976e781d
Step 4 : EXPOSE 8090
 ---> Running in 730300fa66a9
 ---> e058cc1615da
Removing intermediate container 730300fa66a9
Step 5 : ENTRYPOINT java -jar /app.jar
 ---> Running in b79116f3e54b
 ---> 5a8d0d6e0bf7
Removing intermediate container b79116f3e54b
Successfully built 5a8d0d6e0bf7
rvslab:chapter8.search rajeshrv$
```

8. Repeat the same steps for Search API Gateway and Website.
9. Once the images are created, they can be verified by typing the following command. This command will list out the images and their details, including the size of image files:

```
docker images
```

The output will be as follows:

```
rvslab:chapter8 rajeshrv$ docker images
REPOSITORY                          TAG                 IMAGE ID
website                             1.0                 263605f253f3
search-apigateway                   1.0                 322890ae3ec1
search                              1.0                 f094b72d1b41
```

10. The next thing to do is run the Docker container. This can be done with the `docker run` command. This command will load and run the container. On starting, the container calls the Spring Boot executable JAR to start the microservice.

    Before starting the containers, ensure that the Config and the Eureka servers are running:

```
docker run --net host -p 8090:8090 -t search:1.0
docker run --net host -p 8095:8095 -t search-apigateway:1.0
docker run --net host -p 8001:8001 -t website:1.0
```

    The preceding command starts the Search and Search API Gateway microservices and Website.

    In this example, we are using the host network (`--net host`) instead of the bridge network to avoid Eureka registering with the Docker container name. This can be corrected by overriding `EurekaInstanceConfigBean`. The host option is less isolated compared to the bridge option from the network perspective. The advantage and disadvantage of host versus bridge depends on

the project.

11. Once all the services are fully started, verify with the `docker ps` command, as shown in the following screenshot:

```
rvslab:chapter8 rajeshrv$ docker ps
CONTAINER ID        IMAGE                       COMMAND
32af53b56945        website:1.0                 "java -jar /website.j"
ce35355aea65        search-apigateway:1.0       "java -jar /search-ap"
5577c5107fc6        search:1.0                  "java -jar /search.ja"
4a423d0872b5        registry:2                  "/bin/registry /etc/d"
rvslab:chapter8 rajeshrv$
```

12. The next step is to point the browser to `http://192.168.99.100:8001`. This will open the BrownField PSS website.

    Note the IP address. This is the IP address of the Docker machine if you are running with Boot2Docker on Mac or Windows. In Mac or Windows, if the IP address is not known, then type the following command to find out the Docker machine's IP address for the default machine:

    ```
    docker-machine ip default
    ```

    If Docker is running on Linux, then this is the host IP address.

Apply the same changes to **Booking**, **Fares**, **Check-in**, and their respective gateway microservices.

Running RabbitMQ on Docker

As our example also uses RabbitMQ, let's explore how to set up RabbitMQ as a Docker container. The following command pulls the RabbitMQ image from Docker Hub and starts RabbitMQ:

```
docker run –net host rabbitmq3
```

Ensure that the URL in `*-service.properties` is changed to the Docker host's IP address. Apply the earlier rule to find out the IP address in the case of Mac or Windows.

Using the Docker registry

The Docker Hub provides a central location to store all the Docker images. The images can be stored as public as well as private. In many cases, organizations deploy their own private registries on premises due to security-related concerns.

Perform the following steps to set up and run a local registry:

1. The following command will start a registry, which will bind the registry on port `5000`:

   ```
   docker run -d -p 5000:5000 --restart=always --name registry registry:2
   ```

2. Tag `search:1.0` to the registry, as follows:

   ```
   docker tag search:1.0 localhost:5000/search:1.0
   ```

3. Then, push the image to the registry via the following command:

   ```
   docker push localhost:5000/search:1.0
   ```

4. Pull the image back from the registry, as follows:

   ```
   docker pull localhost:5000/search:1.0
   ```

Setting up the Docker Hub

In the previous chapter, we played with a local Docker registry. This section will show how to set up and use the Docker Hub to publish the Docker containers. This is a convenient mechanism to globally access Docker images. Later in this chapter, Docker images will be published to the Docker Hub from the local machine and downloaded from the EC2 instances.

In order to do this, create a public Docker Hub account and a repository. For Mac, follow the steps as per the following URL: https://docs.docker.com/mac/step_five/.

In this example, the Docker Hub account is created using the `brownfield` username.

The registry, in this case, acts as the microservices repository in which all the dockerized microservices will be stored and accessed. This is one of the capabilities explained in the microservices capability model.

Publishing microservices to the Docker Hub

In order to push dockerized services to the Docker Hub, follow these steps. The first command tags the Docker image, and the second one pushes the Docker image to the Docker Hub repository:

```
docker tag search:1.0brownfield/search:1.0
docker push brownfield/search:1.0
```

To verify whether the container images are published, go to the Docker Hub repository at `https://hub.docker.com/u/brownfield`.

Repeat this step for all the other BrownField microservices as well. At the end of this step, all the services will be published to the Docker Hub.

Microservices on the cloud

One of the capabilities mentioned in the microservices capability model is the use of the cloud infrastructure for microservices. Earlier in this chapter, we also explored the necessity of using the cloud for microservices deployments. So far, we have not deployed anything to the cloud. As we have eight microservices in total—`Config-server`, `Eureka-server`, Turbine, RabbitMQ, Elasticsearch, Kibana, and Logstash—in our overall BrownField PSS microservices ecosystem, it is hard to run all of them on the local machine.

In the rest of this book, we will operate using AWS as the cloud platform to deploy BrownField PSS microservices.

Installing Docker on AWS EC2

In this section, we will install Docker on the EC2 instance.

This example assumes that readers are familiar with AWS and an account is already created on AWS.

Perform the following steps to set up Docker on EC2:

1. Launch a new EC2 instance. In this case, if we have to run all the instances together, we may need a large instance. The example uses **t2.large**.

   In this example, the following Ubuntu AMI image is used: `ubuntu-trusty-14.04-amd64-server-20160114.5 (ami-fce3c696)`.
2. Connect to the EC2 instance and run the following commands:

   ```
   sudo apt-get update
   sudo apt-get install docker.io
   ```
3. The preceding command will install Docker on an EC2 instance. Verify the installation with the following command:

   ```
   docker version
   ```

Running BrownField services on EC2

In this section, we will set up BrownField microservices on the EC2 instances created. In this case, the build is set up in the local desktop machine, and the binaries will be deployed to AWS.

Perform the following steps to set up services on an EC2 instance:

1. Install Git via the following command:

```
sudo apt-get install git
```

2. Create a Git repository on any folder of your choice.
3. Change the Config server's `bootstrap.properties` to point to the appropriate Git repository created for this example.
4. Change the `bootstrap.properties` of all the microservices to point to the config-server using the private IP address of the EC2 instance.
5. Copy all `*.properties` from the local Git repository to the EC2 Git repository and perform a commit.
6. Change the Eureka server URLs and RabbitMQ URLs in the `*.properties` file to match the EC2 private IP address. Commit the changes to Git once they have been completed.
7. On the local machine, recompile all the projects and create Docker images for the `search`, `search-apigateway`, and `website` microservices. Push all of them to the Docker Hub registry.
8. Copy the config-server and the Eureka-server binaries from the local machine to the EC2 instance.
9. Set up Java 8 on the EC2 instance.
10. Then, execute the following commands in sequence:

```
java –jar config-server.jar
java –jar eureka-server.jar
docker run –net host rabbitmq:3
docker run --net host -p 8090:8090 rajeshrv/search:1.0
docker run --net host -p 8095:8095 rajeshrv/search-apigateway:1.0
docker run --net host -p 8001:8001 rajeshrv/website:1.0
```

11. Check whether all the services are working by opening the URL of the website and executing a search. Note that we will use the public IP address in this case: `http://54.165.128.23:8001`.

Updating the life cycle manager

In Chapter 6, *Autoscaling Microservices*, we considered a life cycle manager to automatically start and stop instances. We used SSH and executed a Unix script to start the Spring Boot microservices on the target machine. With Docker, we no longer need SSH connections as the Docker daemon provides REST-based APIs to start and stop instances. This greatly simplifies the complexities of the deployment engine component of the life cycle manager.

In this section, we will not rewrite the life cycle manager. By and large, we will replace the life cycle manager in the next chapter.

The future of containerization – unikernels and hardened security

Containerization is still evolving, but the number of organizations adopting containerization techniques has gone up in recent times. While many organizations are aggressively adopting Docker and other container technologies, the downside of these techniques is still in the size of the containers and security concerns.

Currently, Docker images are generally heavy. In an elastic automated environment, where containers are created and destroyed quite frequently, size is still an issue. A larger size indicates more code, and more code means that it is more prone to security vulnerabilities.

The future is definitely in small footprint containers. Docker is working on unikernels, lightweight kernels that can run Docker even on low-powered IoT devices. Unikernels are not full-fledged operating systems, but they provide the basic necessary libraries to support the deployed applications.

The security issues of containers are much discussed and debated. The key security issues are around the user namespace segregation or user ID isolation. If the container is on root, then it can by default gain the root privilege of the host. Using container images from untrusted sources is another security concern. Docker is bridging these gaps as quickly as possible, but there are many organizations that use a combination of VMs and Docker to circumvent some of the security concerns.

Summary

In this chapter, you learned about the need to have a cloud environment when dealing with Internet-scale microservices.

We explored the concept of containers and compared them with traditional virtual machines. You also learned the basics of Docker, and we explained the concepts of Docker images, containers, and registries. The importance and benefits of containers were explained in the context of microservices.

This chapter then switched to a hands-on example by dockerizing the BrownField microservice. We demonstrated how to deploy the Spring Boot microservice developed earlier on Docker. You learned the concept of registries by exploring a local registry as well as the Docker Hub to push and pull dockerized microservices.

As the last step, we explored how to deploy a dockerized BrownField microservice in the AWS cloud environment.

Chapter 9. Managing Dockerized Microservices with Mesos and Marathon

In an Internet-scale microservices deployment, it is not easy to manage thousands of dockerized microservices. It is essential to have an infrastructure abstraction layer and a strong cluster control platform to successfully manage Internet-scale microservice deployments.

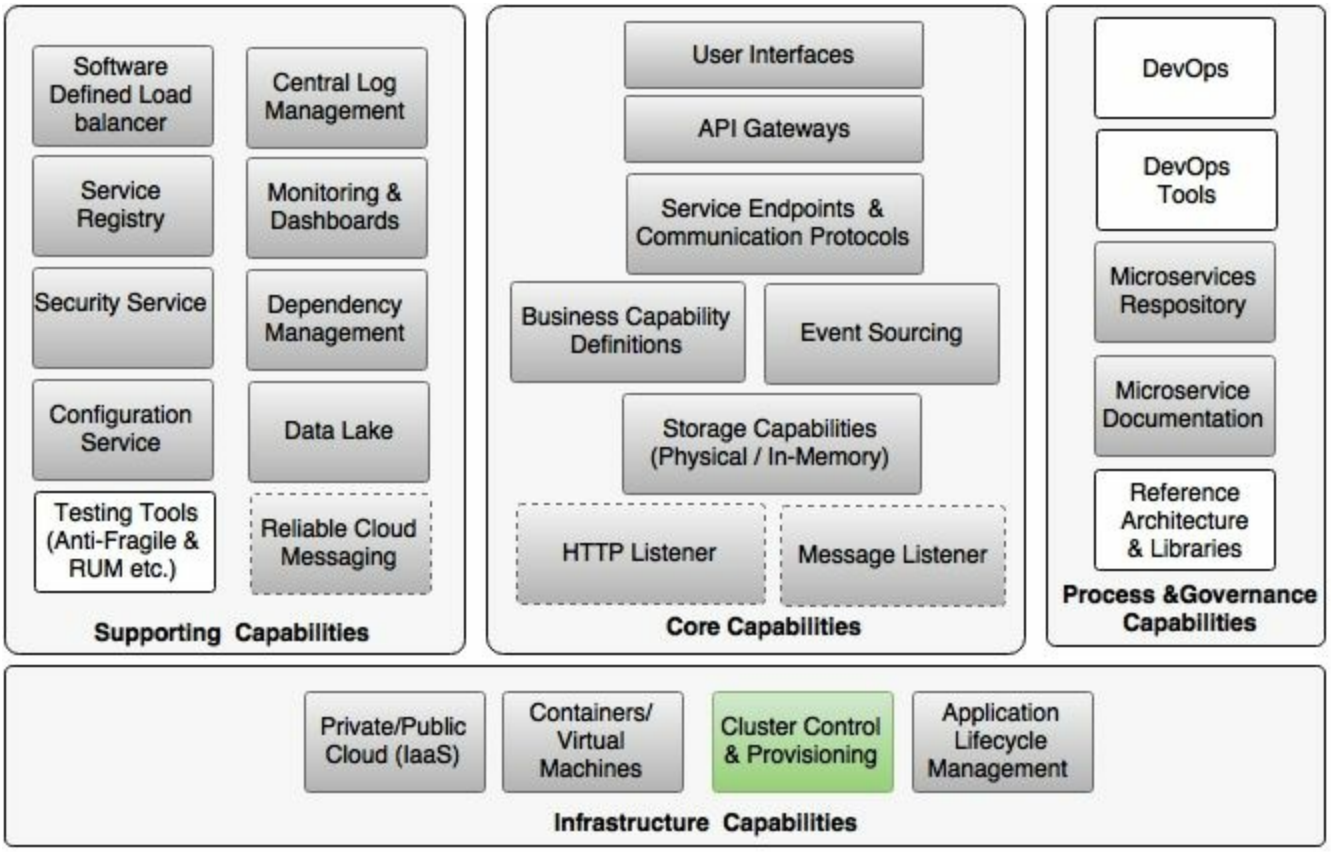
This chapter will explain the need and use of Mesos and Marathon as an infrastructure abstraction layer and a cluster control system, respectively, to achieve optimized resource usage in a cloud-like environment when deploying microservices at scale. This chapter will also provide a step-by-step approach to setting up Mesos and Marathon in a cloud environment. Finally, this chapter will demonstrate how to manage dockerized microservices in the Mesos and Marathon environment.

By the end of this chapter, you will have learned about:

- The need to have an abstraction layer and cluster control software
- Mesos and Marathon from the context of microservices
- Managing dockerized BrownField Airline's PSS microservices with Mesos and Marathon

Reviewing the microservice capability model

In this chapter, we will explore the **Cluster Control & Provisioning** microservices capability from the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*:



The missing pieces

In Chapter 8, *Containerizing Microservices with Docker*, we discussed how to dockerize BrownField Airline's PSS microservices. Docker helped package the JVM runtime and OS parameters along with the application so that there is no special consideration required when moving dockerized microservices from one environment to another. The REST APIs provided by Docker have simplified the life cycle manager's interaction with the target machine in starting and stopping artifacts.

In a large-scale deployment, with hundreds and thousands of Docker containers, we need to ensure that Docker containers run with their own resource constraints, such as memory, CPU, and so on. In addition to this, there may be rules set for Docker deployments, such as replicated copies of the container should not be run on the same machine. Also, a mechanism needs to be in place to optimally use the server infrastructure to avoid incurring extra cost.

There are organizations that deal with billions of containers. Managing them manually is next to impossible. In the context of large-scale Docker deployments, some of the key questions to be answered are:

- How do we manage thousands of containers?
- How do we monitor them?
- How do we apply rules and constraints when deploying artifacts?
- How do we ensure that we utilize containers properly to gain resource efficiency?
- How do we ensure that at least a certain number of minimal instances are running at any point in time?
- How do we ensure dependent services are up and running?
- How do we do rolling upgrades and graceful migrations?
- How do we roll back faulty deployments?

All these questions point to the need to have a solution to address two key capabilities, which are as follows:

- A cluster abstraction layer that provides a uniform abstraction over many physical or virtual machines
- A cluster control and init system to manage deployments intelligently on top of the cluster abstraction

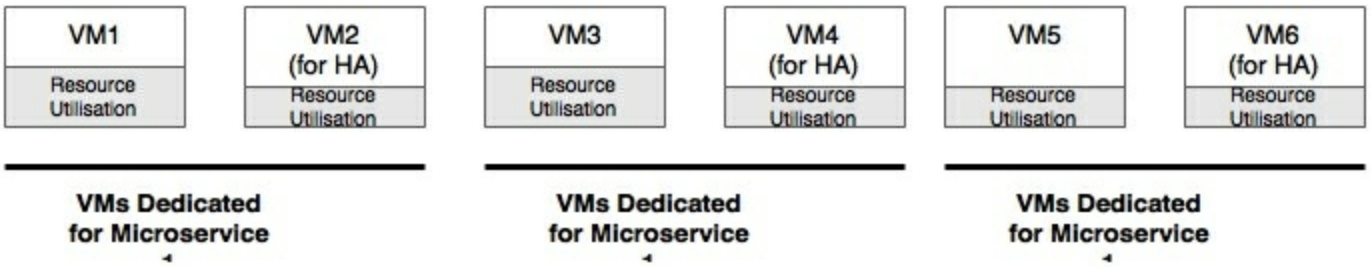
The life cycle manager is ideally placed to deal with these situations. One can add enough intelligence to the life cycle manager to solve these issues. However, before attempting to modify the life cycle manager, it is important to understand the role of cluster management solutions a bit more.

Why cluster management is important

As microservices break applications into different micro-applications, many developers request more server nodes for deployment. In order to manage microservices properly, developers tend to deploy one microservice per VM, which further drives down the resource utilization. In many cases, this results in an overallocation of CPUs and memory.

In many deployments, the high-availability requirements of microservices force engineers to add more and more service instances for redundancy. In reality, though it provides the required high availability, this will result in underutilized server instances.

In general, microservice deployment requires more infrastructure compared to monolithic application deployments. Due to the increase in cost of the infrastructure, many organizations fail to see the value of microservices:



In order to address the issue stated before, we need a tool that is capable of the following:

- Automating a number of activities, such as the allocation of containers to the infrastructure efficiently and keeping it transparent to developers and administrators
- Providing a layer of abstraction for the developers so that they can deploy their application against a data center without knowing which machine is to be used to host their applications
- Setting rules or constraints against deployment artifacts
- Offering higher levels of agility with minimal management overheads for developers and administrators, perhaps with minimal human interaction
- Building, deploying, and managing the application's cost effectively by driving a maximum utilization of the available resources

Containers solve an important issue in this context. Any tool that we select with these capabilities can handle containers in a uniform way, irrespective of the underlying microservice technologies.

What does cluster management do?

Typical cluster management tools help virtualize a set of machines and manage them as a single cluster. Cluster management tools also help move the workload or containers across machines while being transparent to the consumer. Technology evangelists and practitioners use different terminologies, such as cluster orchestration, cluster management, data center virtualization, container schedulers, or container life cycle management, container orchestration, data center operating system, and so on.

Many of these tools currently support both Docker-based containers as well as noncontainerized binary artifact deployments, such as a standalone Spring Boot application. The fundamental function of these cluster management tools is to abstract the actual server instance from the application developers and administrators.

Cluster management tools help the self-service and provisioning of infrastructure rather than requesting the infrastructure teams to allocate the required machines with a predefined specification. In this automated cluster management approach, machines are no longer provisioned upfront and preallocated to the applications. Some of the cluster management tools also help virtualize data centers across many heterogeneous machines or even across data centers, and create an elastic, private cloud-like infrastructure. There is no standard reference model for cluster management tools. Therefore, the capabilities vary between vendors.

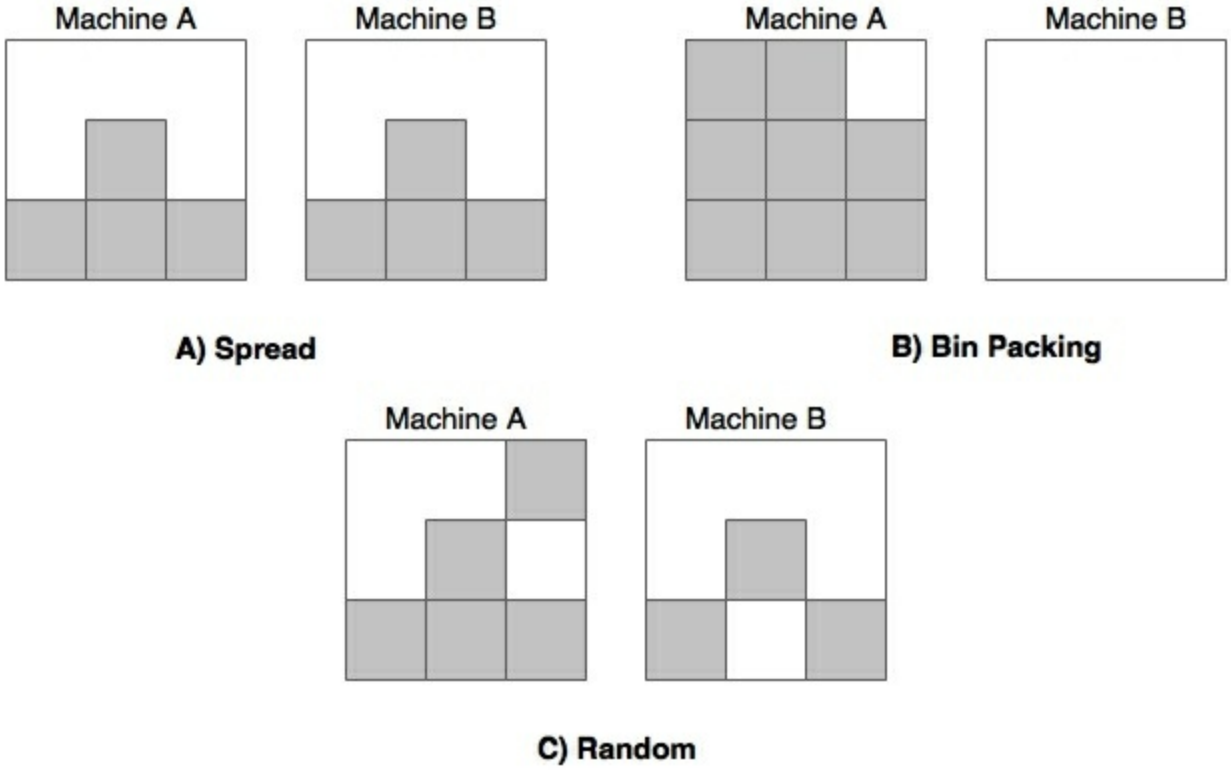
Some of the key capabilities of cluster management software are summarized as follows:

- **Cluster management**: It manages a cluster of VMs and physical machines as a single large machine. These machines could be heterogeneous in terms of resource capabilities, but they are, by and large, machines with Linux as the operating system. These virtual clusters can be formed on the cloud, on-premises, or a combination of both.
- **Deployments**: It handles the automatic deployment of applications and containers with a large set of machines. It supports multiple versions of the application containers and also rolling upgrades across a large number of cluster machines. These tools are also capable of handling the rollback of faulty promotes.
- **Scalability**: It handles the automatic and manual scalability of application instances as and when required, with optimized utilization as the primary goal.
- **Health**: It manages the health of the cluster, nodes, and applications. It removes faulty machines and application instances from the cluster.
- **Infrastructure abstraction**: It abstracts the developers from the actual machine on which the applications are deployed. The developers need not worry about the machine, its capacity, and so on. It is entirely the cluster management software's decision to decide how to schedule and run the applications. These tools also abstract machine details, their capacity, utilization, and location from the developers. For application owners, these are equivalent to a single large machine with almost unlimited capacity.
- **Resource optimization**: The inherent behavior of these tools is to allocate container workloads

across a set of available machines in an efficient way, thereby reducing the cost of ownership. Simple to extremely complicated algorithms can be used effectively to improve utilization.

- **Resource allocation**: It allocates servers based on resource availability and the constraints set by application developers. Resource allocation is based on these constraints, affinity rules, port requirements, application dependencies, health, and so on.
- **Service availability**: It ensures that the services are up and running somewhere in the cluster. In case of a machine failure, cluster control tools automatically handle failures by restarting these services on some other machine in the cluster.
- **Agility**: These tools are capable of quickly allocating workloads to the available resources or moving the workload across machines if there is change in resource requirements. Also, constraints can be set to realign the resources based on business criticality, business priority, and so on.
- **Isolation**: Some of these tools provide resource isolation out of the box. Hence, even if the application is not containerized, resource isolation can be still achieved.

A variety of algorithms are used for resource allocation, ranging from simple algorithms to complex algorithms, with machine learning and artificial intelligence. The common algorithms used are random, bin packing, and spread. Constraints set against applications will override the default algorithms based on resource availability:



The preceding diagram shows how these algorithms fill the available machines with deployments. In this case, it is demonstrated with two machines:

- **Spread**: This algorithm performs the allocation of workload equally across the available

machines. This is showed in diagram **A**.

- **Bin packing**: This algorithm tries to fill in data machine by machine and ensures the maximum utilization of machines. Bin packing is especially good when using cloud services in a pay-as-you-use style. This is shown in diagram **B**.
- **Random**: This algorithm randomly chooses machines and deploys containers on randomly selected machines. This is showed in diagram **C**.

There is a possibility of using cognitive computing algorithms such as machine learning and collaborative filtering to improve efficiency. Techniques such as **oversubscription** allow a better utilization of resources by allocating underutilized resources for high-priority tasks—for example, revenue-generating services for best-effort tasks such as analytics, video, image processing, and so on.

Relationship with microservices

The infrastructure of microservices, if not properly provisioned, can easily result in oversized infrastructures and, essentially, a higher cost of ownership. As discussed in the previous sections, a cloud-like environment with a cluster management tool is essential to realize cost benefits when dealing with large-scale microservices.

The Spring Boot microservices turbocharged with the Spring Cloud project is the ideal candidate workload to leverage cluster management tools. As Spring Cloud-based microservices are location unaware, these services can be deployed anywhere in the cluster. Whenever services come up, they automatically register to the service registry and advertise their availability. On the other hand, consumers always look for the registry to discover the available service instances. This way, the application supports a full fluid structure without preassuming a deployment topology. With Docker, we were able to abstract the runtime so that the services could run on any Linux-based environments.

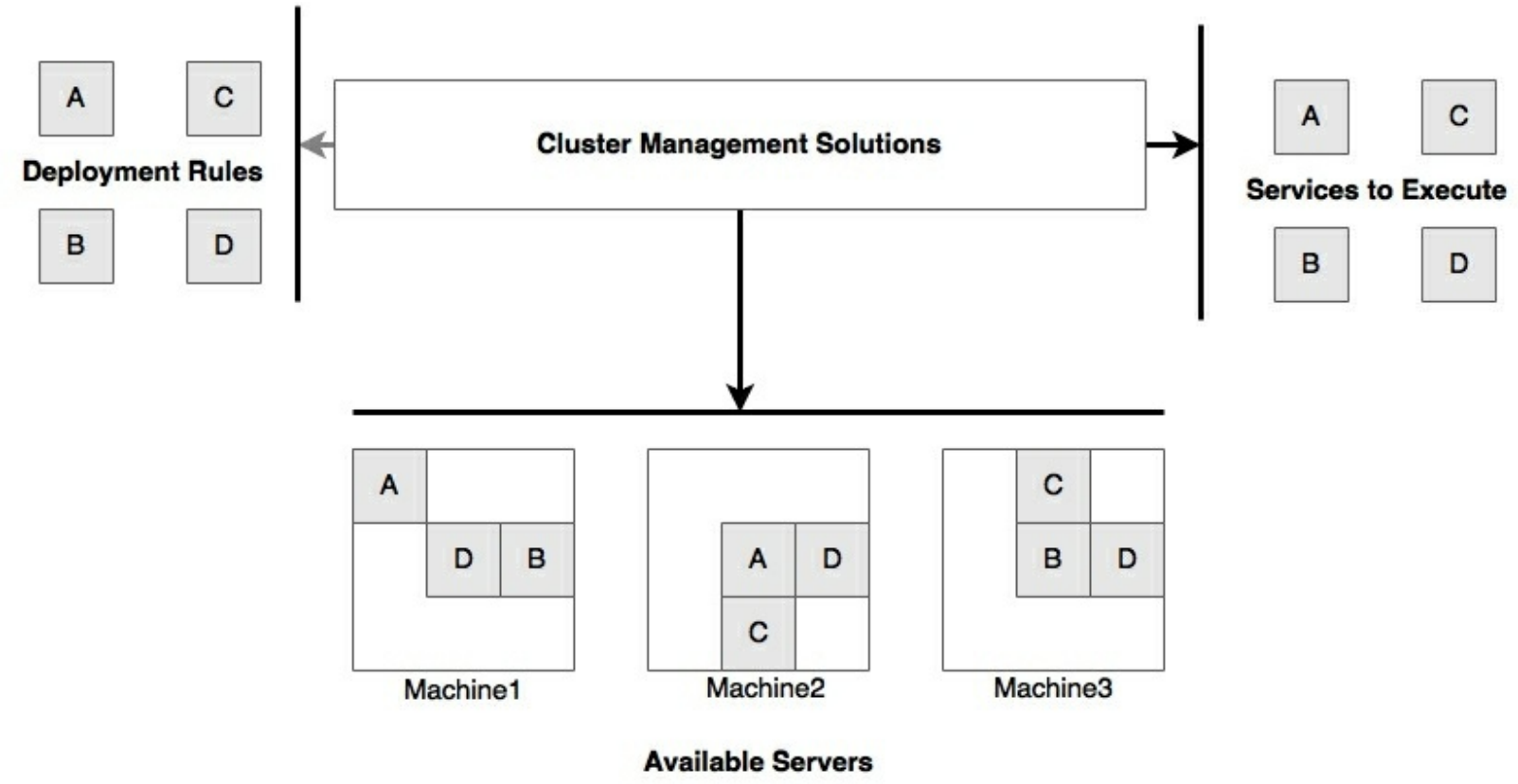
Relationship with virtualization

Cluster management solutions are different from server virtualization solutions in many aspects. Cluster management solutions run on top of VMs or physical machines as an application component.

Cluster management solutions

There are many cluster management software tools available. It is unfair to do an apple-to-apple comparison between them. Even though there are no one-to-one components, there are many areas of overlap in capabilities between them. In many situations, organizations use a combination of one or more of these tools to fulfill their requirements.

The following diagram shows the position of cluster management tools from the microservices context:



In this section, we will explore some of the popular cluster management solutions available on the market.

Docker Swarm

Docker Swarm is Docker's native cluster management solution. Swarm provides a native and deeper integration with Docker and exposes APIs that are compatible with Docker's remote APIs. Docker Swarm logically groups a pool of Docker hosts and manages them as a single large Docker virtual host. Instead of application administrators and developers deciding on which host the container is to be deployed in, this decision making will be delegated to Docker Swarm. Docker Swarm will decide which host to be used based on the bin packing and spread algorithms.

As Docker Swarm is based on Docker's remote APIs, its learning curve for those already using Docker is narrower compared to any other container orchestration tools. However, Docker Swarm is a relatively new product on the market, and it only supports Docker containers.

Docker Swarm works with the concepts of **manager** and **nodes**. A manager is the single point for administrations to interact and schedule the Docker containers for execution. Nodes are where Docker containers are deployed and run.

Kubernetes

Kubernetes (k8s) comes from Google's engineering, is written in the Go language, and is battle-tested for large-scale deployments at Google. Similar to Swarm, Kubernetes helps manage containerized applications across a cluster of nodes. Kubernetes helps automate container deployments, scheduling, and the scalability of containers. Kubernetes supports a number of useful features out of the box, such as automatic progressive rollouts, versioned deployments, and container resiliency if containers fail due to some reason.

The Kubernetes architecture has the concepts of **master**, **nodes**, and **pods**. The master and nodes together form a Kubernetes cluster. The master node is responsible for allocating and managing workload across a number of nodes. Nodes are nothing but a VM or a physical machine. Nodes are further subsegmented as pods. A node can host multiple pods. One or more containers are grouped and executed inside a pod. Pods are also helpful in managing and deploying co-located services for efficiency. Kubernetes also supports the concept of labels as key-value pairs to query and find containers. Labels are user-defined parameters to tag certain types of nodes that execute a common type of workloads, such as frontend web servers. The services deployed on a cluster get a single IP/DNS to access the service.

Kubernetes has out-of-the-box support for Docker; however, the Kubernetes learning curve is steeper compared to Docker Swarm. RedHat offers commercial support for Kubernetes as part of its OpenShift platform.

Apache Mesos

Mesos is an open source framework originally developed by the University of California at Berkeley and is used by Twitter at scale. Twitter uses Mesos primarily to manage the large Hadoop ecosystem.

Mesos is slightly different from the previous solutions. Mesos is more of a resource manager that relies on other frameworks to manage workload execution. Mesos sits between the operating system and the application, providing a logical cluster of machines.

Mesos is a distributed system kernel that logically groups and virtualizes many computers to a single large machine. Mesos is capable of grouping a number of heterogeneous resources to a uniform resource cluster on which applications can be deployed. For these reasons, Mesos is also known as a tool to build a private cloud in a data center.

Mesos has the concepts of the **master** and **slave** nodes. Similar to the earlier solutions, master nodes are responsible for managing the cluster, whereas slaves run the workload. Mesos internally uses ZooKeeper for cluster coordination and storage. Mesos supports the concept of frameworks. These frameworks are responsible for scheduling and running noncontainerized applications and containers. Marathon, Chronos, and Aurora are popular frameworks for the scheduling and execution of applications. Netflix Fenzo is another open source Mesos framework. Interestingly, Kubernetes also can be used as a Mesos framework.

Marathon supports the Docker container as well as noncontainerized applications. Spring Boot can be directly configured in Marathon. Marathon provides a number of capabilities out of the box, such as supporting application dependencies, grouping applications to scale and upgrade services, starting and shutting down healthy and unhealthy instances, rolling out promotes, rolling back failed promotes, and so on.

Mesosphere offers commercial support for Mesos and Marathon as part of its DCOS platform.

Nomad

Nomad from HashiCorp is another cluster management software. Nomad is a cluster management system that abstracts lower-level machine details and their locations. Nomad has a simpler architecture compared to the other solutions explored earlier. Nomad is also lightweight. Similar to other cluster management solutions, Nomad takes care of resource allocation and the execution of applications. Nomad also accepts user-specific constraints and allocates resources based on this.

Nomad has the concept of **servers**, in which all jobs are managed. One server acts as the **leader**, and others act as **followers**. Nomad has the concept of **tasks**, which is the smallest unit of work. Tasks are grouped into **task groups**. A task group has tasks that are to be executed in the same location. One or more task groups or tasks are managed as **jobs**.

Nomad supports many workloads, including Docker, out of the box. Nomad also supports deployments across data centers and is region and data center aware.

Fleet

Fleet is a cluster management system from CoreOS. It runs on a lower level and works on top of systemd. Fleet can manage application dependencies and make sure that all the required services are running somewhere in the cluster. If a service fails, it restarts the service on another host. Affinity and constraint rules are possible to supply when allocating resources.

Fleet has the concepts of **engine** and **agents**. There is only one engine at any point in the cluster with multiple agents. Tasks are submitted to the engine and agent run these tasks on a cluster machine.

Fleet also supports Docker out of the box.

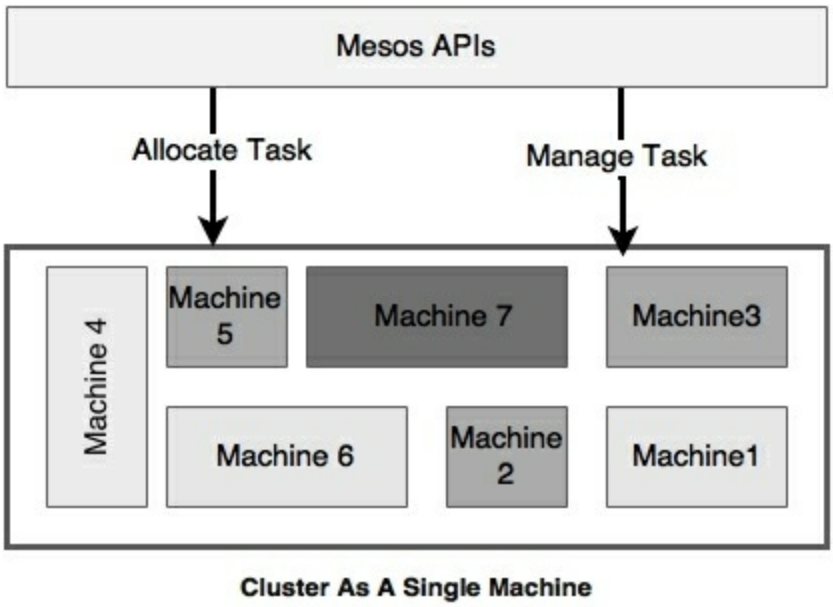
Cluster management with Mesos and Marathon

As we discussed in the previous section, there are many cluster management solutions or container orchestration tools available. Different organizations choose different solutions to address problems based on their environment. Many organizations choose Kubernetes or Mesos with a framework such as Marathon. In most cases, Docker is used as a default containerization method to package and deploy workloads.

For the rest of this chapter, we will show how Mesos works with Marathon to provide the required cluster management capability. Mesos is used by many organizations, including Twitter, Airbnb, Apple, eBay, Netflix, PayPal, Uber, Yelp, and many others.

Diving deep into Mesos

Mesos can be treated as a data center kernel. DCOS is the commercial version of Mesos supported by Mesosphere. In order to run multiple tasks on one node, Mesos uses resource isolation concepts. Mesos relies on the Linux kernel's **cgroups** to achieve resource isolation similar to the container approach. It also supports containerized isolation using Docker. Mesos supports both batch workload as well as the OLTP kind of workloads:

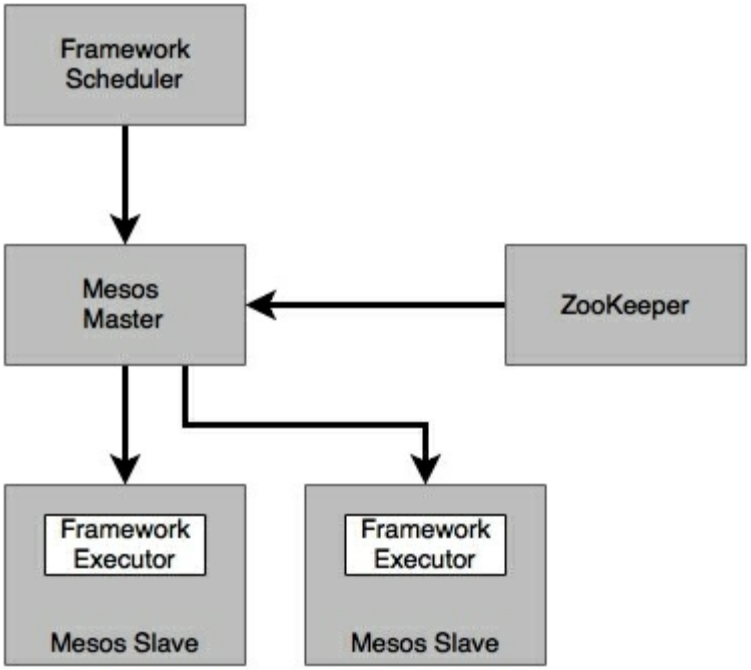


Mesos is an open source top-level Apache project under the Apache license. Mesos abstracts lower-level computing resources such as CPU, memory, and storage from lower-level physical or virtual machines.

Before we examine why we need both Mesos and Marathon, let's understand the Mesos architecture.

The Mesos architecture

The following diagram shows the simplest architectural representation of Mesos. The key components of Mesos includes a Mesos master node, a set of slave nodes, a ZooKeeper service, and a Mesos framework. The Mesos framework is further subdivided into two components: a scheduler and an executor:



The boxes in the preceding diagram are explained as follows:

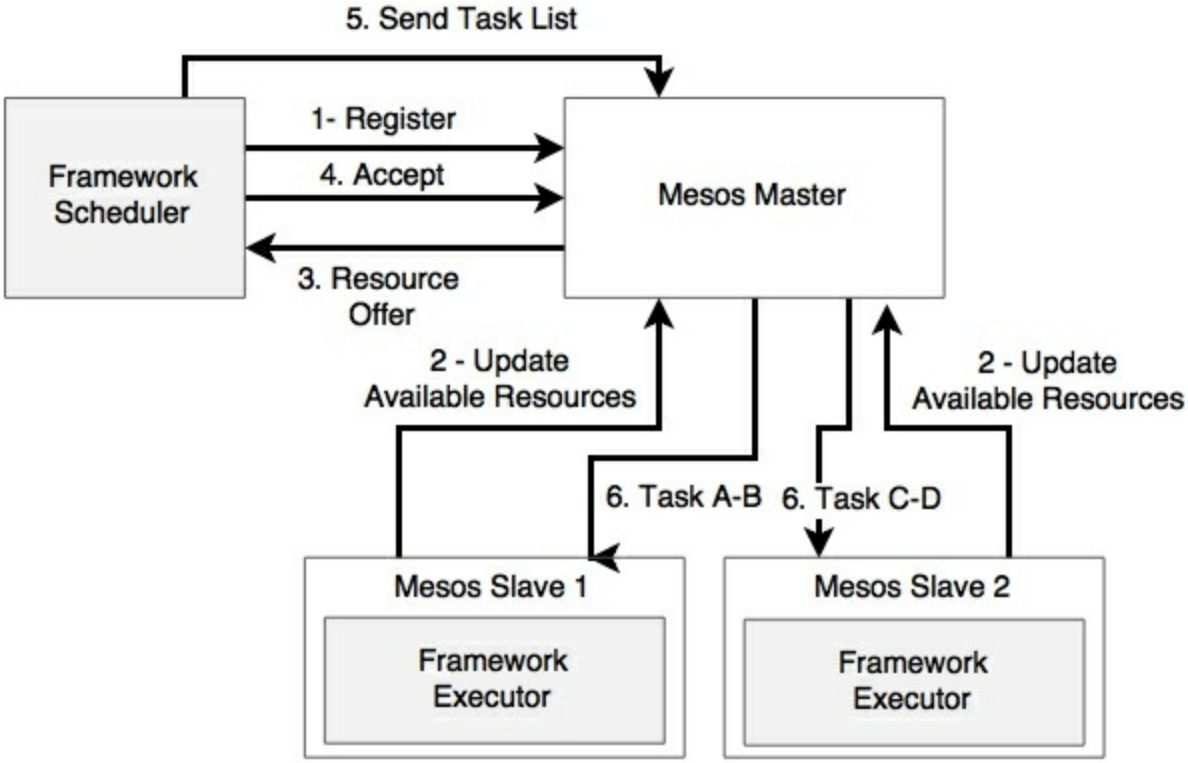
- **Master**: The Mesos master is responsible for managing all the Mesos slaves. The Mesos master gets information on the resource availability from all slave nodes and take the responsibility of filling the resources appropriately based on certain resource policies and constraints. The Mesos master preempts available resources from all slave machines and pools them as a single large machine. The master offers resources to frameworks running on slave machines based on this resource pool.

  For high availability, the Mesos master is supported by the Mesos master's standby components. Even if the master is not available, the existing tasks can still be executed. However, new tasks cannot be scheduled in the absence of a master node. The master standby nodes are nodes that wait for the failure of the active master and take over the master's role in the case of a failure. It uses ZooKeeper for the master leader election. A minimum quorum requirement must be met for leader election.
- **Slave**: Mesos slaves are responsible for hosting task execution frameworks. Tasks are executed on the slave nodes. Mesos slaves can be started with attributes as key-value pairs, such as *data center = X*. This is used for constraint evaluations when deploying workloads. Slave machines share resource availability with the Mesos master.
- **ZooKeeper**: ZooKeeper is a centralized coordination server used in Mesos to coordinate activities across the Mesos cluster. Mesos uses ZooKeeper for leader election in case of a Mesos master failure.
- **Framework**: The Mesos framework is responsible for understanding the application's constraints, accepting resource offers from the master, and finally running tasks on the slave resources offered by the master. The Mesos framework consists of two components: the framework scheduler and the framework executor:
  - The scheduler is responsible for registering to Mesos and handling resource offers

- The executor runs the actual program on Mesos slave nodes

The framework is also responsible for enforcing certain policies and constraints. For example, a constraint can be, let's say, that a minimum of 500 MB of RAM is available for execution.

Frameworks are pluggable components and are replaceable with another framework. The framework workflow is depicted in the following diagram:



The steps denoted in the preceding workflow diagram are elaborated as follows:

1. The framework registers with the Mesos master and waits for resource offers. The scheduler may have many tasks in its queue to be executed with different resource constraints (tasks **A** to **D**, in this example). A task, in this case, is a unit of work that is scheduled—for example, a Spring Boot microservice.
2. The Mesos slave offers the available resources to the Mesos master. For example, the slave advertises the CPU and memory available with the slave machine.
3. The Mesos master then creates a resource offer based on the allocation policies set and offers it to the scheduler component of the framework. Allocation policies determine which framework the resources are to be offered to and how many resources are to be offered. The default policies can be customized by plugging additional allocation policies.
4. The scheduler framework component, based on the constraints, capabilities, and policies, may accept or reject the resource offering. For example, a framework rejects the resource offer if the resources are insufficient as per the constraints and policies set.
5. If the scheduler component accepts the resource offer, it submits the details of one more task to the Mesos master with resource constraints per task. Let's say, in this example, that it is ready to submit tasks **A** to **D**.

6. The Mesos master sends this list of tasks to the slave where the resources are available. The framework executor component installed on the slave machines picks up and runs these tasks.

Mesos supports a number of frameworks, such as:

- Marathon and Aurora for **long-running** processes, such as web applications
- Hadoop, Spark, and Storm for **big data** processing
- Chronos and Jenkins for **batch scheduling**
- Cassandra and Elasticsearch for **data management**

In this chapter, we will use Marathon to run dockerized microservices.

Marathon

Marathon is one of the Mesos framework implementations that can run both container as well as noncontainer execution. Marathon is particularly designed for long-running applications, such as a web server. Marathon ensures that the service started with Marathon continues to be available even if the Mesos slave it is hosted on fails. This will be done by starting another instance.

Marathon is written in Scala and is highly scalable. Marathon offers a UI as well as REST APIs to interact with Marathon, such as the start, stop, scale, and monitoring applications.

Similar to Mesos, Marathon's high availability is achieved by running multiple Marathon instances pointing to a ZooKeeper instance. One of the Marathon instances acts as a leader, and others are in standby mode. In case the leading master fails, a leader election will take place, and the next active master will be determined.

Some of the basic features of Marathon include:

- Setting resource constraints
- Scaling up, scaling down, and the instance management of applications
- Application version management
- Starting and killing applications

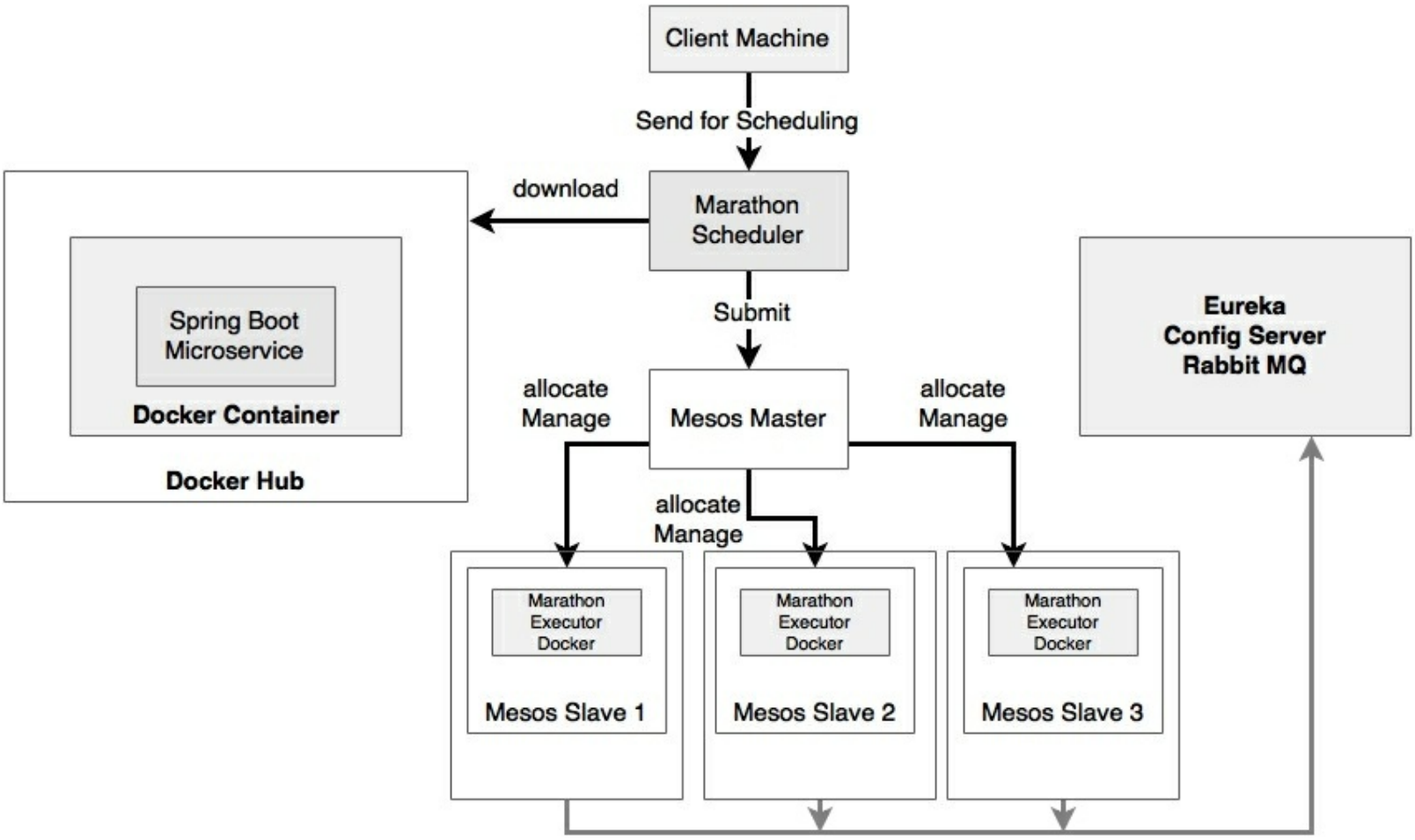
Some of the advanced features of Marathon include:

- Rolling upgrades, rolling restarts, and rollbacks
- Blue-green deployments

Implementing Mesos and Marathon for BrownField microservices

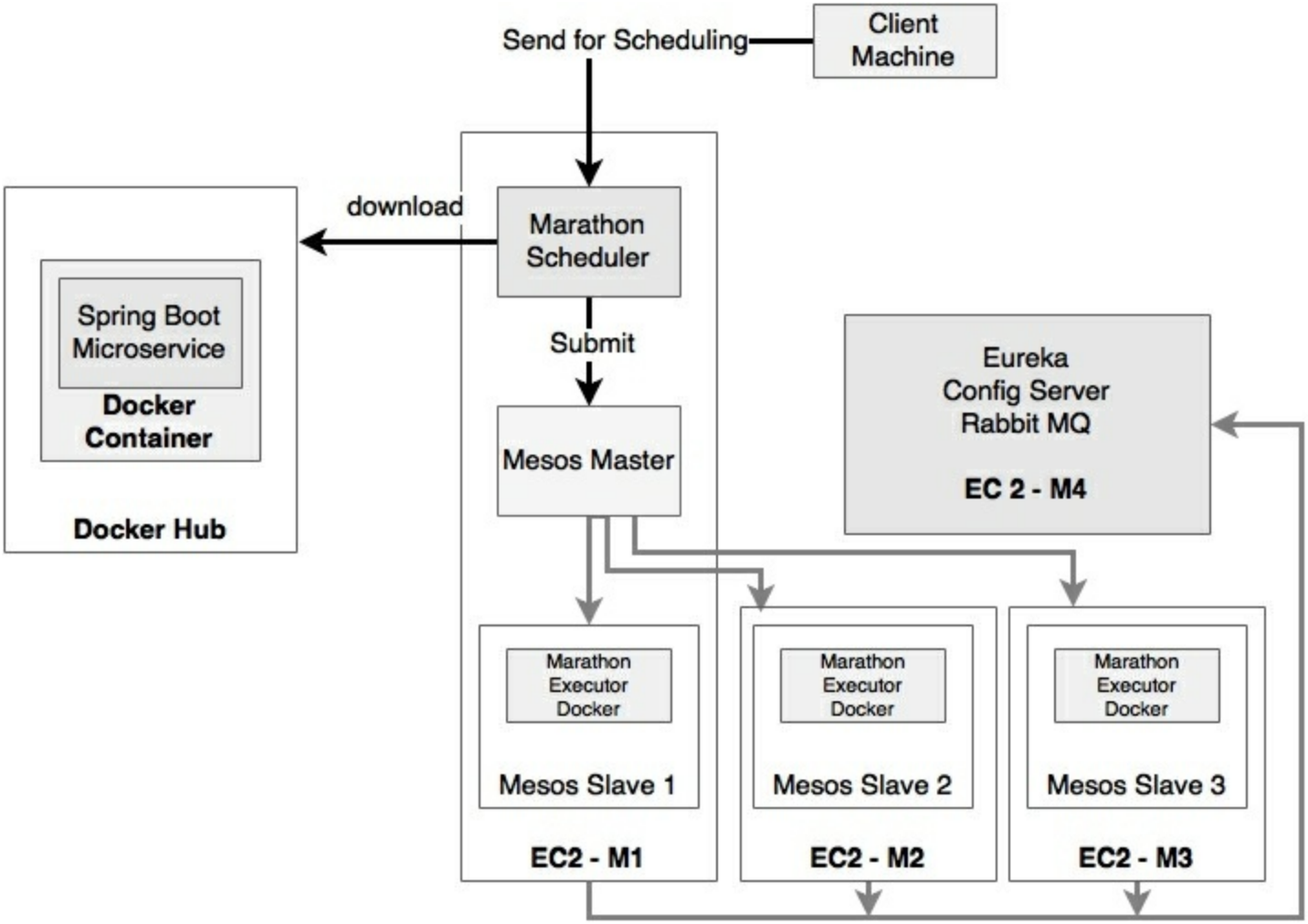
In this section, the dockerized Brownfield microservice developed in Chapter 8, *Containerizing Microservices with Docker*, will be deployed into the AWS cloud and managed with Mesos and Marathon.

For the purposes of demonstration, only three of the services (**Search**, **Search API Gateway**, and **Website**) are covered in the explanations:



The logical architecture of the target state implementation is shown in the preceding diagram. The implementation uses multiple Mesos slaves to execute dockerized microservices with a single Mesos master. The Marathon scheduler component is used to schedule dockerized microservices. Dockerized microservices are hosted on the Docker Hub registry. Dockerized microservices are implemented using Spring Boot and Spring Cloud.

The following diagram shows the physical deployment architecture:



As shown in the preceding diagram, in this example, we will use four EC2 instances:

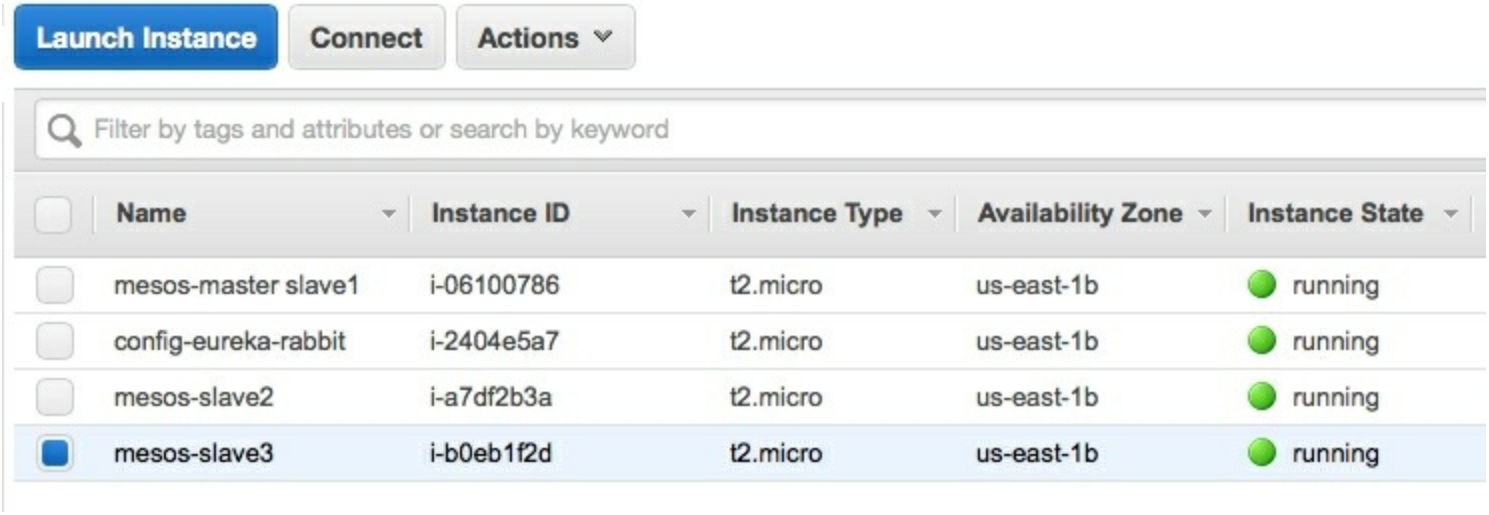
- **EC2-M1**: This hosts the Mesos master, ZooKeeper, the Marathon scheduler, and one Mesos slave instance
- **EC2-M2**: This hosts one Mesos slave instance
- **EC2-M3**: This hosts another Mesos slave instance
- **EC2-M4**: This hosts Eureka, Config server, and RabbitMQ

For a real production setup, multiple Mesos masters as well as multiple instances of Marathon are required for fault tolerance.

Setting up AWS

Launch the four **t2.micro** EC2 instances that will be used for this deployment. All four instances have to be on the same security group so that the instances can see each other using their local IP addresses.

The following tables show the machine details and IP addresses for indicative purposes and to link subsequent instructions:



Instance ID	Private DNS/IP	Public DNS/IP
i-06100786	ip-172-31-54-69.ec2.internal 172.31.54.69	ec2-54-85-107-37.compute-1.amazonaws.com 54.85.107.37
i-2404e5a7	ip-172-31-62-44.ec2.internal 172.31.62.44	ec2-52-205-251-150.compute-1.amazonaws.com 52.205.251.150
i-a7df2b3a	ip-172-31-49-55.ec2.internal 172.31.49.55	ec2-54-172-213-51.compute-1.amazonaws.com 54.172.213.51
i-b0eb1f2d	ip-172-31-53-109.ec2.internal 172.31.53.109	ec2-54-86-31-240.compute-1.amazonaws.com 54.86.31.240

Replace the IP and DNS addresses based on your AWS EC2 configuration.

Installing ZooKeeper, Mesos, and Marathon

The following software versions will be used for the deployment. The deployment in this section follows the physical deployment architecture explained in the earlier section:

- Mesos version 0.27.1
- Docker version 1.6.2, build 7c8fca2
- Marathon version 0.15.3

> **Note**
>
> The detailed instructions to set up ZooKeeper, Mesos, and Marathon are available at https://open.mesosphere.com/getting-started/install/.

Perform the following steps for a minimal installation of ZooKeeper, Mesos, and Marathon to deploy the BrownField microservice:

1. As a prerequisite, JRE 8 must be installed on all the machines. Execute the following command:

```
sudo apt-get -y install oracle-java8-installer
```

2. Install Docker on all machines earmarked for the Mesos slave via the following command:

```
sudo apt-get install docker
```

3. Open a terminal window and execute the following commands. These commands set up the repository for installation:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
# Add the repository
echo "deb http://repos.mesosphere.com/${DISTRO} ${CODENAME} main" | \
  sudo tee /etc/apt/sources.list.d/mesosphere.list
sudo apt-get -y update
```

4. Execute the following command to install Mesos and Marathon. This will also install Zookeeper as a dependency:

```
sudo apt-get -y install mesos marathon
```

Repeat the preceding steps on all the three EC2 instances reserved for the Mesos slave execution. As the next step, ZooKeeper and Mesos have to be configured on the machine identified for the Mesos master.

Configuring ZooKeeper

Connect to the machine reserved for the Mesos master and Marathon scheduler. In this case, `172.31.54.69` will be used to set up ZooKeeper, the Mesos master, and Marathon.

There are two configuration changes required in ZooKeeper, as follows:

1. The first step is to set `/etc/zookeeper/conf/myid` to a unique integer between `1` and `255`, as follows:

   Open vi /etc/zookeeper/conf/myid and set 1.

2. The next step is to edit `/etc/zookeeper/conf/zoo.cfg`. Update the file to reflect the following changes:

   ```
   # specify all zookeeper servers
   # The first port is used by followers to connect to the leader
   # The second one is used for leader election
   server.1= 172.31.54.69:2888:3888
   #server.2=zookeeper2:2888:3888
   #server.3=zookeeper3:2888:3888
   ```

   Replace the IP addresses with the relevant private IP address. In this case, we will use only one ZooKeeper server, but in a production scenario, multiple servers are required for high availability.

Configuring Mesos

Make changes to the Mesos configuration to point to ZooKeeper, set up a quorum, and enable Docker support via the following steps:

1. Edit `/etc/mesos/zk` to set the following value. This is to point Mesos to a ZooKeeper instance for quorum and leader election:

   ```
   zk:// 172.31.54.69:2181/mesos
   ```

2. Edit the `/etc/mesos-master/quorum` file and set the value as `1`. In a production scenario, we may need a minimum quorum of three:

   vi /etc/mesos-master/quorum

3. The default Mesos installation does not support Docker on Mesos slaves. In order to enable Docker, update the following `mesos-slave` configuration:

   echo 'docker,mesos' > /etc/mesos-slave/containerizers

Running Mesos, Marathon, and ZooKeeper as services

All the required configuration changes are implemented. The easiest way to start Mesos, Marathon, and Zookeeper is to run them as services, as follows:

- The following commands start services. The services need to be started in the following order:

  ```
  sudo service zookeeper start
  sudo service mesos-master start
  sudo service mesos-slave start
  sudo service marathon start
  ```

- At any point, the following commands can be used to stop these services:

```
sudo service zookeeper stop
sudo service mesos-master stop
sudo service mesos-slave stop
sudo service marathon stop
```

- Once the services are up and running, use a terminal window to verify whether the services are running:

```
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep zookeeper
ubuntu    1240  1180  0 16:33 pts/0    00:00:00 grep --color=auto zookeeper
zookeep+ 17056     1  0 Apr10 ?        00:00:54 /usr/bin/java -cp /etc/zookeeper/conf:/usr/share/java/jline.jar:/usr/
share/java/log4j-1.2.jar:/usr/share/java/xercesImpl.jar:/usr/share/java/xmlParserAPIs.jar:/usr/share/java/netty.jar:/
usr/share/java/slf4j-api.jar:/usr/share/java/slf4j-log4j12.jar:/usr/share/java/zookeeper.jar -Dcom.sun.management.jmx
remote -Dcom.sun.management.jmxremote.local.only=false -Dzookeeper.log.dir=/var/log/zookeeper -Dzookeeper.root.logger
=INFO,ROLLINGFILE org.apache.zookeeper.server.quorum.QuorumPeerMain /etc/zookeeper/conf/zoo.cfg
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep sbin/mesos-master
ubuntu    1245  1180  0 16:33 pts/0    00:00:00 grep --color=auto sbin/mesos-master
root     17066     1  0 Apr10 ?        00:00:28 /usr/sbin/mesos-master --zk=zk://172.31.54.69:2181/mesos  --port=5050
 --log_dir=/var/log/mesos --quorum=1 --work_dir=/var/lib/mesos
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep marathon.Main
ubuntu    1252  1180  0 16:33 pts/0    00:00:00 grep --color=auto marathon.Main
root     17117     1  0 Apr10 ?        00:04:07 java -Djava.library.path=/usr/local/lib:/usr/lib:/usr/lib64 -Djava.ut
il.logging.SimpleFormatter.format=%2$s%5$s%6$s%n -Xmx512m -cp /usr/bin/marathon mesosphere.marathon.Main --zk zk://17
2.31.54.69:2181/marathon --master zk://172.31.54.69:2181/mesos
ubuntu@ip-172-31-54-69:~$
```

Running the Mesos slave in the command line

In this example, instead of using the Mesos slave service, we will use a command-line version to invoke the Mesos slave to showcase additional input parameters. Stop the Mesos slave and use the command line as mentioned here to start the slave again:

```
$sudo service mesos-slave stop

$sudo /usr/sbin/mesos-slave  --master=172.31.54.69:5050 --log_dir=/var/log/mesos
--work_dir=/var/lib/mesos --containerizers=mesos,docker --resources="ports(*):
[8000-9000, 31000-32000]"
```

The command-line parameters used are explained as follows:

- `--master=172.31.54.69:5050`: This parameter is to tell the Mesos slave to connect to the correct Mesos master. In this case, there is only one master running at `172.31.54.69:5050`. All the slaves connect to the same Mesos master.
- `--containerizers=mesos,docker`: This parameter is to enable support for Docker container execution as well as noncontainerized executions on the Mesos slave instances.
- `--resources="ports(*):[8000-9000, 31000-32000]`: This parameter indicates that the slave can offer both ranges of ports when binding resources. `31000` to `32000` is the default range. As we are using port numbers starting with `8000`, it is important to tell the Mesos slave to

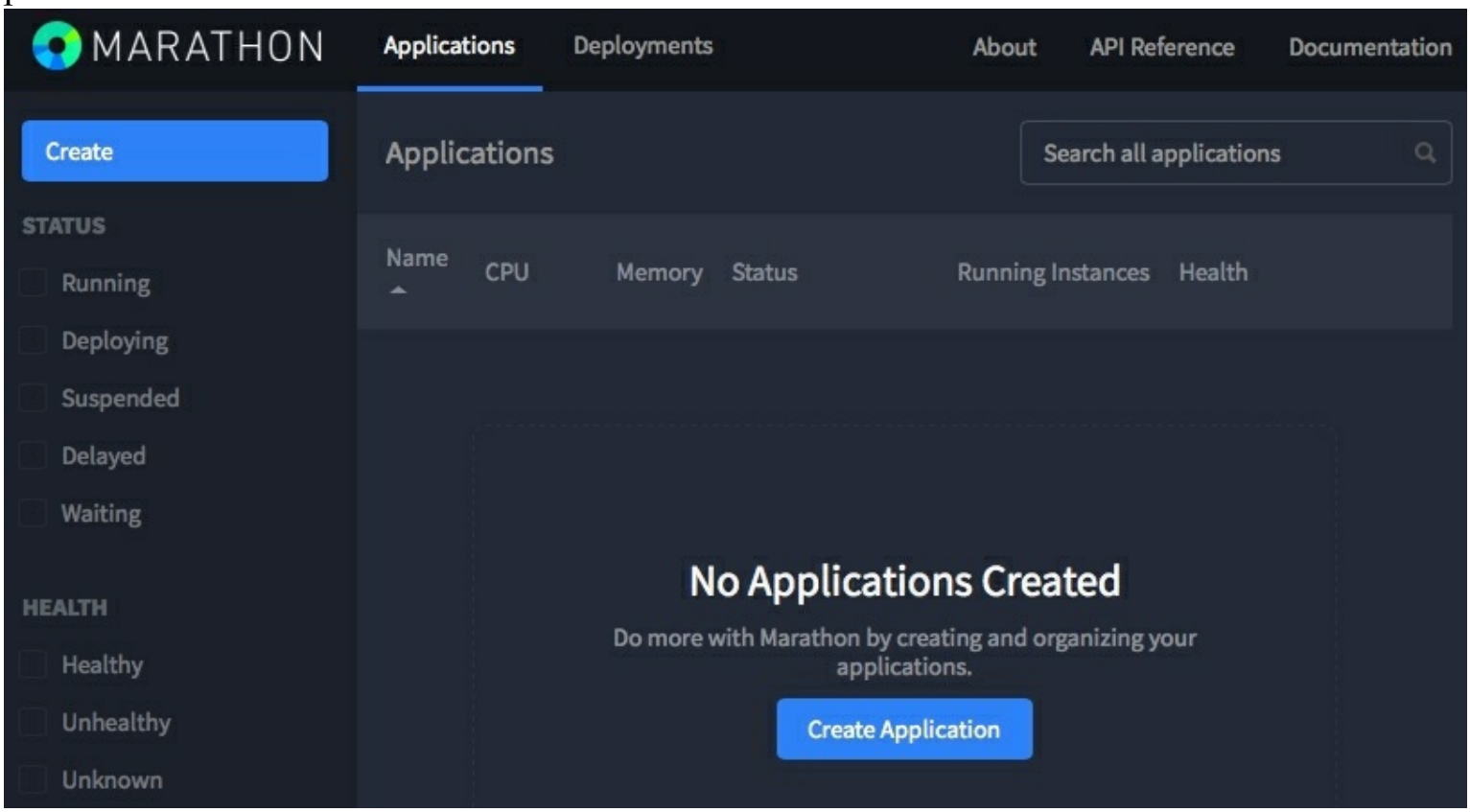
allow exposing ports starting from `8000` as well.

Perform the following steps to verify the installation of Mesos and Marathon:

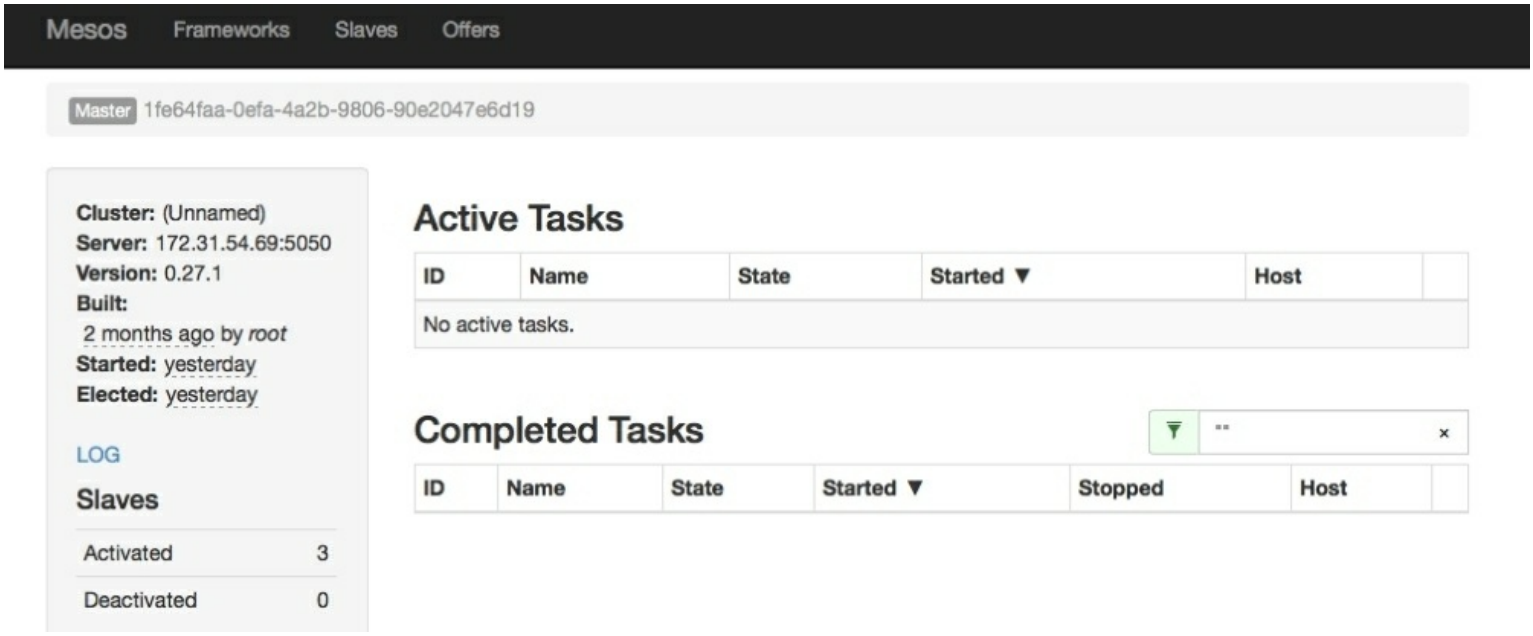
1. Execute the command mentioned in the previous step to start the Mesos slave on all the three instances designated for the slave. The same command can be used across all three instances as all of them connect to the same master.
2. If the Mesos slave is successfully started, a message similar to the following will appear in the console:

```
I0411 18:11:39.684809 16665 slave.cpp:1030] Forwarding total oversubscribed resources
```

   The preceding message indicates that the Mesos slave started sending the current state of resource availability periodically to the Mesos master.
3. Open `http://54.85.107.37:8080` to inspect the Marathon UI. Replace the IP address with the public IP address of the EC2 instance:



   As there are no applications deployed so far, the **Applications** section of the UI is empty.
4. Open the Mesos UI, which runs on port `5050`, by going to `http://54.85.107.37:5050`:



The **Slaves** section of the console shows that there are three activated Mesos slaves available for execution. It also indicates that there is no active task.

Preparing BrownField PSS services

In the previous section, we successfully set up Mesos and Marathon. In this section, we will take a look at how to deploy the BrownField PSS application previously developed using Mesos and Marathon.

Note

The full source code of this chapter is available under the `Chapter 9` project in the code files. Copy `chapter8.configserver`, `chapter8.eurekaserver`, `chapter8.search`, `chapter8.search-apigateway`, and `chapter8.website` into a new STS workspace and rename them `chapter9.*`.

1. Before we deploy any application, we have to set up the Config server, Eureka server, and RabbitMQ in one of the servers. Follow the steps described in the *Running BrownField services on EC2* section in Chapter 8, *Containerizing Microservices with Docker*. Alternately, we can use the same instance as used in the previous chapter for this purpose.
2. Change all `bootstrap.properties` files to reflect the Config server IP address.
3. Before we deploy our services, there are a few specific changes required on the microservices. When running dockerized microservices with the BRIDGE mode on, we need to tell the Eureka client the hostname to be used to bind. By default, Eureka uses the **instance ID** to register. However, this is not helpful as Eureka clients won't be able to look up these services using the instance ID. In the previous chapter, the HOST mode was used instead of the BRIDGE mode.

   The hostname setup can be done using the `eureka.instance.hostname` property. However, when running on AWS specifically, an alternate approach is to define a bean in the

microservices to pick up AWS-specific information, as follows:

```
@Configuration
class EurekaConfig {
@Bean
    public EurekaInstanceConfigBean eurekaInstanceConfigBean() {
    EurekaInstanceConfigBean config = new EurekaInstanceConfigBean(new
InetUtils(new InetUtilsProperties()));
AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
        config.setDataCenterInfo(info);

info.getMetadata().put(AmazonInfo.MetaDataKey.publicHostname.getName(),
info.get(AmazonInfo.MetaDataKey.publicIpv4));
        config.setHostname(info.get(AmazonInfo.MetaDataKey.localHostname));
config.setNonSecurePortEnabled(true);
config.setNonSecurePort(PORT);
config.getMetadataMap().put("instanceId",
info.get(AmazonInfo.MetaDataKey.localHostname));
return config;
}
```

The preceding code provides a custom Eureka server configuration using the Amazon host information using Netflix APIs. The code overrides the hostname and instance ID with the private DNS. The port is read from the Config server. This code also assumes one host per service so that the port number stays constant across multiple deployments. This can also be overridden by dynamically reading the port binding information at runtime.

The previous code has to be applied in all microservices.

4. Rebuild all the microservices using Maven. Build and push the Docker images to the Docker Hub. The steps for the three services are shown as follows. Repeat the same steps for all the other services. The working directory needs to be switched to the respective directories before executing these commands:

```
docker build -t search-service:1.0 .
docker tag search-service:1.0 rajeshrv/search-service:1.0
docker push rajeshrv/search-service:1.0

docker build -t search-apigateway:1.0 .
docker tag search-apigateway:1.0 rajeshrv/search-apigateway:1.0
docker push rajeshrv/search-apigateway:1.0

docker build -t website:1.0 .
docker tag website:1.0 rajeshrv/website:1.0
docker push rajeshrv/website:1.0
```

Deploying BrownField PSS services

The Docker images are now published to the Docker Hub registry. Perform the following steps to deploy and run BrownField PSS services:

1. Start the Config server, Eureka server, and RabbitMQ on its dedicated instance.
2. Make sure that the Mesos server and Marathon are running on the machine where the Mesos master is configured.
3. Run the Mesos slave on all the machines as described earlier using the command line.
4. At this point, the Mesos Marathon cluster is up and running and is ready to accept deployments. The deployment can be done by creating one JSON file per service, as shown here:

```
{
  "id": "search-service-1.0",
  "cpus": 0.5,
  "mem": 256.0,
  "instances": 1,
  "container": {
   "docker": {
    "type": "DOCKER",
      "image": "rajeshrv/search-service:1.0",
       "network": "BRIDGE",
       "portMappings": [
        {  "containerPort": 0, "hostPort": 8090 }
      ]
    }
  }
}
```

The preceding JSON code will be stored in the `search.json` file. Similarly, create a JSON file for other services as well.

The JSON structure is explained as follows:

- `id`: This is the unique ID of the application. This can be a logical name.
- `cpus` and `mem`: This sets the resource constraints for this application. If the resource offer does not satisfy this resource constraint, Marathon will reject this resource offer from the Mesos master.
- `instances`: This decides how many instances of this application to start with. In the preceding configuration, by default, it starts one instance as soon as it gets deployed. Marathon maintains the number of instances mentioned at any point.
- `container`: This parameter tells the Marathon executor to use a Docker container for execution.
- `image`: This tells the Marathon scheduler which Docker image has to be used for deployment. In this case, this will download the `search-service:1.0` image from the Docker Hub repository `rajeshrv`.
- `network`: This value is used for Docker runtime to advise on the network mode to be used when starting the new docker container. This can be BRIDGE or HOST. In this case, the BRIDGE mode will be used.
- `portMappings`: The port mapping provides information on how to map the internal and external ports. In the preceding configuration, the host port is set as `8090`, which tells the Marathon executor to use `8090` when starting the service. As the container port is set as `0`, the same host port will be assigned to the container. Marathon picks up random ports if the

host port value is `0`.

5. Additional health checks are also possible with the JSON descriptor, as shown here:

```
"healthChecks": [
    {
      "protocol": "HTTP",
      "portIndex": 0,
      "path": "/admin/health",
      "gracePeriodSeconds": 100,
      "intervalSeconds": 30,
      "maxConsecutiveFailures": 5
    }
  ]
```

6. Once this JSON code is created and saved, deploy it to Marathon using the Marathon REST APIs as follows:

```
curl -X POST http://54.85.107.37:8080/v2/apps -d @search.json -H "Content-type: application/json"
```

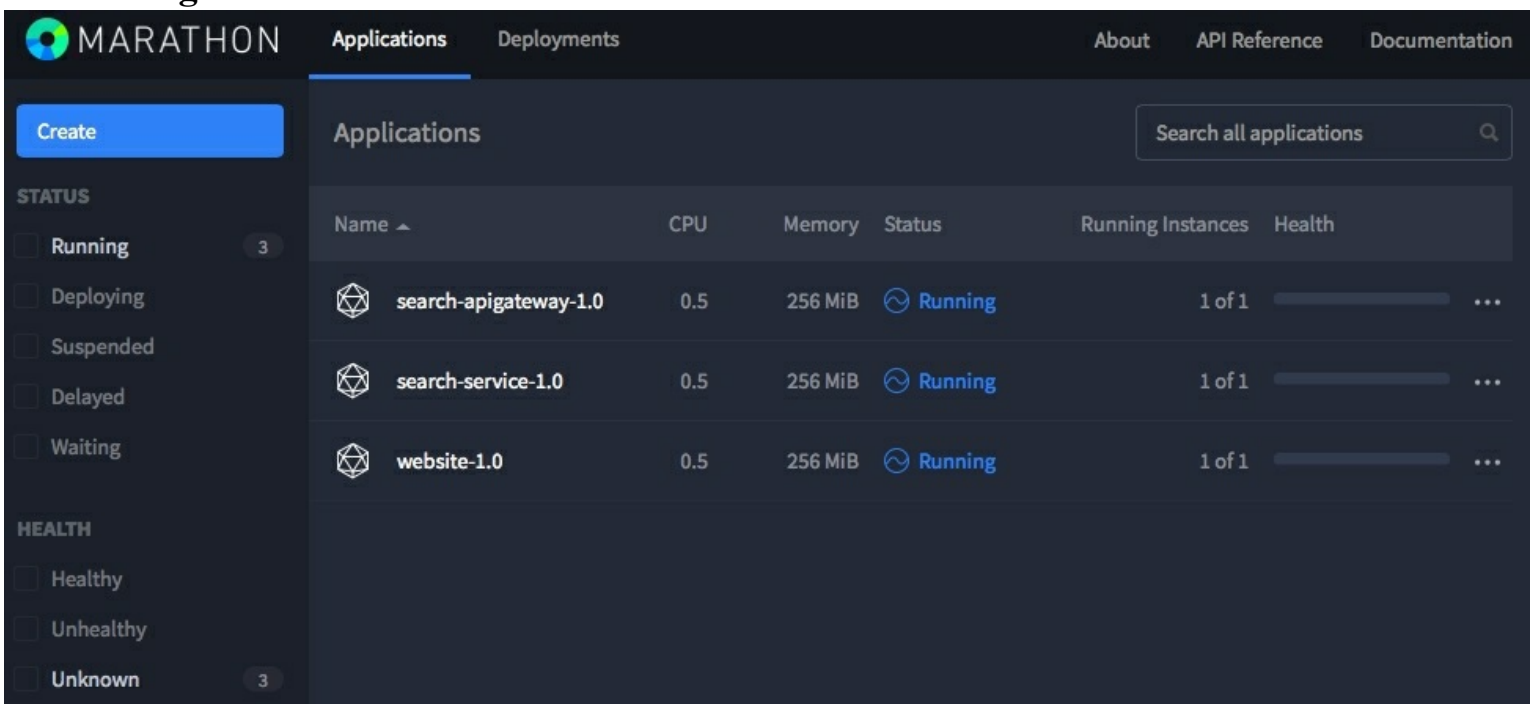
Repeat this step for all the other services as well.

The preceding step will automatically deploy the Docker container to the Mesos cluster and start one instance of the service.

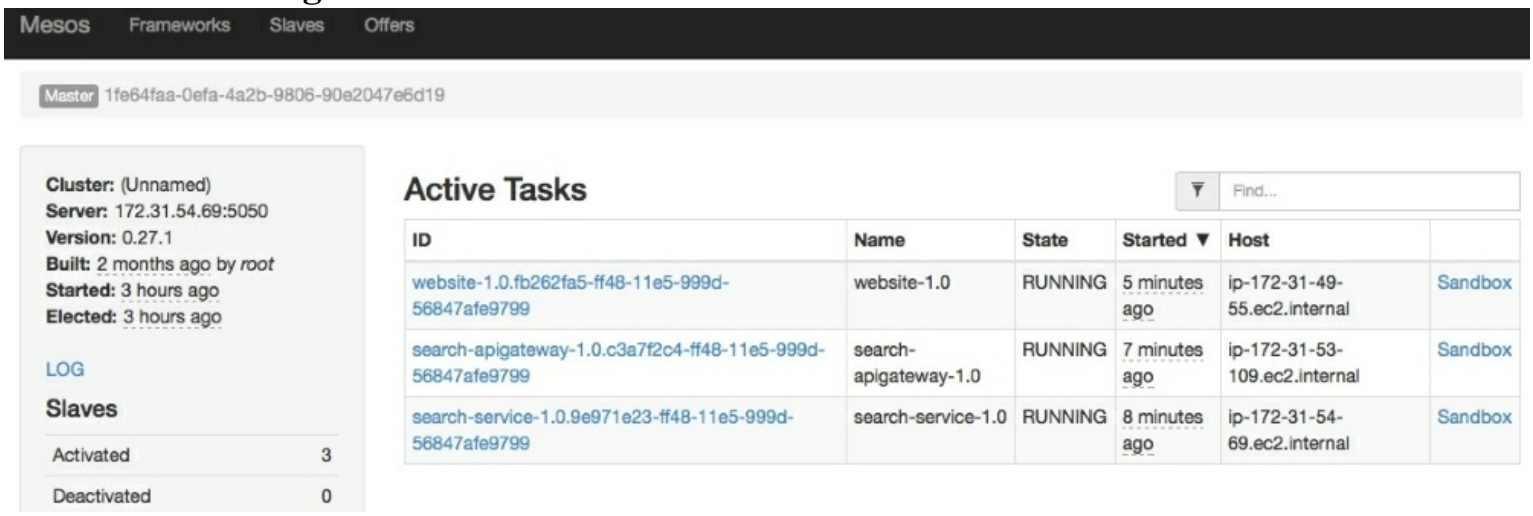
Reviewing the deployment

The steps for this are as follows:

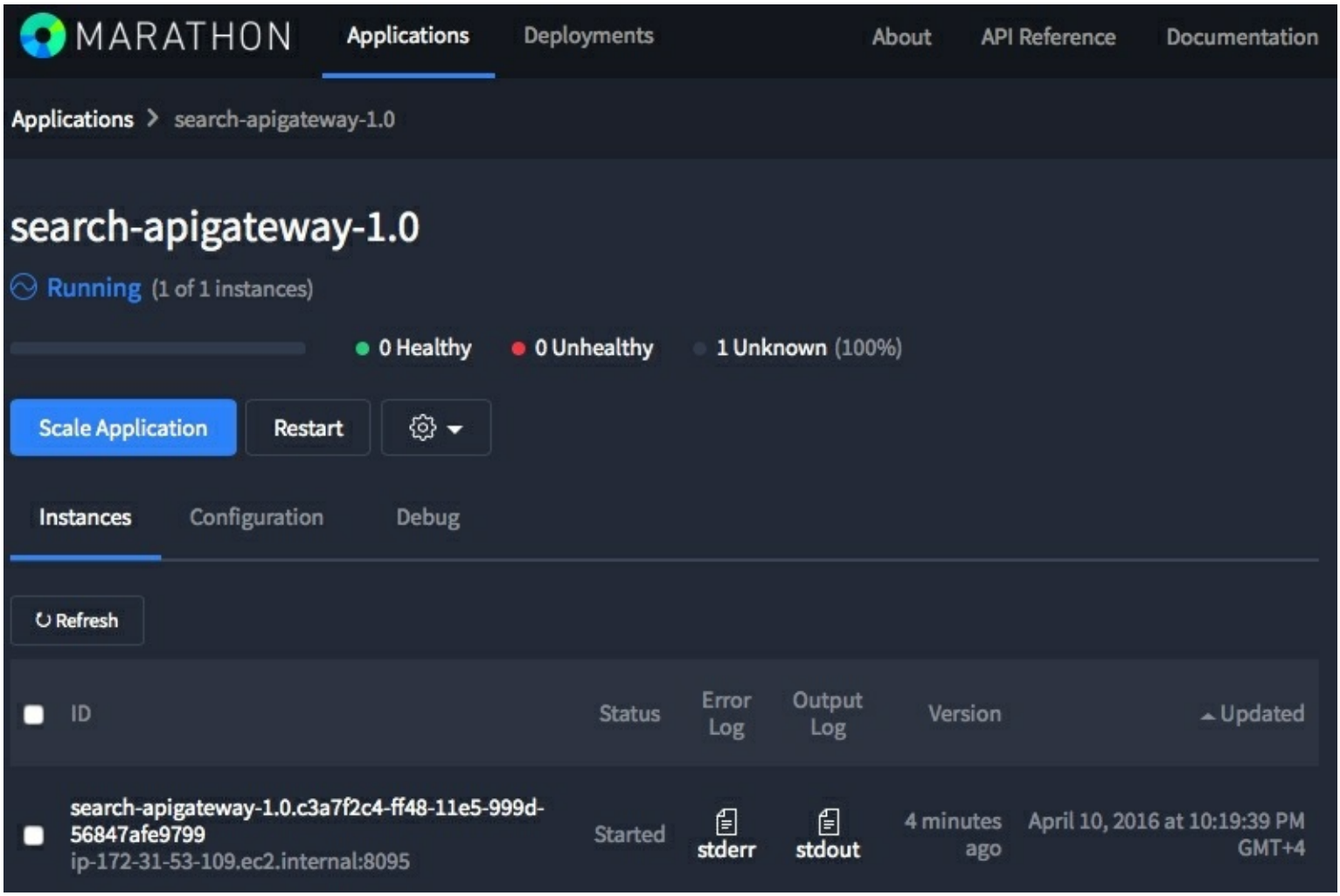
1. Open the Marathon UI. As shown in the following screenshot, the UI shows that all the three applications are deployed and are in the **Running** state. It also indicates that **1 of 1** instance is in the **Running** state:



2. Visit the Mesos UI. As shown in the following screenshot, there are three **Active Tasks**, all of them in the **Running** state. It also shows the host in which these services run:

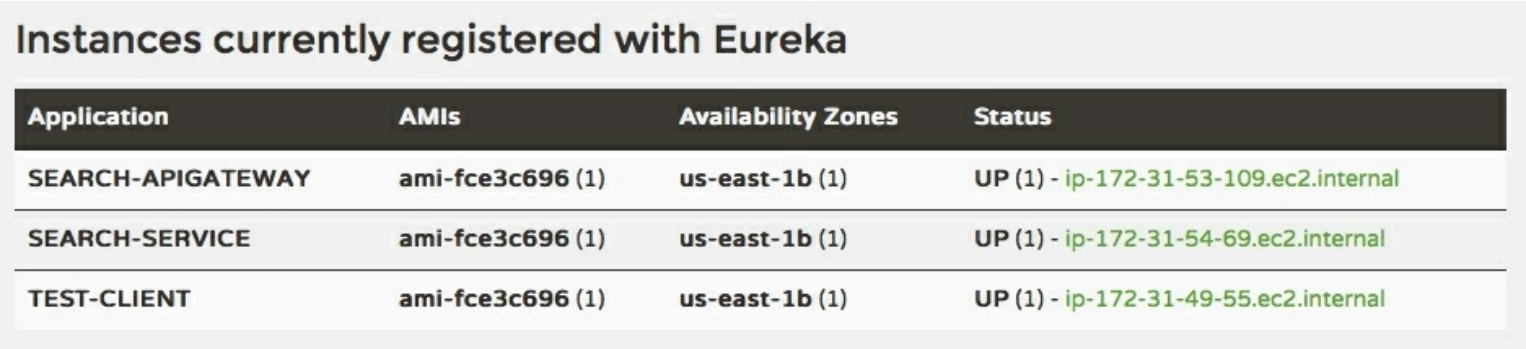


3. In the Marathon UI, click on a running application. The following screenshot shows the **search-apigateway-1.0** application. In the **Instances** tab, the IP address and port in which the service is bound is indicated:



The **Scale Application** button allows administrators to specify how many instances of the service are required. This can be used to scale up as well as scale down instances.

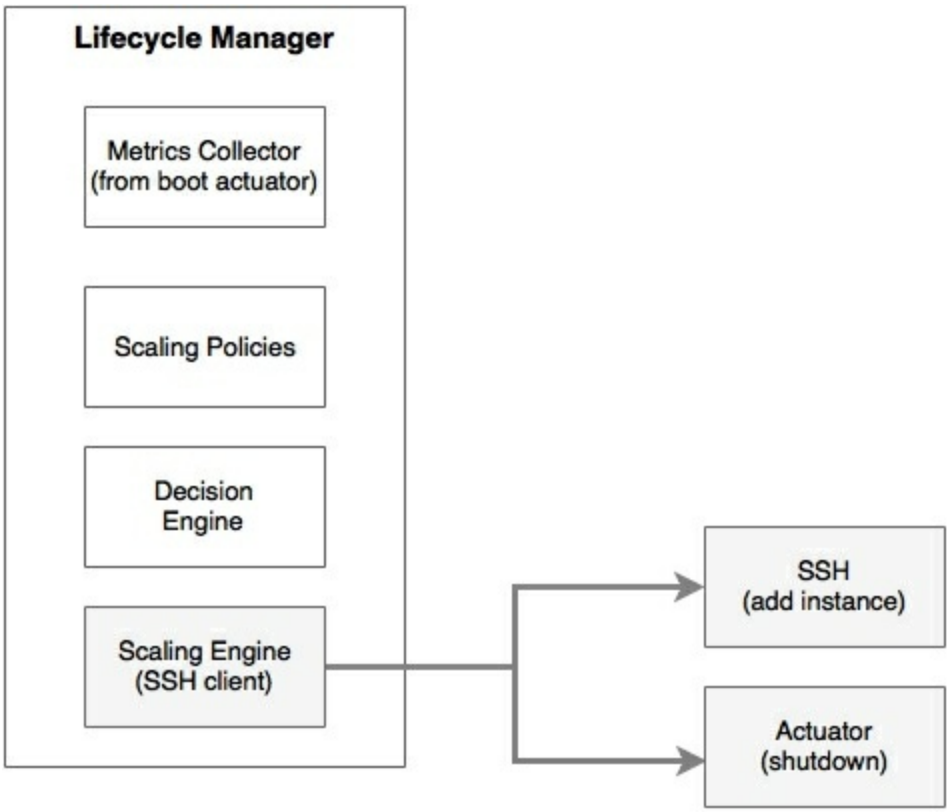
4. Open the Eureka server console to take a look at how the services are bound. As shown in the screenshot, **AMIs** and **Availability Zones** are reflected when services are registered. Follow `http://52.205.251.150:8761`:



5. Open `http://54.172.213.51:8001` in a browser to verify the **Website** application.

A place for the life cycle manager

The life cycle manager introduced in Chapter 6, *Autoscaling Microservices*, has the capability of autoscaling up or down instances based on demand. It also has the ability to take decisions on where to deploy and how to deploy applications on a cluster of machines based on polices and constraints. The life cycle manager's capabilities are shown in the following figure:



Marathon has the capability to manage clusters and deployments to clusters based on policies and constraints. The number of instances can be altered using the Marathon UI.

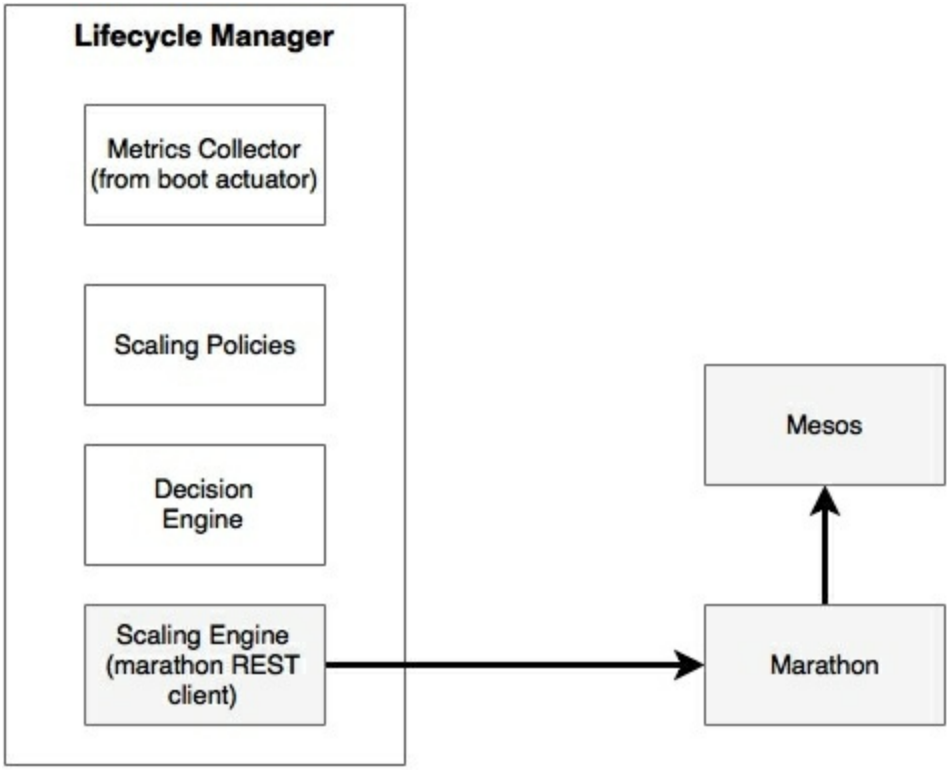
There are redundant capabilities between our life cycle manager and Marathon. With Marathon in place, SSH work or machine-level scripting is no longer required. Moreover, deployment policies and constraints can be delegated to Marathon. The REST APIs exposed by Marathon can be used to initiate scaling functions.

Marathon autoscale is a proof-of-concept project from Mesosphere for autoscaling. The Marathon autoscale provides basic autoscale features such as the CPU, memory, and rate of request.

Rewriting the life cycle manager with Mesos and Marathon

We still need a custom life cycle manager to collect metrics from the Spring Boot actuator endpoints. A custom life cycle manager is also handy if the scaling rules are beyond the CPU, memory, and rate of scaling.

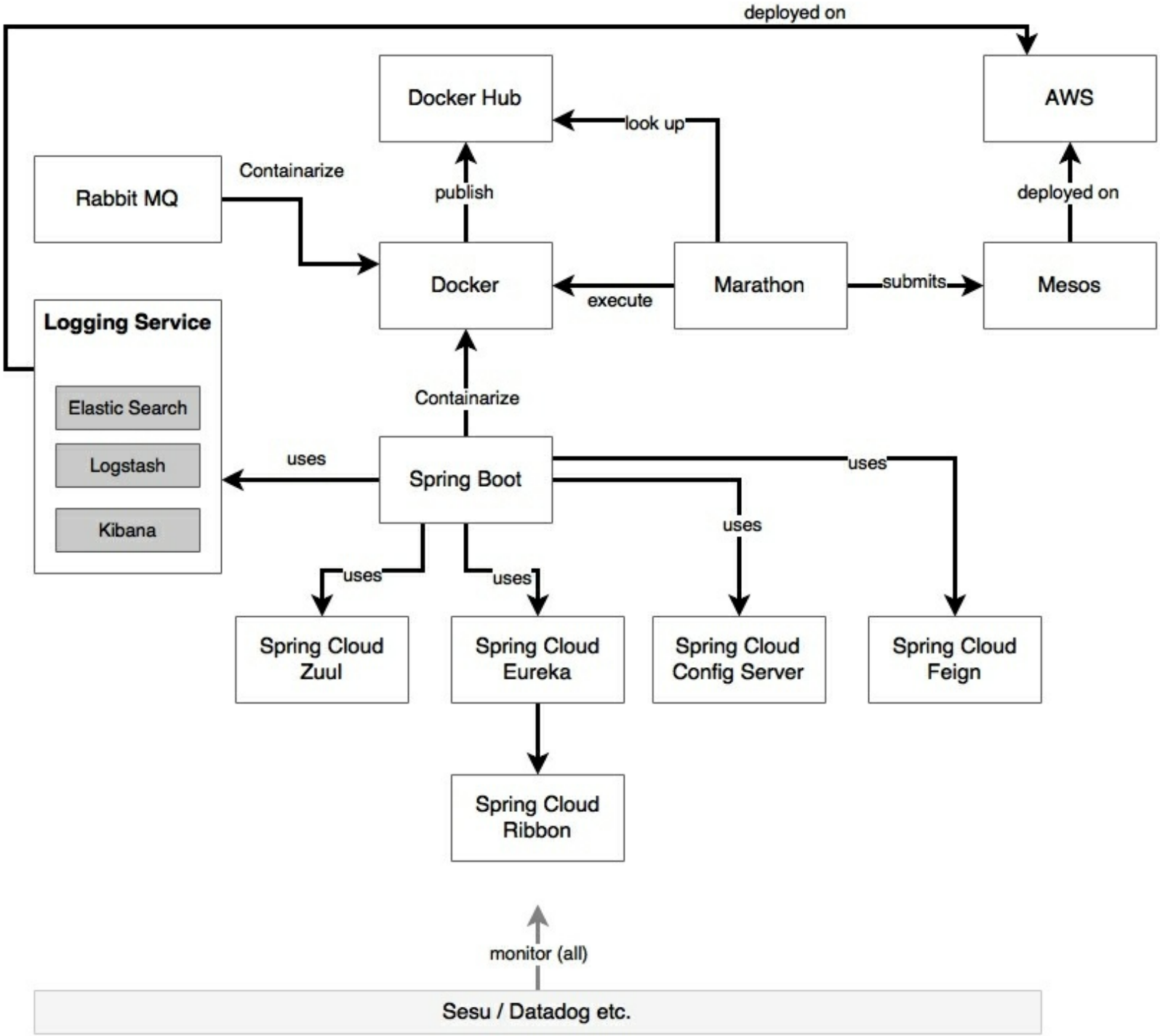
The following diagram shows the updated life cycle manager using the Marathon framework:



The life cycle manager, in this case, collects actuator metrics from different Spring Boot applications, combines them with other metrics, and checks for certain thresholds. Based on the scaling policies, the decision engine informs the scaling engine to either scale down or scale up. In this case, the scaling engine is nothing but a Marathon REST client. This approach is cleaner and neater than our earlier primitive life cycle manager implementation using SSH and Unix scripts.

The technology metamodel

We have covered a lot of ground on microservices with the BrownField PSS microservices. The following diagram sums it up by bringing together all the technologies used into a technology metamodel:



Summary

In this chapter, you learned the importance of a cluster management and init system to efficiently manage dockerized microservices at scale.

We explored the different cluster control or cluster orchestration tools before diving deep into Mesos and Marathon. We also implemented Mesos and Marathon in the AWS cloud environment to demonstrate how to manage dockerized microservices developed for BrownField PSS.

At the end of this chapter, we also explored the position of the life cycle manager in conjunction with Mesos and Marathon. Finally, we concluded this chapter with a technology metamodel based on the BrownField PSS microservices implementation.

So far, we have discussed all the core and supporting technology capabilities required for a successful microservices implementation. A successful microservice implementation also requires processes and practices beyond technology. The next chapter, the last in the book, will cover the process and practice perspectives of microservices.

Chapter 10. The Microservices Development Life Cycle

Similar to the **software development life cycle** (**SDLC**), it is important to understand the aspects of the microservice development life cycle processes for a successful implementation of the microservices architecture.

This final chapter will focus on the development process and practice of microservices with the help of BrownField Airline's PSS microservices example. Furthermore, this chapter will describe best practices in structuring development teams, development methodologies, automated testing, and continuous delivery of microservices in line with DevOps practices. Finally, this chapter will conclude by shedding light on the importance of the reference architecture in a decentralized governance approach to microservices.

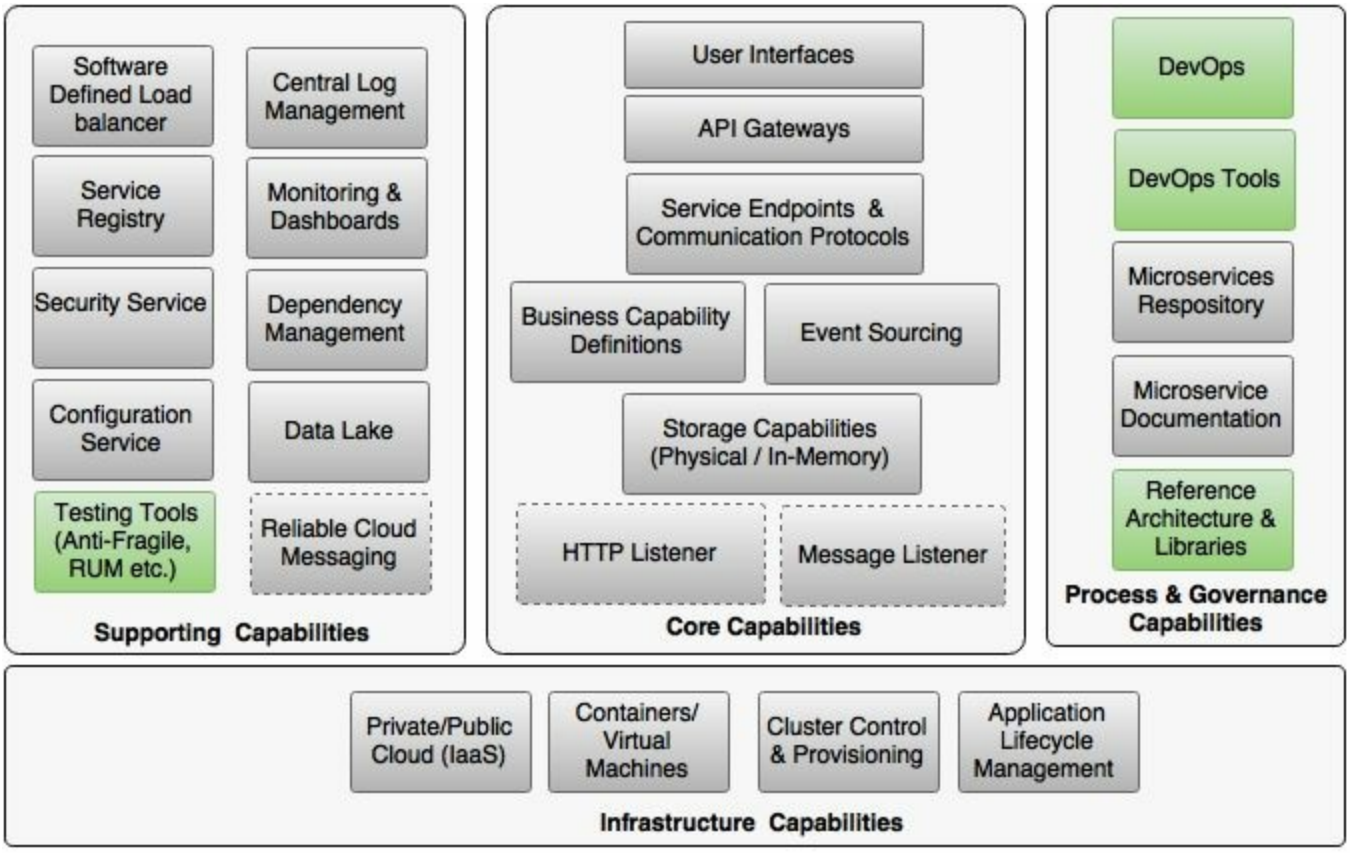
By the end of this chapter, you will learn about the following topics:

- Reviewing DevOps in the context of microservices development
- Defining the microservices life cycle and related processes
- Best practices around the development, testing, and deployment of Internet-scale microservices

Reviewing the microservice capability model

This chapter will cover the following microservices capabilities from the microservices capability model discussed in Chapter 3, *Applying Microservices Concepts*:

- **DevOps**
- **DevOps Tools**
- **Reference Architecture & Libraries**
- **Testing Tools (Anti-Fragile, RUM etc)**



The new mantra of lean IT – DevOps

We discussed the definition of DevOps in Chapter 2, *Building Microservices with Spring Boot*. Here is a quick recap of the DevOps definition.

Gartner defines DevOps as follows:

> *"DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology — especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective."*

DevOps and microservices evolved independently. Chapter 1, *Demystifying Microservices*, explored the evolution of microservices. In this section, we will review the evolution of DevOps and then take a look at how DevOps supports microservices adoption.

In the era of digital disruption and in order to support modern business, IT organizations have to master two key areas: speed of delivery and value-driven delivery. This is obviously apart from being expert in leading technologies.

Many IT organizations failed to master this change, causing frustration to business users. To overcome this situation, many business departments started their own shadow IT or stealth IT under their control. Some smart IT organizations then adopted a lean IT model to respond to these situations.

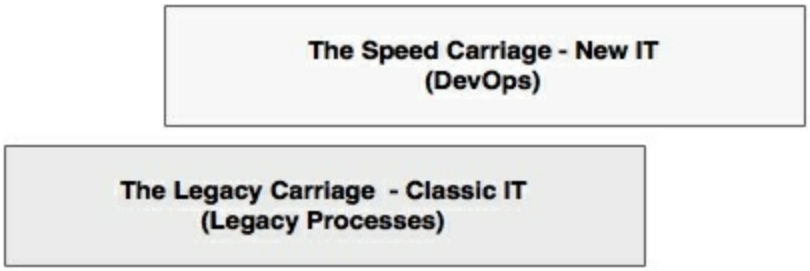
However, many organizations still struggle with this transformation due to the large baggage of legacy systems and processes. Gartner coined the concept of a **pace-layered application strategy**. Gartner's view is that high speed is required only for certain types of applications or certain business areas. Gartner termed this a **system of innovation**. A system of innovation requires rapid innovations compared to a **system of records**. As a system of innovations needs rapid innovation, a lean IT delivery model is essential for such applications. Practitioners evangelized the lean IT model as DevOps.

There are two key strategies used by organizations to adopt DevOps.

Some organizations positioned DevOps as a process to fill the gaps in their existing processes. Such organizations adopted an incremental strategy for their DevOps journey. The adoption path starts with Agile development, then incrementally adopts continuous integration, automated testing, and release to production and then all DevOps practices. The challenge in such organizations is the time to realize the full benefits as well as the mixed culture of people due to legacy processes.

Many organizations, therefore, take a disruptive approach to adopt DevOps. This will be achieved by partitioning IT into two layers or even as two different IT units. The high-speed layer of IT uses

DevOps-style practices to dramatically change the culture of the organization with no connection to the legacy processes and practices. A selective application cluster will be identified and moved to the new IT based on the business value:



The intention of DevOps is not just to reduce cost. It also enables the business to disrupt competitors by quickly moving ideas to production. DevOps attacks traditional IT issues in multiple ways, as explained here.

Reducing wastage

DevOps processes and practices essentially speed up deliveries which improves quality. The speed of delivery is achieved by cutting IT wastage. This is achieved by avoiding work that adds no value to the business nor to desired business outcomes. IT wastage includes software defects, productivity issues, process overheads, time lag in decision making, time spent in reporting layers, internal governance, overestimation, and so on. By reducing these wastages, organizations can radically improve the speed of delivery. The wastage is reduced by primarily adopting Agile processes, tools, and techniques.

Automating every possible step

By automating the manually executed tasks, one can dramatically improve the speed of delivery as well as the quality of deliverables. The scope of automation goes from planning to customer feedback. Automation reduces the time to move business ideas to production. This also reduces a number of manual gate checks, bureaucratic decision making, and so on. Automated monitoring mechanisms and feedback go back to the development factory, which gets it fixed and quickly moved to production.

Value-driven delivery

DevOps reduces the gap between IT and business through value-driven delivery. Value-driven delivery closely aligns IT to business by understanding true business values and helps the business by quickly delivering these values, which can give a competitive advantage. This is similar to the shadow IT concept, in which IT is collocated with the business and delivers business needs quickly, rather than waiting for heavy project investment-delivery cycles.

Traditionally, IT is partially disconnected from the business and works with IT KPIs, such as the number of successful project deliveries, whereas in the new model, IT shares business KPIs. As an example, a new IT KPI could be that IT helped business to achieve a 10% increase in sales orders or led to 20% increase in customer acquisition. This will shift IT's organizational position from merely a support organization to a business partner.

Bridging development and operations

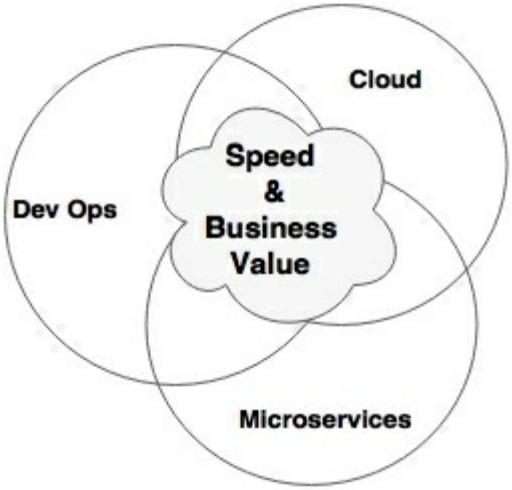
Traditionally, IT has different teams for development and operations. In many cases, they are differentiated with logical barriers. DevOps reduces the gap between the development and operations teams so that it can potentially reduce wastage and improve quality. Multidisciplinary teams work together to address problems at hand rather than throwing mud across the wall.

With DevOps, operations teams will have a fairly good understanding about the services and applications developed by development teams. Similarly, development teams will have a good handle on the infrastructure components and configurations used by the applications. As a result, operations teams can make decisions based exactly on service behaviors rather than enforcing standard organizational policies and rules when designing infrastructure components. This would eventually help the IT organization to improve the quality of the product as well as the time to resolve incidents and problem management.

In the DevOps world, speed of delivery is achieved through the automation of high-velocity changes, and quality is achieved through automation and people. Business values are achieved through efficiency, speed of delivery, quality, and the ability to innovate. Cost reduction is achieved through automation, productivity, and reducing wastage.

Meeting the trio – microservices, DevOps, and cloud

The trio—cloud, microservices, and DevOps—targets a set of common objectives: speed of delivery, business value, and cost benefit. All three can stay and evolve independently, but they complement each other to achieve the desired common goals. Organizations embarking on any of these naturally tend to consider the other two as they are closely linked together:



Many organizations start their journey with DevOps as an organizational practice to achieve high-velocity release cycles but eventually move to the microservices architecture and cloud. It is not mandatory to have microservices and cloud support DevOps. However, automating the release cycles of large monolithic applications does not make much sense, and in many cases, it would be impossible to achieve. In such scenarios, the microservices architecture and cloud will be handy when implementing DevOps.

If we flip a coin, cloud does not need a microservices architecture to achieve its benefits. However, to effectively implement microservices, both cloud and DevOps are essential.

In summary, if the objective of an organization is to achieve a high speed of delivery and quality in a cost-effective way, the trio together can bring tremendous success.

Cloud as the self-service infrastructure for Microservices

The main driver for cloud is to improve agility and reduce cost. By reducing the time to provision the infrastructure, the speed of delivery can be increased. By optimally utilizing the infrastructure, one can bring down the cost. Therefore, cloud directly helps achieve both speed of delivery and cost.

As discussed in Chapter 9, *Managing Dockerized Microservices with Mesos and Marathon*, without having a cloud infrastructure with cluster management software, it would be hard to control the infrastructure cost when deploying microservices. Hence, the cloud with self-service capabilities is essential for microservices to achieve their full potential benefits. In the microservices context, the cloud not only helps abstract the physical infrastructure but also provides software APIs for dynamic provisioning and automatic deployments. This is referred to as **infrastructure as code** or **software-defined infrastructure**.

DevOps as the practice and process for microservices

Microservice is an architecture style that enables quick delivery. However, microservices cannot provide the desired benefits by themselves. A microservices-based project with a delivery cycle of 6 months does not give the targeted speed of delivery or business agility. Microservices need a set of supporting delivery practices and processes to effectively achieve their goal.

DevOps is the ideal candidate for the underpinning process and practices for microservice delivery. DevOps processes and practices gel well with the microservices architecture's philosophies.

Practice points for microservices development

For a successful microservice delivery, a number of development-to-delivery practices need to be considered, including the DevOps philosophy. In the previous chapters, you learned the different architecture capabilities of microservices. In this section, we will explore the nonarchitectural aspects of microservice developments.

Understanding business motivation and value

Microservices should not be used for the sake of implementing a niche architecture style. It is extremely important to understand the business value and business KPIs before selecting microservices as an architectural solution for a given problem. A good understanding of business motivation and business value will help engineers focus on achieving these goals in a cost-effective way.

Business motivation and value should justify the selection of microservices. Also, using microservices, the business value should be realizable from a business point of view. This will avoid situations where IT invests in microservices but there is no appetite from the business to leverage any of the benefits that microservices can bring to the table. In such cases, a microservices-based development would be an overhead to the enterprise.

Changing the mindset from project to product development

As discussed in Chapter 1, *Demystifying Microservices*, microservices are more aligned to product development. Business capabilities that are delivered using microservices should be treated as products. This is in line with the DevOps philosophy as well.

The mindset for project development and product development is different. The product team will always have a sense of ownership and take responsibility for what they produce. As a result, product teams always try to improve the quality of the product. The product team is responsible not only for delivering the software but also for production support and maintenance of the product.

Product teams are generally linked directly to a business department for which they are developing the product. In general, product teams have both an IT and a business representative. As a result, product thinking is closely aligned with actual business goals. At every moment, product teams understand the value they are adding to the business to achieve business goals. The success of the product directly lies with the business value being gained out of the product.

Because of the high-velocity release cycles, product teams always get a sense of satisfaction in their delivery, and they always try to improve on it. This brings a lot more positive dynamics within the team.

In many cases, typical product teams are funded for the long term and remain intact. As a result, product teams become more cohesive in nature. As they are small in size, such teams focus on improving their process from their day-to-day learnings.

One common pitfall in product development is that IT people represent the business in the product team. These IT representatives may not fully understand the business vision. Also, they may not be empowered to take decisions on behalf of the business. Such cases can result in a misalignment with the business and lead to failure quite rapidly.

It is also important to consider a collocation of teams where business and IT representatives reside at the same place. Collocation adds more binding between IT and business teams and reduces communication overheads.

Choosing a development philosophy

Different organizations take different approaches to developing microservices, be it a migration or a new development. It is important to choose an approach that suits the organization. There is a wide verity of approaches available, out of which a few are explained in this section.

Design thinking

Design thinking is an approach primarily used for innovation-centric development. It is an approach that explores the system from an end user point of view: what the customers see and how they experience the solution. A story is then built based on observations, patterns, intuition, and interviews.

Design thinking then quickly devises solutions through solution-focused thinking by employing a number of theories, logical reasoning, and assumptions around the problem. The concepts are expanded through brainstorming before arriving at a converged solution.

Once the solution is identified, a quick prototype is built to consider how the customer responds to it, and then the solution is adjusted accordingly. When the team gets satisfactory results, the next step is taken to scale the product. Note that the prototype may or may not be in the form of code.

Design thinking uses human-centric thinking with feelings, empathy, intuition, and imagination at its core. In this approach, solutions will be up for rethinking even for known problems to find innovative and better solutions.

The start-up model

More and more organizations are following the start-up philosophy to deliver solutions. Organizations create internal start-up teams with the mission to deliver specific solutions. Such teams stay away from day-to-day organizational activities and focus on delivering their mission.

Many start-ups kick off with a small, focused team—a highly cohesive unit. The unit is not worried about how they achieve things; rather, the focus is on what they want to achieve. Once they have a product in place, the team thinks about the right way to build and scale it.

This approach addresses quick delivery through production-first thinking. The advantage with this approach is that teams are not disturbed by organizational governance and political challenges. The team is empowered to think out of the box, be innovative, and deliver things. Generally, a higher level of ownership is seen in such teams, which is one of the key catalysts for success. Such teams employ just enough processes and disciplines to take the solution forward. They also follow a fail fast approach and course correct sooner than later.

The Agile practice

The most commonly used approach is the Agile methodology for development. In this approach,

software is delivered in an incremental, iterative way using the principles put forth in the Agile manifesto. This type of development uses an Agile method such as Scrum or XP. The Agile manifesto defines four key points that Agile software development teams should focus on:

- Individuals and interaction over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Note

The 12 principles of Agile software development can be found at http://www.agilemanifesto.org/principles.html.

Using the concept of Minimum Viable Product

Irrespective of the development philosophy explained earlier, it is essential to identify a **Minimum Viable Product** (**MVP**) when developing microservice systems for speed and agility.

Eric Ries, while pioneering the lean start-up movement, defined MVP as:

> *"A Minimum Viable Product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort."*

The objective of the MVP approach is to quickly build a piece of software that showcases the most important aspects of the software. The MVP approach realizes the core concept of an idea and perhaps chooses those features that add maximum value to the business. It helps get early feedback and then course corrects as necessary before building a heavy product.

The MVP may be a full-fledged service addressing limited user groups or partial services addressing wider user groups. Feedback from customers is extremely important in the MVP approach. Therefore, it is important to release the MVP to the real users.

Overcoming the legacy hotspot

It is important to understand the environmental and political challenges in an organization before embarking on microservices development.

It is common in microservices to have dependencies on other legacy applications, directly or indirectly. A common issue with direct legacy integration is the slow development cycle of the legacy application. An example would be an innovative railway reservation system relaying on an age-old **transaction processing facility** (**TPF**) for some of the core backend features, such as reservation. This is especially common when migrating legacy monolithic applications to microservices. In many cases, legacy systems continue to undergo development in a non-Agile way with larger release cycles. In such cases, microservices development teams may not be able to move so quickly because of the coupling with legacy systems. Integration points might drag the microservices developments heavily. Organizational political challenges make things even worse.

There is no silver bullet to solve this issue. The cultural and process differences could be an ongoing issue. Many enterprises ring-fence such legacy systems with focused attention and investments to support fast-moving microservices. Targeted C-level interventions on these legacy platforms could reduce the overheads.

Addressing challenges around databases

Automation is key in microservices development. Automating databases is one of the key challenges in many microservice developments.

In many organizations, DBAs play a critical role in database management, and they like to treat the databases under their control differently. Confidentiality and access control on data is also cited as a reason for DBAs to centrally manage all data.

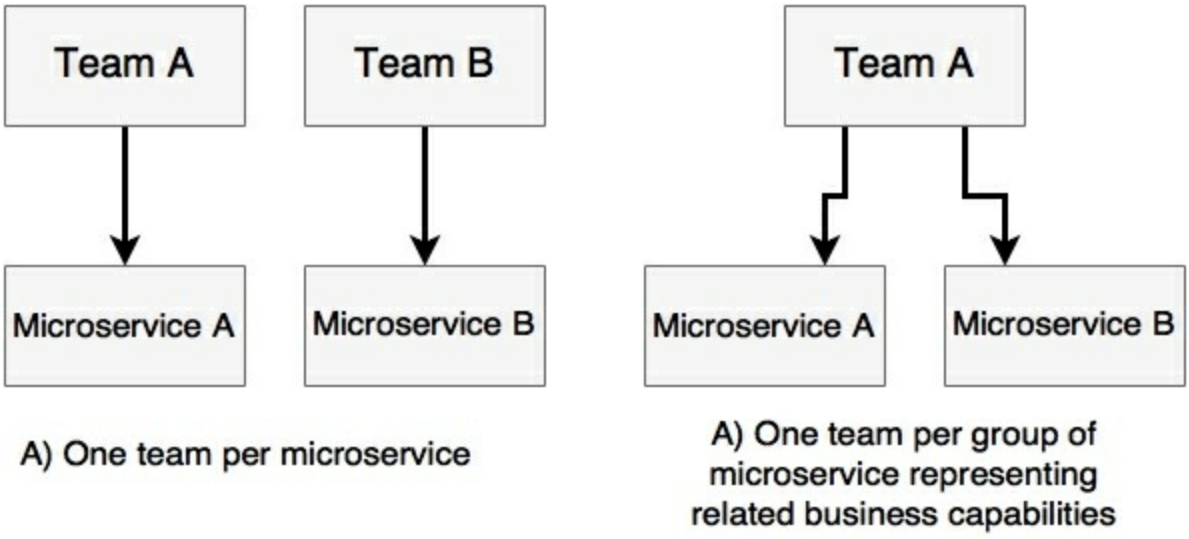
Many automation tools focus on the application logic. As a result, many development teams completely ignore database automation. Ignoring database automation can severely impact the overall benefits and can derail microservices development.

In order to avoid such situations, the database has to be treated in the same way as applications with appropriate source controls and change management. When selecting a database, it is also important to consider automation as one of the key aspects.

Database automation is much easier in the case of NoSQL databases but is hard to manage with traditional RDBMs. **Database Lifecycle Management** (**DLM**) as a concept is popular in the DevOps world, particularly to handle database automation. Tools such as DBmaestro, Redgate DLM, Datical DB, and Delphix support database automation.

Establishing self-organizing teams

One of the most important activities in microservices development is to establish the right teams for development. As recommended in many DevOps processes, a small, focused team always delivers the best results.



As microservices are aligned with business capabilities and are fairly loosely coupled products, it is ideal to have a dedicated team per microservice. There could be cases where the same team owns multiple microservices from the same business area representing related capabilities. These are generally decided by the coupling and size of the microservices.

Team size is an important aspect in setting up effective teams for microservices development. The general notion is that the team size should not exceed 10 people. The recommended size for optimal delivery is between 4 and 7. The founder of Amazon.com, Jeff Bezos, coined the theory of two-pizza teams. Jeff's theory says the team will face communication issues if the size gets bigger. Larger teams work with consensus, which results in increased wastage. Large teams also lose ownership and accountability. A yardstick is that the product owner should get enough time to speak to individuals in the team to make them understand the value of what they are delivering.

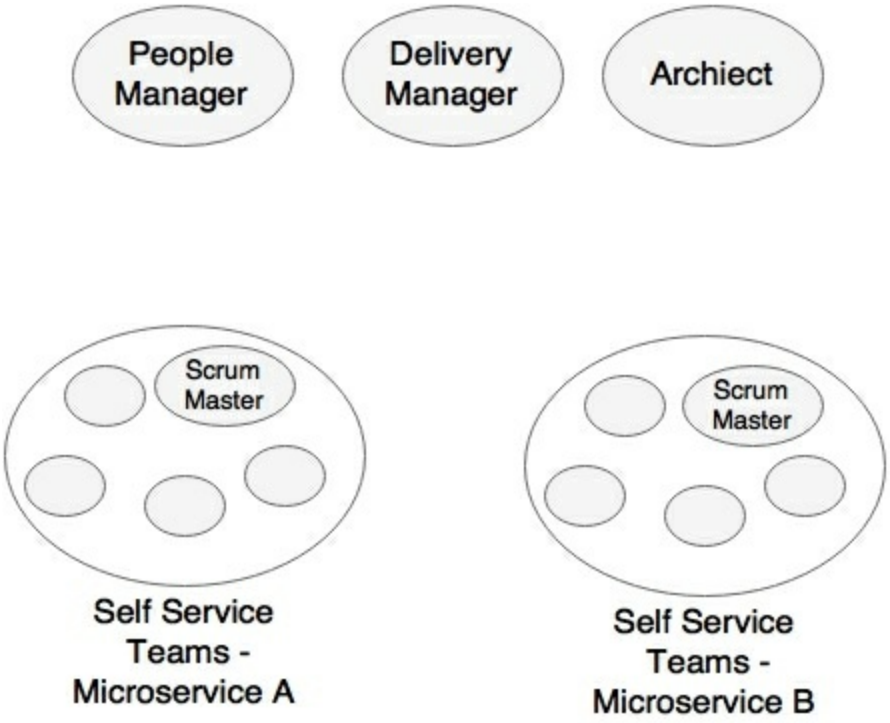
Teams are expected to take full ownership in ideating for, analyzing, developing, and supporting services. Werner Vogels from Amazon.com calls this *you build it and you run it*. As per Werner's theory, developers pay more attention to develop quality code to avoid unexpected support calls. The members in the team consist of fullstack developers and operational engineers. Such a team is fully aware of all the areas. Developers understand operations as well as operations teams understand applications. This not only reduces the changes of throwing mud across teams but also improves quality.

Teams should have multidisciplinary skills to satisfy all the capabilities required to deliver a service. Ideally, the team should not rely on external teams to deliver the components of the service. Instead,

the team should be self-sufficient. However, in most organizations, the challenge is on specialized skills that are rare. For example, there may not be many experts on a graph database in the organization. One common solution to this problem is to use the concept of consultants. Consultants are SMEs and are engaged to gain expertise on specific problems faced by the team. Some organizations also use shared or platform teams to deliver some common capabilities.

Team members should have a complete understanding of the products, not only from the technical standpoint but also from the business case and the business KPIs. The team should have collective ownership in delivering the product as well as in achieving business goals together.

Agile software development also encourages having self-organizing teams. Self-organizing teams act as a cohesive unit and find ways to achieve their goals as a team. The team automatically align themselves and distribute the responsibilities. The members in the team are self-managed and empowered to make decisions in their day-to-day work. The team's communication and transparency are extremely important in such teams. This emphasizes the need for collocation and collaboration, with a high bandwidth for communication:



In the preceding diagram, both **Microservice A** and **Microservice B** represent related business capabilities. Self-organizing teams treat everyone in the team equally, without too many hierarchies and management overheads within the team. The management would be thin in such cases. There won't be many designated vertical skills in the team, such as team lead, UX manager, development manager, testing manager, and so on. In a typical microservice development, a shared product manager, shared architect, and a shared people manager are good enough to manage the different microservice teams. In some organizations, architects also take up responsibility for delivery.

Self-organizing teams have some level of autonomy and are empowered to take decisions in a quick

and Agile mode rather than having to wait for long-running bureaucratic decision-making processes that exist in many enterprises. In many of these cases, enterprise architecture and security are seen as an afterthought. However, it is important to have them on board from the beginning. While empowering the teams with maximum freedom for developers in decision-making capabilities, it is equally important to have fully automated QA and compliance so as to ensure that deviations are captured at the earliest.

Communication between teams is important. However, in an ideal world, it should be limited to interfaces between microservices. Integrations between teams ideally has to be handled through consumer-driven contracts in the form of test scripts rather than having large interface documents describing various scenarios. Teams should use mock service implementations when the services are not available.

Building a self-service cloud

One of the key aspects that one should consider before embarking on microservices is to build a cloud environment. When there are only a few services, it is easy to manage them by manually assigning them to a certain predesignated set of virtual machines.

However, what microservice developers need is more than just an IaaS cloud platform. Neither the developers nor the operations engineers in the team should worry about where the application is deployed and how optimally it is deployed. They also should not worry about how the capacity is managed.

This level of sophistication requires a cloud platform with self-service capabilities, such as what we discussed in Chapter 9, *Managing Dockerized Microservices with Mesos and Marathon*, with the Mesos and Marathon cluster management solutions. Containerized deployment discussed in Chapter 8, *Containerizing Microservices with Docker*, is also important in managing and end to-end-automation. Building this self-service cloud ecosystem is a prerequisite for microservice development.

Building a microservices ecosystem

As we discussed in the capability model in Chapter 3, *Applying Microservices Concepts*, microservices require a number of other capabilities. All these capabilities should be in place before implementing microservices at scale.

These capabilities include service registration, discovery, API gateways, and an externalized configuration service. All are provided by the Spring Cloud project. Capabilities such as centralized logging, monitoring, and so on are also required as a prerequisite for microservices development.

Defining a DevOps-style microservice life cycle process

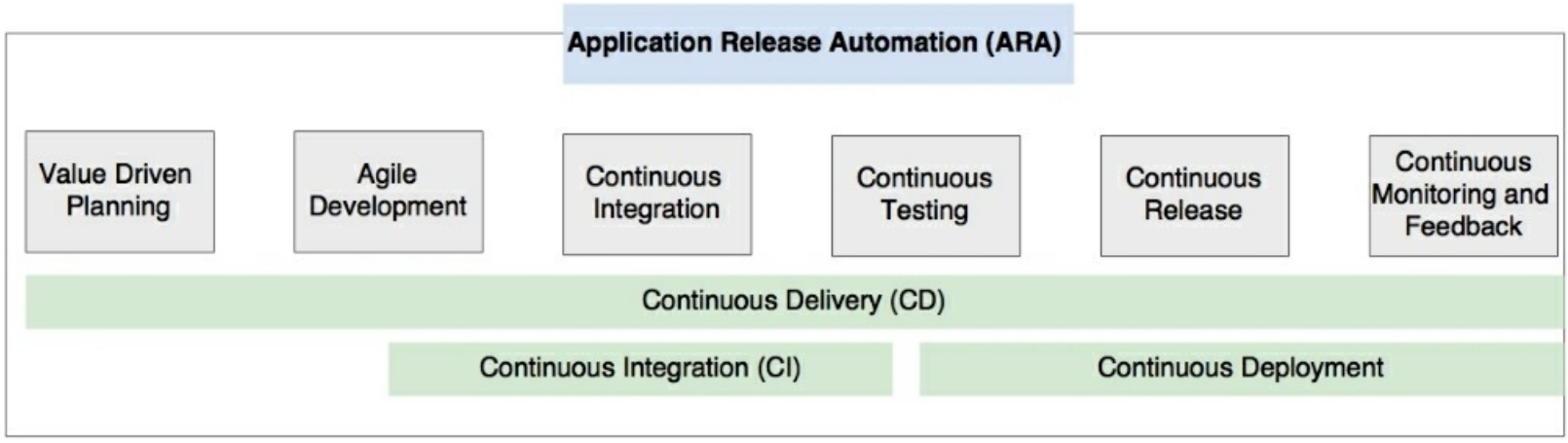
DevOps is the best-suited practice for microservices development. Organizations already practicing DevOps do not need another practice for microservices development.

In this section, we will explore the life cycle of microservices development. Rather than reinventing a process for microservices, we will explore DevOps processes and practices from the microservice perspective.

Before we explore DevOps processes, let's iron out some of the common terminologies used in the DevOps world:

- **Continuous integration** (**CI**): This automates the application build and quality checks continuously in a designated environment, either in a time-triggered manner or on developer commits. CI also publishes code metrics to a central dashboard as well as binary artifacts to a central repository. CI is popular in Agile development practices.
- **Continuous delivery** (**CD**): This automates the end-to-end software delivery practice from idea to production. In a non-DevOps model, this used to be known as **Application Lifecycle Management** (**ALM**). One of the common interpretations of CD is that it is the next evolution of CI, which adds QA cycles into the integration pipeline and makes the software ready to release to production. A manual action is required to move it to production.
- **Continuous deployment**: This is an approach to automating the deployment of application binaries to one or more environments by managing binary movement and associated configuration parameters. Continuous deployment is also considered as the next evolution of CD by integrating automatic release processes into the CD pipeline.
- **Application Release Automation** (**ARA**): ARA tools help monitor and manage end-to-end delivery pipelines. ARA tools use CI and CD tools and manage the additional steps of release management approvals. ARA tools are also capable of rolling out releases to different environments and rolling them back in case of a failed deployment. ARA provides a fully orchestrated workflow pipeline, implementing delivery life cycles by integrating many specialized tools for repository management, quality assurance, deployment, and so on. XL Deploy and Automic are some of the ARA tools.

The following diagram shows the DevOps process for microservices development:



Let's now further explore these life cycle stages of microservices development.

Value-driven planning

Value-driven planning is a term used in Agile development practices. Value-driven planning is extremely important in microservices development. In value-driven planning, we will identify which microservices to develop. The most important aspect is to identify those requirements that have the highest value to business and those that have the lowest risks. The MVP philosophy is used when developing microservices from the ground up. In the case of monolithic to microservices migration, we will use the guidelines provided in Chapter 3, *Applying Microservices Concepts*, to identify which services have to be taken first. The selected microservices are expected to precisely deliver the expected value to the business. Business KPIs to measure this value have to be identified as part of value-driven planning.

Agile development

Once the microservices are identified, development must be carried out in an Agile approach following the Agile manifesto principles. The scrum methodology is used by most of the organizations for microservices development.

Continuous integration

The continuous integration steps should be in place to automatically build the source code produced by various team members and generate binaries. It is important to build only once and then move the binary across the subsequent phases. Continuous integration also executes various QAs as part of the build pipeline, such as code coverage, security checks, design guidelines, and unit test cases. CI typically delivers binary artefacts to a binary artefact repository and also deploys the binary artefacts into one or more environments. Part of the functional testing also happens as part of CI.

Continuous testing

Once continuous integration generates the binaries, they are moved to the testing phase. A fully automated testing cycle is kicked off in this phase. It is also important to automate security testing as

part of the testing phase. Automated testing helps improve the quality of deliverables. The testing may happen in multiple environments based on the type of testing. This could range from the integration test environment to the production environment to test in production.

Continuous release

Continuous release to production takes care of actual deployment, infrastructure provisioning, and rollout. The binaries are automatically shipped and deployed to production by applying certain rules. Many organizations stop automation with the staging environment and make use of manual approval steps to move to production.

Continuous monitoring and feedback

The continuous monitoring and feedback phase is the most important phase in Agile microservices development. In an MVP scenario, this phase gives feedback on the initial acceptance of the MVP and also evaluates the value of the service developed. In a feature addition scenario, this further gives insight into how this new feature is accepted by users. Based on the feedback, the services are adjusted and the same cycle is then repeated.

Automating the continuous delivery pipeline

In the previous section, we discussed the life cycle of microservices development. The life cycle stages can be altered by organizations based on their organizational needs but also based on the nature of the application. In this section, we will take a look at a sample continuous delivery pipeline as well as toolsets to implement a sample pipeline.

There are many tools available to build end-to-end pipelines, both in the open source and commercial space. Organizations can select the products of their choice to connect pipeline tasks.

Tip

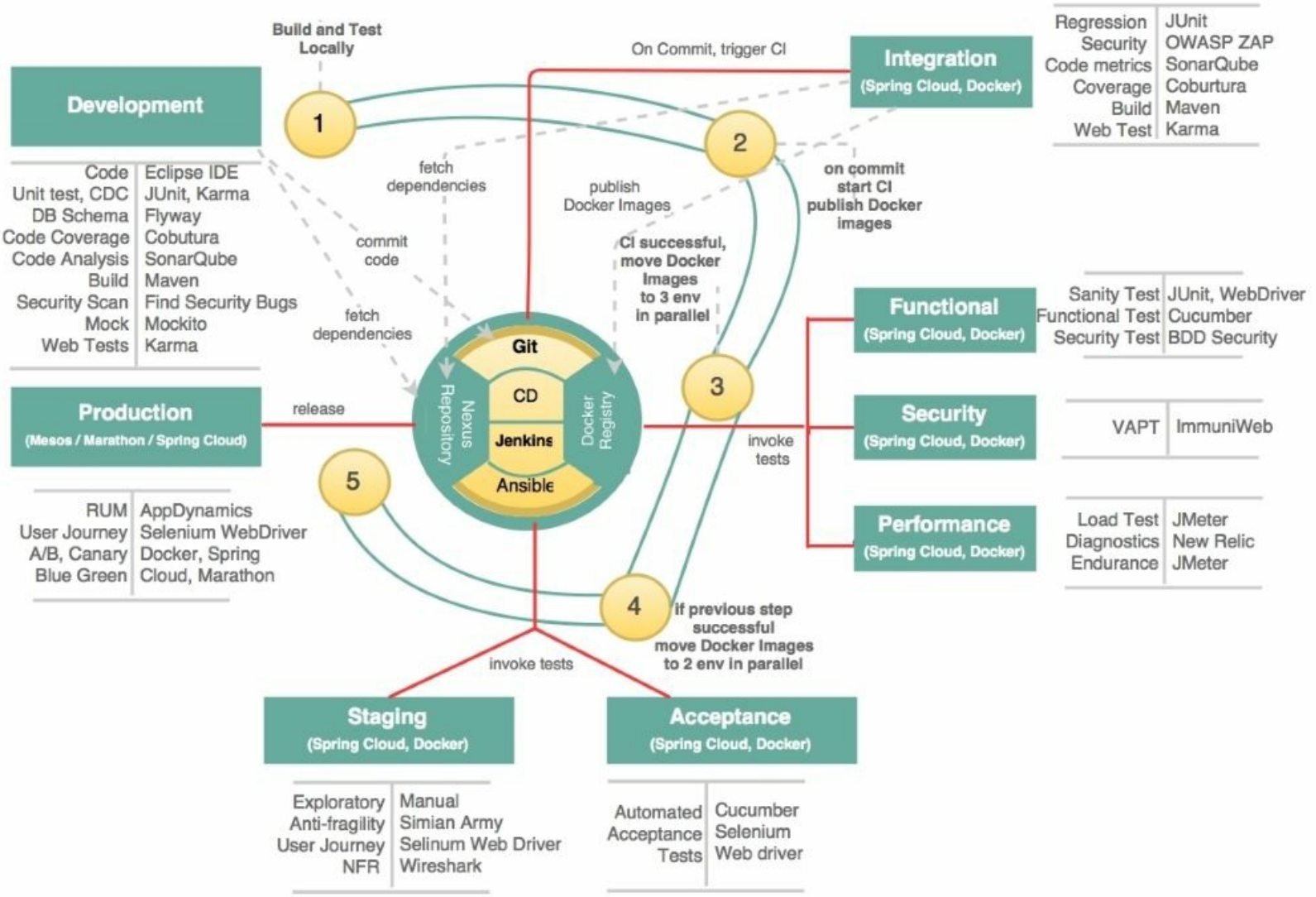
Refer to the XebiaLabs periodic table for a tool reference to build continuous delivery pipelines. It is available at https://xebialabs.com/periodic-table-of-devops-tools/.

The pipelines may initially be expensive to set up as they require many toolsets and environments. Organizations may not realize an immediate cost benefit in implementing the delivery pipeline. Also, building a pipeline needs high-power resources. Large build pipelines may involve hundreds of machines. It also takes hours to move changes through the pipeline from one end to the other. Hence, it is important to have different pipelines for different microservices. This will also help decoupling between the releases of different microservices.

Within a pipeline, parallelism should be employed to execute tests on different environments. It is also important to parallelize the execution of test cases as much as possible. Hence, designing the pipeline based on the nature of the application is important. There is no one size fits all scenario.

The key focus in the pipeline is on end-to-end automation, from development to production, and on failing fast if something goes wrong.

The following pipeline is an indicative one for microservices and explores the different capabilities that one should consider when developing a microservices pipeline:



The continuous delivery pipeline stages are explained in the following sections.

Development

The development stage has the following activities from a development perspective. This section also indicates some of the tools that can be used in the development stage. These tools are in addition to the planning, tracking, and communication tools such as Agile JIRA, Slack, and others used by Agile development teams. Take a look at the following:

- **Source code**: The development team requires an IDE or a development environment to cut source code. In most organizations, developers get the freedom to choose the IDEs they want. Having said this, the IDEs can be integrated with a number of tools to detect violations against guidelines. Generally, Eclipse IDEs have plugins for static code analysis and code matrices. SonarQube is one example that integrates other plugins such as Checkstyle for code conventions, PMD to detect bad practices, FindBugs to detect potential bugs, and Cobertura for code coverage. It is also recommended to use Eclipse plugins such as ESVD, Find Security Bugs, SonarQube Security Rules, and so on to detect security vulnerabilities.
- **Unit test cases**: The development team also produces unit test cases using JUnit, NUnit,

TestNG, and so on. Unit test cases are written against components, repositories, services, and so on. These unit test cases are integrated with the local Maven builds. The unit test cases targeting the microservice endpoints (service tests) serve as the regression test pack. Web UI, if written in AngularJS, can be tested using Karma.

- **Consumer-driven contracts**: Developers also write CDCs to test integration points with other microservices. Contract test cases are generally written as JUnit, NUnit, TestNG, and so on and are added to the service tests pack mentioned in the earlier steps.
- **Mock testing**: Developers also write mocks to simulate the integration endpoints to execute unit test cases. Mockito, PowerMock, and others are generally used for mock testing. It is good practice to deploy a mock service based on the contract as soon as the service contract is identified. This acts as a simple mechanism for service virtualization for the subsequent phases.
- **Behavior driven design** (**BDD**): The Agile team also writes BDD scenarios using a BDD tool, such as Cucumber. Typically, these scenarios are targeted against the microservices contract or the user interface that is exposed by a microservice-based web application. Cucumber with JUnit and Cucumber with Selenium WebDriver, respectively, are used in these scenarios. Different scenarios are used for functional testing, user journey testing, as well as acceptance testing.
- **Source code repository**: A source control repository is a part and parcel of development. Developers check-in their code to a central repository, mostly with the help of IDE plugins. One microservice per repository is a common pattern used by many organizations. This disallows other microservice developers from modifying other microservices or writing code based on the internal representations of other microservices. Git and Subversion are the popular choices to be used as source code repositories.
- **Build tools**: A build tool such as Maven or Gradle is used to manage dependencies and build target artifacts—in this case, Spring Boot services. There are many cases, such as basic quality checks, security checks and unit test cases, code coverage, and so on, that are integrated as part of the build itself. These are similar to the IDE, especially when IDEs are not used by developers. The tools that we examined as part of the IDEs are also available as Maven plugins. The development team does not use containers such as Docker until the CI phase of the project. All the artifacts have to be versioned properly for every change.
- **Artifact repository**: The artifact repository plays a pivotal role in the development process. The artifact repository is where all build artifacts are stored. The artifact repository could be Artifactory, Nexus, or any similar product.
- **Database schemas**: Liquibase and Flyway are commonly used to manage, track, and apply database changes. Maven plugins allow interaction with the Liquibase or Flyway libraries. The schema changes are versioned and maintained, just like source code.

Continuous integration

Once the code is committed to the repository, the next phase, continuous integration, automatically starts. This is done by configuring a CI pipeline. This phase builds the source code with a repository snapshot and generates deployable artifacts. Different organizations use different events to kickstart the build. A CI start event may be on every developer commit or may be based on a time window, such as daily, weekly, and so on.

The CI workflow is the key aspect of this phase. Continuous integration tools such as Jenkins, Bamboo, and others play the central role of orchestrating the build pipeline. The tool is configured with a workflow of activities to be invoked. The workflow automatically executes configured steps such as build, deploy, and QA. On the developer commit or on a set frequency, the CI kickstarts the workflow.

The following activities take place in a continuous integration workflow:

1. **Build and QA**: The workflow listens to Git webhooks for commits. Once it detects a change, the first activity is to download the source code from the repository. A build is executed on the downloaded snapshot source code. As part of the build, a number of QA checks are automatically performed, similarly to QA executed in the development environment. These include code quality checks, security checks, and code coverage. Many of the QAs are done with tools such as SonarQube, with the plugins mentioned earlier. It also collects code metrics such as code coverage and more and publishes it to a central database for analysis. Additional security checks are executed using OWASP ZAP Jenkins' plugins. As part of the build, it also executes JUnit or similar tools used to write test cases. If the web application supports Karma for UI testing, Jenkins is also capable of running web tests written in Karma. If the build or QA fails, it sends out alarms as configured in the system.
2. **Packaging**: Once build and QA are passed, the CI creates a deployable package. In our microservices case, it generates the Spring Boot standalone JAR. It is recommended to build Docker images as part of the integration build. This is the one and only place where we build binary artifacts. Once the build is complete, it pushes the immutable Docker images to a Docker registry. This could be on Docker Hub or a private Docker registry. It is important to properly version control the containers at this stage itself.
3. **Integration tests**: The Docker image is moved to the integration environment where regression tests (service tests) and the like are executed. This environment has other dependent microservices capabilities, such as Spring Cloud, logging, and so on, in place. All dependent microservices are also present in this environment. If an actual dependent service is not yet deployed, service virtualization tools such as MockServer are used. Alternately, a base version of the service is pushed to Git by the respective development teams. Once successfully deployed, Jenkins triggers service tests (JUnits against services), a set of end-to-end sanity tests written in Selenium WebDriver (in the case of web) and security tests with OWASP ZAP.

Automated testing

There are many types of testing to be executed as part of the automated delivery process before declaring the build ready for production. The testing may happen by moving the application across multiple environments. Each environment is designated for a particular kind of testing, such as acceptance testing, performance testing, and so on. These environments are adequately monitored to gather the respective metrics.

In a complex microservices environment, testing should not be seen as a last-minute gate check; rather, testing should be considered as a way to improve software quality as well as to avoid last-

minute failures. Shift left testing is an approach of shifting tests as early as possible in the release cycle. Automated testing turns software development to every-day development and every-day testing mode. By automating test cases, we will avoid manual errors as well as the effort required to complete testing.

CI or ARA tools are used to move Docker images across multiple test environments. Once deployed in an environment, test cases are executed based on the purpose of the environment. By default, a set of sanity tests are executed to verify the test environment.

In this section, we will cover all the types of tests that are required in the automated delivery pipeline, irrespective of the environment. We have already considered some types of tests as part of the development and integration environment. Later in this section, we will also map test cases against the environments in which they are executed.

Different candidate tests for automation

In this section, we will explore different types of tests that are candidates for automation when designing an end-to-end delivery pipeline. The key testing types are described as follows.

Automated sanity tests

When moving from one environment to another, it is advisable to run a few sanity tests to make sure that all the basic things are working. This is created as a test pack using JUnit service tests, Selenium WebDriver, or a similar tool. It is important to carefully identify and script all the critical service calls. Especially if the microservices are integrated using synchronous dependencies, it is better to consider these scenarios to ensure that all dependent services are also up and running.

Regression testing

Regression tests ensure that changes in software don't break the system. In a microservices context, the regression tests could be at the service level (Rest API or message endpoints) and written using JUnit or a similar framework, as explained earlier. Service virtualizations are used when dependent services are not available. Karma and Jasmine can be used for web UI testing.

In cases where microservices are used behind web applications, Selenium WebDriver or a similar tool is used to prepare regression test packs, and tests are conducted at the UI level rather than focusing on the service endpoints. Alternatively, BDD tools, such as Cucumber with JUnit or Cucumber with Selenium WebDriver, can also be used to prepare regression test packs. CI tools such as Jenkins or ARA are used to automatically trigger regression test packs. There are other commercial tools, such as TestComplete, that can also be used to build regression test packs.

Automated functional testing

Functional test cases are generally targeted at the UIs that consume the microservices. These are business scenarios based on user stories or features. These functional tests are executed on every build to ensure that the microservice is performing as expected.

BDD is generally used in developing functional test cases. Typically in BDD, business analysts write test cases in a domain-specific language but in plain English. Developers then add scripts to execute these scenarios. Automated web testing tools such as Selenium WebDriver are useful in such scenarios, together with BDD tools such as Cucumber, JBehave, SpecFlow, and so on. JUnit test cases are used in the case of headless microservices. There are pipelines that combine both regression testing and functional testing as one step with the same set of test cases.

Automated acceptance testing

This is much similar to the preceding functional test cases. In many cases, automated acceptance tests generally use the screenplay or journey pattern and are applied at the web application level. The customer perspective is used in building the test cases rather than features or functions. These tests mimic user flows.

BDD tools such as Cucumber, JBehave, and SpecFlow are generally used in these scenarios together with JUnit or Selenium WebDriver, as discussed in the previous scenario. The nature of the test cases is different in functional testing and acceptance testing. Automation of acceptance test packs is achieved by integrating them with Jenkins. There are many other specialized automatic acceptance testing tools available on the market. FitNesse is one such tool.

Performance testing

It is important to automate performance testing as part of the delivery pipeline. This positions performance testing from a gate check model to an integral part of the delivery pipeline. By doing so, bottlenecks can be identified at very early stages of build cycles. In some organizations, performance tests are conducted only for major releases, but in others, performance tests are part of the pipeline. There are multiple options for performance testing. Tools such as JMeter, Gatling, Grinder, and so on can be used for load testing. These tools can be integrated into the Jenkins workflow for automation. Tools such as BlazeMeter can then be used for test reporting.

Application Performance Management tools such as AppDynamics, New Relic, Dynatrace, and so on provide quality metrics as part of the delivery pipeline. This can be done using these tools as part of the performance testing environment. In some pipelines, these are integrated into the functional testing environment to get better coverage. Jenkins has plugins in to fetch measurements.

Real user flow simulation or journey testing

This is another form of test typically used in staging and production environments. These tests continuously run in staging and production environments to ensure that all the critical transactions perform as expected. This is much more useful than a typical URL ping monitoring mechanism. Generally, similar to automated acceptance testing, these test cases simulate user journeys as they happen in the real world. These are also useful to check whether the dependent microservices are up and running. These test cases could be a carved-out subset of acceptance test cases or test packs created using Selenium WebDriver.

Automated security testing

It is extremely important to make sure that the automation does not violate the security policies of the organization. Security is the most important thing, and compromising security for speed is not desirable. Hence, it is important to integrate security testing as part of the delivery pipeline. Some security evaluations are already integrated in the local build environment as well as in the integration environment, such as SonarQube, Find Security Bugs, and so on. Some security aspects are covered as part of the functional test cases. Tools such as BDD-Security, Mittn, and Gauntlt are other security test automation tools following the BDD approach. VAPT can be done using tools such as ImmuniWeb. OWASP ZAP and Burp Suite are other useful tools in security testing.

Exploratory testing

Exploratory testing is a manual testing approach taken by testers or business users to validate the specific scenarios that they think automated tools may not capture. Testers interact with the system in any manner they want without prejudgment. They use their intellect to identify the scenarios that they think some special users may explore. They also do exploratory testing by simulating certain user behavior.

A/B testing, canary testing, and blue-green deployments

When moving applications to production, A/B testing, blue-green deployments, and canary testing are generally applied. A/B testing is primarily used to review the effectiveness of a change and how the market reacts to the change. New features are rolled out to a certain set of users. Canary release is moving a new product or feature to a certain community before fully rolling out to all customers. Blue-green is a deployment strategy from an IT point of view to test the new version of a service. In this model, both blue and green versions are up and running at some point of time and then gracefully migrate from one to the other.

Other nonfunctional tests

High availability and antifragility testing (failure injection tests) are also important to execute before production. This helps developers unearth unknown errors that may occur in a real production scenario. This is generally done by breaking the components of the system to understand their failover behavior. This is also helpful to test circuit breakers and fallback services in the system. Tools such as Simian Army are useful in these scenarios.

Testing in production

Testing in Production (**TiP**) is as important as all the other environments as we can only simulate to a certain extend. There are two types of tests generally executed against production. The first approach is running real user flows or journey tests in a continuous manner, simulating various user actions. This is automated using one of the **Real User Monitoring** (**RUM**) tools, such as AppDynamics. The second approach is to wiretap messages from production, execute them in a staging environment, and then compare the results in production with those in the staging environment.

Antifragility testing

Antifragility testing is generally conducted in a preproduction environment identical to production or even in the production environment by creating chaos in the environment to take a look at how the

application responds and recovers from these situations. Over a period of time, the application gains the ability to automatically recover from most of these failures. Simian Army is one such tool from Netflix. Simian Army is a suite of products built for the AWS environment. Simian Army is for disruptive testing using a set of autonomous monkeys that can create chaos in the preproduction or production environments. Chaos Monkey, Janitor Monkey, and Conformity Monkey are some of the components of Simian Army.

Target test environments

The different test environments and the types of tests targeted on these environments for execution are as follows:

- **Development environment**: The development environment is used to test the coding style checks, bad practices, potential bugs, unit tests, and basic security scanning.
- **Integration test environment**: Integration environment is used for unit testing and regression tests that span across multiple microservices. Some basic security-related tests are also executed in the integration test environment.
- **Performance and diagnostics**: Performance tests are executed in the performance test environment. Application performance testing tools are deployed in these environments to collect performance metrics and identify bottlenecks.
- **Functional test environment**: The functional test environment is used to execute a sanity test and functional test packs.
- **UAT environment**: The UAT environment has sanity tests, automated acceptance test packs, and user journey simulations.
- **Staging**: The preproduction environment is used primarily for sanity tests, security, antifragility, network tests, and so on. It is also used for user journey simulations and exploratory testing.
- **Production**: User journey simulations and RUM tests are continuously executed in the production environment.

Making proper data available across multiple environments to support test cases is the biggest challenge. Delphix is a useful tool to consider when dealing with test data across multiple environments in an effective way.

Continuous deployment

Continuous deployment is the process of deploying applications to one or more environments and configuring and provisioning these environments accordingly. As discussed in Chapter 9, *Managing Dockerized Microservices with Mesos and Marathon*, infrastructure provisioning and automation tools facilitate deployment automation.

From the deployment perspective, the released Docker images are moved to production automatically once all the quality checks are successfully completed. The production environment, in this case, has to be cloud based with a cluster management tool such as Mesos or Marathon. A self-service cloud environment with monitoring capabilities is mandatory.

Cluster management and application deployment tools ensure that application dependencies are properly deployed. This automatically deploys all the dependencies that are required in case any are missing. It also ensures that a minimum number of instances are running at any point in time. In case of failure, it automatically rolls back the deployments. It also takes care of rolling back upgrades in a graceful manner.

Ansible, Chef, or Puppet are tools useful in moving configurations and binaries to production. The Ansible playbook concepts can be used to launch a Mesos cluster with Marathon and Docker support.

Monitoring and feedback

Once an application is deployed in production, monitoring tools continuously monitor its services. Monitoring and log management tools collect and analyze information. Based on the feedback and corrective actions needed, information is fed to the development teams to take corrective actions, and the changes are pushed back to production through the pipeline. Tools such as APM, Open Web Analytics, Google Analytics, Webalizer, and so on are useful tools to monitor web applications. Real user monitoring should provide end-to-end monitoring. QuBit, Boxever, Channel Site, MaxTraffic, and so on are also useful in analyzing customer behavior.

Automated configuration management

Configuration management also has to be rethought from a microservices and DevOps perspective. Use new methods for configuration management rather than using a traditional statically configured CMDB. The manual maintenance of CMDB is no longer an option. Statically managed CMDB requires a lot of mundane tasks to maintain entries. At the same time, due to the dynamic nature of the deployment topology, it is extremely hard to maintain data in a consistent way.

The new styles of CMDB automatically create CI configurations based on an operational topology. These should be discovery based to get up-to-date information. The new CMDB should be capable of managing bare metals, virtual machines, and containers.

Microservices development governance, reference architectures, and libraries

It is important to have an overall enterprise reference architecture and a standard set of tools for microservices development to ensure that development is done in a consistent manner. This helps individual microservices teams to adhere to certain best practices. Each team may identify specialized technologies and tools that are suitable for their development. In a polyglot microservices development, there are obviously multiple technologies used by different teams. However, they have to adhere to the arching principles and practices.

For quick wins and to take advantage of timelines, microservices development teams may deviate from these practices in some cases. This is acceptable as long as the teams add refactoring tasks in their backlogs. In many organizations, although the teams make attempts to reuse something from the enterprise, reuse and standardization generally come as an afterthought.

It is important to make sure that the services are catalogued and visible in the enterprise. This improves the reuse opportunities of microservices.

Summary

In this chapter, you learned about the relationship between microservices and DevOps. We also examined a number of practice points when developing microservices. Most importantly, you learned the microservices development life cycle.

Later in this chapter, we also examined how to automate the microservices delivery pipeline from development to production. As part of this, we examined a number of tools and technologies that are helpful when automating the microservices delivery pipeline. Finally, we touched base with the importance of reference architectures in microservices governance.

Putting together the concepts of microservices, challenges, best practices, and various capabilities covered in this book makes a perfect recipe for developing successful microservices at scale.

Part 3. Module 3

Developing Microservices with Node.js

Learn to develop efficient and scalable microservices for server-side programming in Node.js using this hands-on guide

Chapter 1. Microservices Architecture

Microservices are becoming more and more popular. Nowadays, pretty much every engineer on a green field project should be considering using microservices in order to improve the quality of the systems they build. They should know the architectural principles involving such systems. We will expose the difference between microservices and **Service-Oriented Architecture** (**SOA**). We will also introduce a great platform to write microservices, **Node.js**, which will allow us to create high-performing microservices with very little effort.

In this chapter, you will learn about microservices from the architectural point of view:

- What are microservices?
- Microservice-oriented architectures
- Key benefits
- SOA versus Microservices
- Why Node.js?

Need for microservices

The world of software development has evolved quickly in the past 40 years. One of the key points of this evolution has been the size of these systems. From the days of MS-DOS, we have taken a hundred-fold leap into our present systems. This growth in size creates a need for better ways of organizing code and software components. Usually, when a company grows due to business needs, known as **organic growth**, the software is organized on a monolithic architecture as it is the easiest and quickest way of building software. After few years (or even months), adding new features becomes harder due to the coupled nature of the created software.

Monolithic software

The natural trend for new high-tech companies such as Amazon or Netflix is building their new software using microservices, which is the ideal scenario: they get a huge advantage of microservices-oriented software (through out this book, you will learn how) in order to scale up their new products without a big effort. The problem is that not all companies can plan their software upfront. Instead of planning, these companies build software based on the organic growth experienced: few software components group business flows by affinity. It is not rare to see companies with two big software components: the user-facing website and the internal administration tools. This is usually known as a **monolithic software architecture**.

Some of these companies face big problems when trying to scale the engineering teams. It is hard to coordinate teams that build, deploy, and maintain a single software component. Clashes on releases and reintroduction of bugs are a common problem that drains a large chunk of energy from the teams. One of the solution to this problem (it comes with benefits) is to split the monolithic software into microservices so that the teams are able to specialize in a few smaller modules and autonomous and isolated software components that can be versioned, updated, and deployed without interfering with the rest of the systems of the company.

Splitting the monolith into microservices enables the engineering team to create isolated and autonomous units of work that are highly specialized in a given task such as sending e-mails, processing card payments, and so on.

Microservices in the real world

Microservices are small software components that are specialized in one task and work together to achieve a higher-level task. Forget about software for a second and think about how a company works. When someone applies for a job in a company, he applies for a given position: software engineer, system administrator, office manager. The reason for this can be summarized in one word: specialization. If you are used to work as a software engineer, you will get better with the experience and add more value to the company. The fact that you don't know how to deal with a customer, won't affect your performance as that is not your area of expertise and will hardly add any value to your day-to-day work.

Tip

Specialization is often the key to improve the efficiency. Doing one thing and doing it right is one of the mantras of software development.

A microservice is an autonomous unit of work that can execute one task without interfering with other parts of the system, similar to what a job position is to a company. This has a number of benefits that can be used in favor of the engineering team in order to help scale the systems of a company.

Nowadays, hundreds of systems are built using microservices-oriented architectures, as follows:

- **Netflix**: This is one of the most popular streaming services, it has built an entire ecosystem of applications that collaborate in order to provide a reliable and scalable streaming system used across the globe.
- **Spotify**: This is one of the leading music streaming services in the world, it has built this application using microservices. Every single widget of the application (which is a website exposed as a desktop app using Chromium Embedded Framework) is a different microservice that can be updated individually.

Microservice-oriented architectures

Microservices-oriented architectures have some particularities that makes them desirable for any mid/large-sized company that wants to keep their IT systems resilient and in scale up/down-ready status.

How is it better?

They are not the holy grail of software engineering, but, when handled with care, they become the perfect approach to solve most of the big problems faced by tech-dependent companies.

It is important to keep the key principles of the microservices-oriented architecture's design in mind, such as resilience, composability, elasticity, and so on; otherwise, you could end up with a monolithic application split across different machines that produces problems rather than an elegant solution.

Shortcomings

There is also some criticism around microservices-oriented architectures, as they introduce some problems to deal with, such as latency, traceability, and configuration management that are not present with monolithic-based software. Some of the problems are described as follows:

- **Network latency**: Microservices have a distributed nature so that network latency has to be accounted for
- **Operations overhead**: More servers indicate more maintenance
- **Eventual consistency**: On highly transactional systems, we need to factor into implementation the fact that the data could be inconsistent during a period of time (we will talk about it later in this chapter)

In general, engineers should try to evaluate the pros and cons of this approach and make a decision on whether to use microservices or not in order to fit the business needs.

Microservices-oriented architectures have some particularities that need to be taken into consideration. When a software engineer is writing monolithic software, there are some problems that are completely overlooked due to the nature of the software being built.

For example, imagine that our software needs to send e-mails. In a monolithic software, we would just add the functionality to the core of the application. We might even choose to create a dedicated module to deal with e-mails (which seems like a good idea). Now, imagine that we are creating a microservice and, instead of adding a functionality to a big software artifact, we create a dedicated service that can be deployed and versioned independently. In this case, we will have an extra step that we didn't have to take into consideration, the **network latency**, to reach the new microservice.

In the preceding example, no matter what approach (monolithic or microservices) you are taking to build the software, is not a big deal; for example, if an e-mail is lost, it is not the end of the world. As per definition, the e-mail delivery is not guaranteed, so our application will still work, although we might receive a few complaints from our customers.

Key design principles

There are a few key design principles that need to be taken into consideration when building microservices. There is no golden rule and, as microservices are a recent concept, sometimes there is even a lack of consensus on what practices to follow. In general, we can assume the following design principles:

- Microservices are business units that model the company processes.
- They are smart endpoints that contain the business logic and communicate using simple channels and protocols.
- Microservices-oriented architectures are decentralized by definition. This helps to build robust and resilient software.

Business units, no components

One of the most enjoyable sides of software engineering is creating a new project. This is where you can apply all your creativity, try new architectural concepts, frameworks, or methodologies. Unfortunately, it is not a common situation in a mature company. Usually, what we do is create new components inside the existing software. One of the best design principles that you can follow when creating new components is keeping the coupling as low as possible with the rest of the software, so that it works as an independent unit.

Tip

Keeping a low level of coupling allows a software component to be converted into a microservice with little to no effort.

Consider a real-world example: the application of your company now needs to be able to process payments.

The logical decision here would be creating a new module that knows how to deal with the chosen payment provider (credit cards, PayPal, and so on) and allows us to keep all the payment-related business logic inside of it. Let's define the interface in the following code:

```
public interface PaymentService {
  PaymentResponse processPayment(PaymentRequest request) throws
MyBusinessException;
}
```

This simple interface can be understood by everyone, but it is the key when moving towards microservices. We have encapsulated all the business knowledge behind an interface so that we could theoretically switch the payment provider without affecting the rest of the application—the implementation details are hidden from the outer world.

The following is what we know until now:

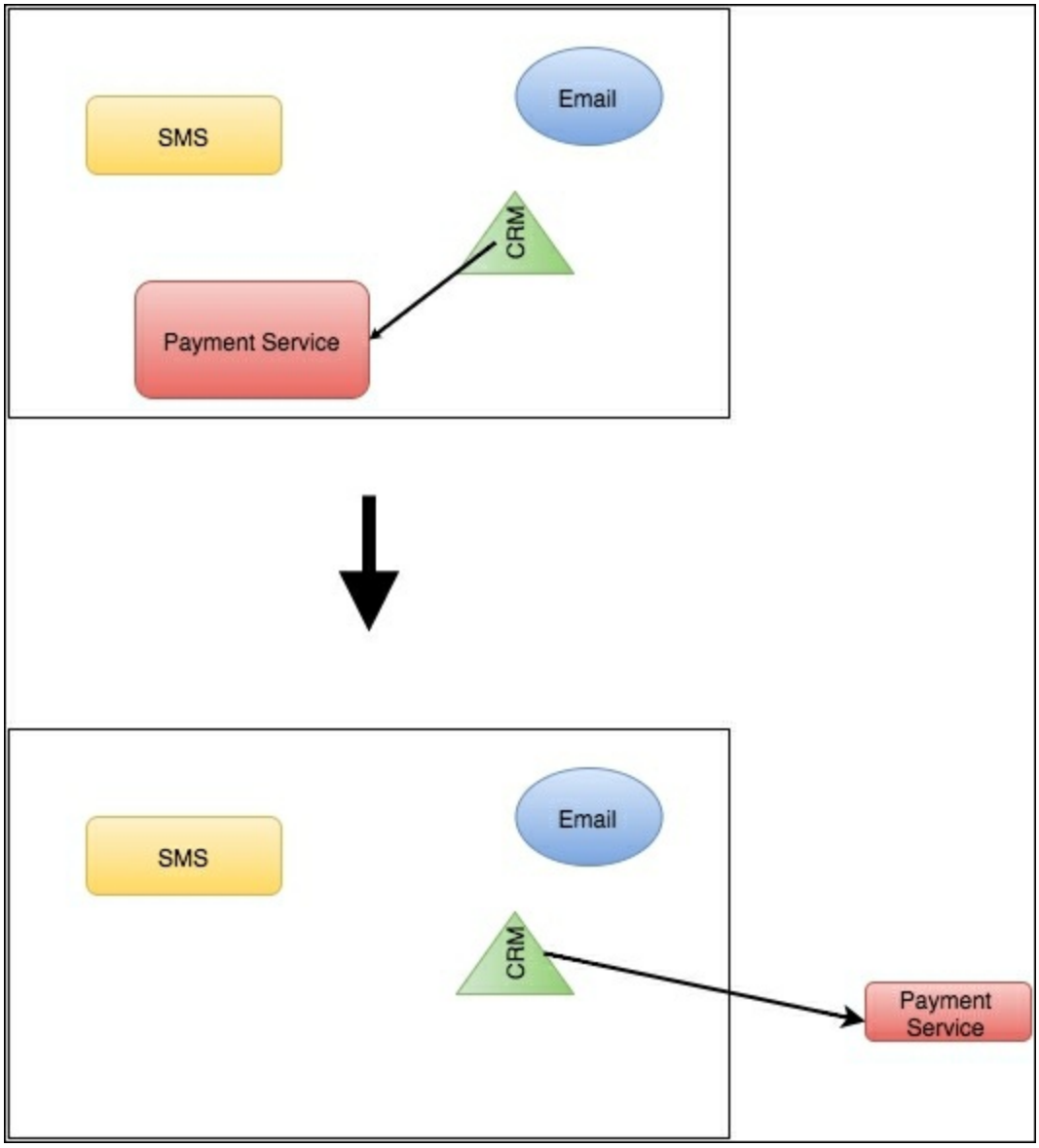
- We know the method name, therefore, we know how to invoke the service
- The method could throw an exception of the `MyBusinessException` type, forcing the calling code to deal with it
- We know that the input parameter is a `PaymentRequest` instance
- The response is a known object

We have created a highly cohesive and loosely coupled business unit. Let's justify this affirmation in the following:

- **Highly cohesive**: All the code inside the payments module will do only one thing, that is, deal with payments and all the aspects of calling a third-party service (connection handling, response codes, and so on), such as a debit card processor.
- **Loosely coupled**: What happens if, for some reason, we need to switch to a new payment

processor? Is there any information bleeding out of the interface? Would we need to change the calling code due to changes in the contract? The answer is no. The implementation of the payment service interface will always be a modular unit of work.

The following diagram shows how a system composed of many components gets one of them (payment service) stripped out into a microservice:



Once this module is implemented, we will be able to process the payments and our monolithic application will have another functionality that could be a good candidate to extract into a microservice.

Now, we can rollout new versions of the payment service, as long as the interface does not change, as well as the contract with the rest of the world (our system or third parties), hasn't changed. That is why it is so important to keep the implementation independent from interfacing, even though the language does not provide support for interfaces.

We can also scale up and deploy as many payment services as we require so that we can satisfy the business needs without unnecessarily scaling the rest of the application that might not be under pressure.

Tip

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the **SUPPORT** tab at the top.
- Click on **Code Downloads & Errata**.
- Enter the name of the book in the **Search** box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Smart services, dumb communication pipes

Hyper Text Transfer Protocol (**HTTP**) is one of the best things to have ever happened to the Internet. Imagine a protocol that was designed to be state-less, but was hacked through the use of cookies in order to hold the status of the client. This was during the age of Web 1.0, when no one was talking about REST APIs or mobile apps. Let's see an example of an HTTP request:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Accept-Ranges: bytes
Connection: close
```

As you can see, it is a *human readable* protocol that does not need to be explained in order to be understood.

Nowadays, it is broadly understood that HTTP is not confined to be used in the Web, and as it was designed, it is now used as a general purpose protocol to transfer data from one endpoint to another. HTTP is all you need for the communication between microservices: a protocol to transfer data and recover from transmission errors (when possible).

In the past few years, especially within the enterprise world, there has been an effort to create smart communication mechanisms such as **BPEL**. BPEL stands for **Business Process Execution Language**, and instead of focusing on communication actions, it focuses on actions around business steps.

This introduces some level of complexity in the communication protocol and makes the business logic of the application bleed into it from the endpoints, causing some level of coupling between the endpoints.

The business logic should stay within the endpoints and not bleed into the communication channel so that the system can be easily tested and scaled. The lesson learned through the years is that the communication layer must be a plain and simple protocol that ensures the transmission of the data and the endpoints (microservices).These endpoints should embed into their implementation the fact that a service could be down for a period of time (this is called resilience, we will talk about this later in this chapter) or the network could cause communication issues.

HTTP usually is the most used protocol when building microservices-oriented software but another interesting option that needs to be explored is the use of queues, such as Rabbit MQ and Kafka, to facilitate the communication between endpoints.

The queueing technology provides a clean approach to manage the communication in a buffered way, encapsulating the complexities of acknowledging messages on highly transactional systems.

Decentralization

One of the major cons of monolithic applications is the centralization of everything on a single (or few) software components and databases. This, more often than not, leads to huge data stores that needs to be replicated and scaled according to the needs of the company and centralized governance of the flows.

Microservices aim for decentralization. Instead of having a huge database, why not split the data according to the business units explained earlier?

Some of the readers could use the transactionality as one of the main reasons for not doing it. Consider the following scenario:

1. A customer buys an item in our microservices-oriented online shop.
2. When paying for the item, the system issues the following calls:
    1. A call to the financial system of the company to create a transaction with the payment.
    2. A call to the warehouse system to dispatch the book.
    3. A call to the mailing system to subscribe the customer to the newsletter.

In a monolithic software, all the calls would be wrapped in a transaction, so if, for some reason, any of the calls fails, the data on the other calls won't be persisted in the database.

When you learn about designing databases, one of the first and the most important principles are summarized by the **ACID** acronym:

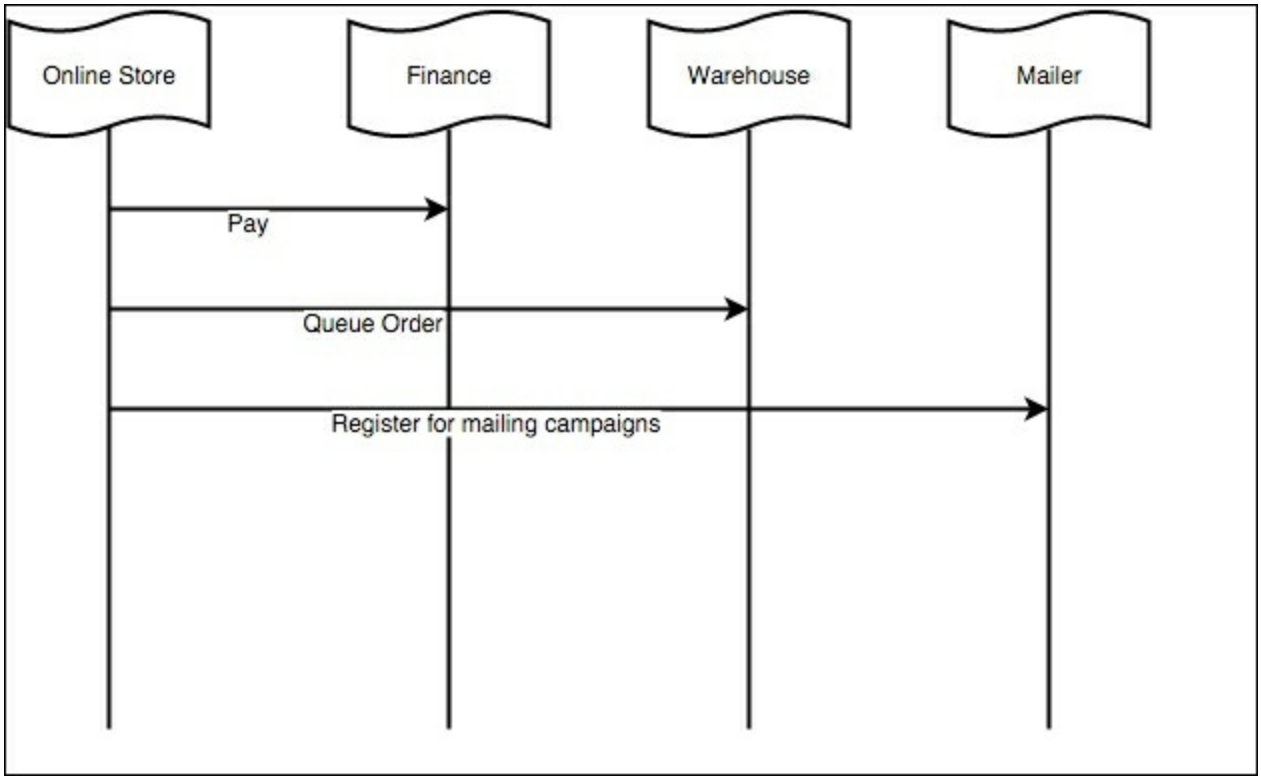
- **Atomicity**: Each transaction will be all or nothing. If one part fails, no changes are persisted on the database.
- **Consistency**: Changes to the data through transactions need to guarantee its consistency.
- **Isolation**: The result of concurrent execution of transactions results in a system state that would be obtained if the transactions were executed serially.
- **Durability**: Once the transaction is committed, the data persists.

On a microservices-oriented software, the ACID principle is not guaranteed globally. Microservices will commit the transaction locally, but there are no mechanisms that can guarantee a 100% integrity of the global transaction. It would be possible to dispatch the book without processing the payment, unless we factor in specific rules to prevent it.

On a microservices-oriented architecture, the transactionality of the data is not guaranteed, so we need to factor the failure into the implementation. A way to solve (although, workaround is a more appropriate word) this problem is decentralizing the governance and data storage.

When building microservices, we need to embed in the design, the fact that one or more components could fail and degrade the functionality according to the availability of the software.

Let's take a look at the following diagram:



This diagram represents the sequence of execution on a monolithic software. A sequential list of calls that, no matter what, are going to be executed following the ACID principle: either all the calls (transactions) succeed or none of them do.

This is only possible as the framework and database engine designers have developed the concept of transactions to guarantee the transactionality of the data.

When working with microservices, we need to account for what the engineers call eventual consistency. After a partial fail on a transaction, each microservice instance should store the information required to recover the transaction so that the information will be eventually consistent. Following the previous example, if we send the book without processing the payment, the payment gateway will have a failed transaction that someone will process later on, making the data consistent again.

The best way to solve this problem is decentralizing the governance. Every endpoint should be able to take a local decision that affects the global scope of the transaction. We will talk more about this subject in Chapter 3, *From the Monolith to Microservices*.

Technology alignment

When building a new software, there is always a concept that every developer should keep in mind: **standards**.

Standards guarantee that your service will be technologically independent so that it will be easy to build the integrations using a different programming language or technologies.

One of the advantages of modeling a system with microservices is that we can choose the right technology for the right job so that we can be quite efficient when tackling problems. When building monolithic software, it is fairly hard to combine technologies like we can do with microservices. Usually, in a monolithic software, we are tied to the technology that we choose in the beginning.

Java Remote Method Invocation (**RMI**) is one example of the non-standard protocols that should be avoided if you want your system to be open to new technologies. It is a great way of connecting software components written in Java, but the developers will struggle (if not fail) to invoke an RMI method using Node.js. This will tie our architecture to a given language, which from the microservices point of view, will kill one of the most interesting advantages: **technology heterogeneity**.

How small is too small?

Once we have decided to model our system as a set of microservices, there is always one question that needs an answer: *how small is too small?*

The answer is always tricky and probably disappointing: *it depends*.

The right size of the microservices in a given system depends on the structure of the company as well as the ability to create software components that are easily manageable by a small team of developers. It also depends on the technical needs.

Imagine a system that receives and processes banking files; as you are probably aware, all the payments between banks are sent in files with a specific known format (such as **Single Euro Payments Area** (**SEPA**) for Euro payments). One of the particularities of this type of systems is the large number of different files that the system needs to know how to process.

The first approach for this problem is tackling it from the microservices point of view, separating it from any other service creating a unit of work, and creating one microservice for each type of file. It will enable us to be able to rollout modifications for the existing file processors without interfering with the rest of the system. It will also enable us to keep processing files even though one of the services is experiencing problems.

The microservices should be as small as needed, but keep in mind that every microservice adds an overhead to the operations team that needs to manage a new service. Try to answer the question *how small is too small?* in terms of manageability, scalability, and specialization. The microservice should be small enough to be managed and scaled up (or down) quickly without affecting the rest of the system, by a single person; and it should do only one thing.

Tip

As a general rule, a microservice should be small enough to be completely rewritten in a sprint.

Key benefits

In the previous topic, we talked about what a microservices-oriented architecture is. I also exposed the design principles that I have learned from experience, as well as showed a few benefits of this type of architecture.

Now, it is time to outline these key benefits and show how they will help us to improve the quality of our software, as well as be able to quickly accommodate the new business requirements.

Resilience

Resilience is defined in Wikipedia as *the ability of a system to cope with change*. I like to think about resilience as the *ability of a system to gracefully recover from an exception* (transitory hardware failure, unexpectedly high network latency, and so on) or a stress period without affecting the performance of the system once the situation has been resolved.

Although it sounds simple, when building microservices-oriented software, the source of problems broadens due to the distributed nature of the system, sometimes making it hard (or even impossible) to prevent all abnormal situations.

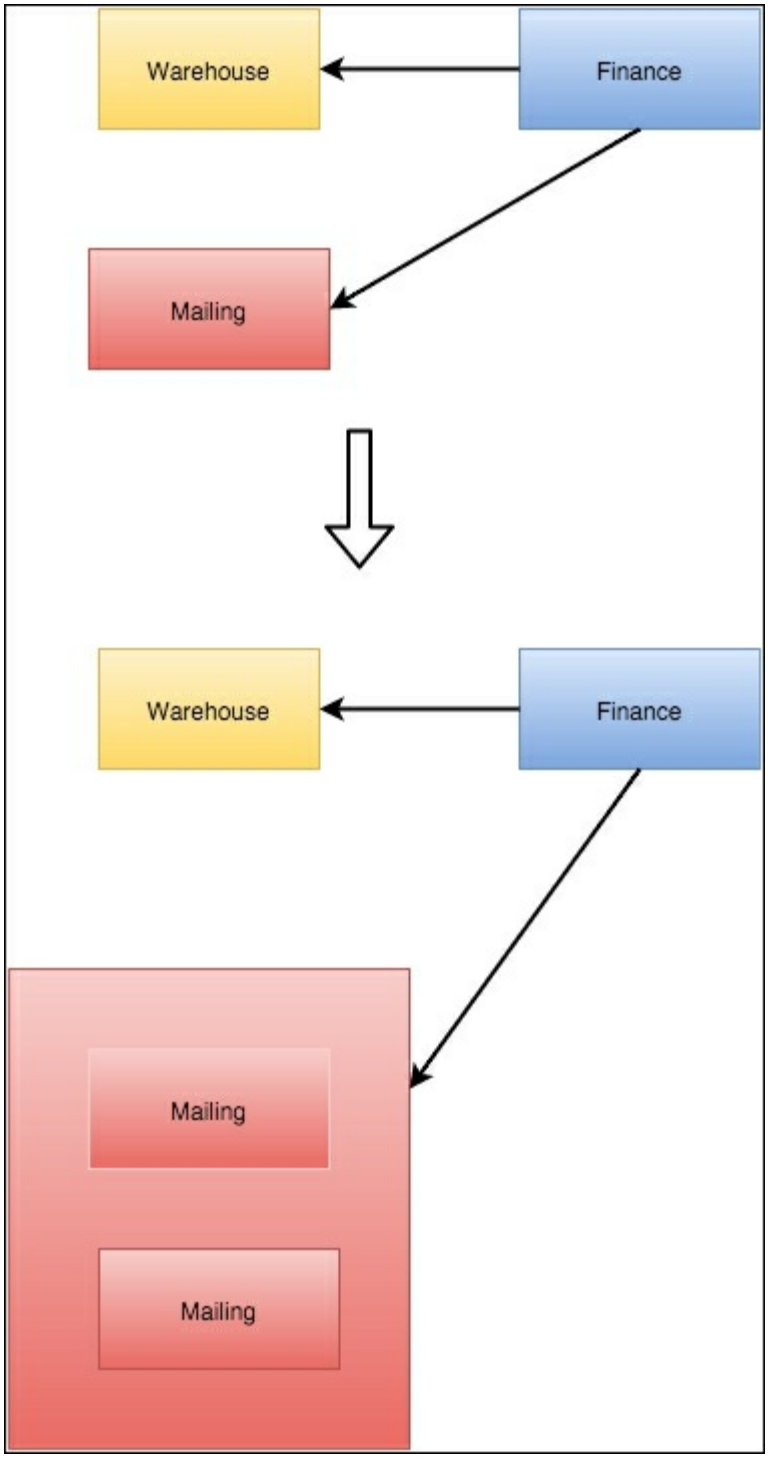
Resilience is the ability to gracefully recover from errors. It also adds another level of complexity: if one microservice is experiencing problems, can we prevent a general failure? Ideally, we should build our system in a way that the service response is degraded instead of resulting in a general failure, although this is not always easy.

Scalability

Nowadays, one of the common problems in companies is the scalability of the systems. If you have worked on a monolithic software before, I am sure that you have experienced capacity problems at some point, alongside the growth of the company.

Usually, these problems are not across all the layers or subsystems of the application. There is always a subsystem or service that performs significantly slower than the rest, causing the entire application to fail if it is not able to cope with the demand.

The following diagram describes how a microservice can be scaled up (two mailing services) without interfering with the rest of the system:



Tip

An example of these weak points in the world of car insurance is the service that calculates the quote for a given list of risk factors. Would it make sense to scale the full application just to satisfy the demand for this particular part? If the answer that you have in mind is *no*, you are one step closer to embracing microservices. Microservices enable you to scale parts of the system as the demand ramps up for a particular area of it.

If our insurance system was a microservice-oriented software, the only thing needed to resolve the high demand for quote calculations would've been to spawn more instances of the microservice (or

microservices) responsible for their calculation. Please bear in mind that scaling up services could add an overhead for operating them.

Technology heterogeneity

The world of software is changing every few months. New languages are coming to the industry as a de facto standard for a certain type of systems. A few years ago, Ruby on Rails appeared at the scene and rose as one of the most used web frameworks for new projects in 2013. Golang (a language created by Google) is becoming a trend nowadays as it combines huge performance with an elegant and simple syntax that can be learned by anyone with some experience in another programming language in a matter of days.

In the past, I have used Python and Java as successful alternatives to write microservices.

Java especially, since Spring Boot was released, is an attractive technology stack to write agile (to write and operate) microservices.

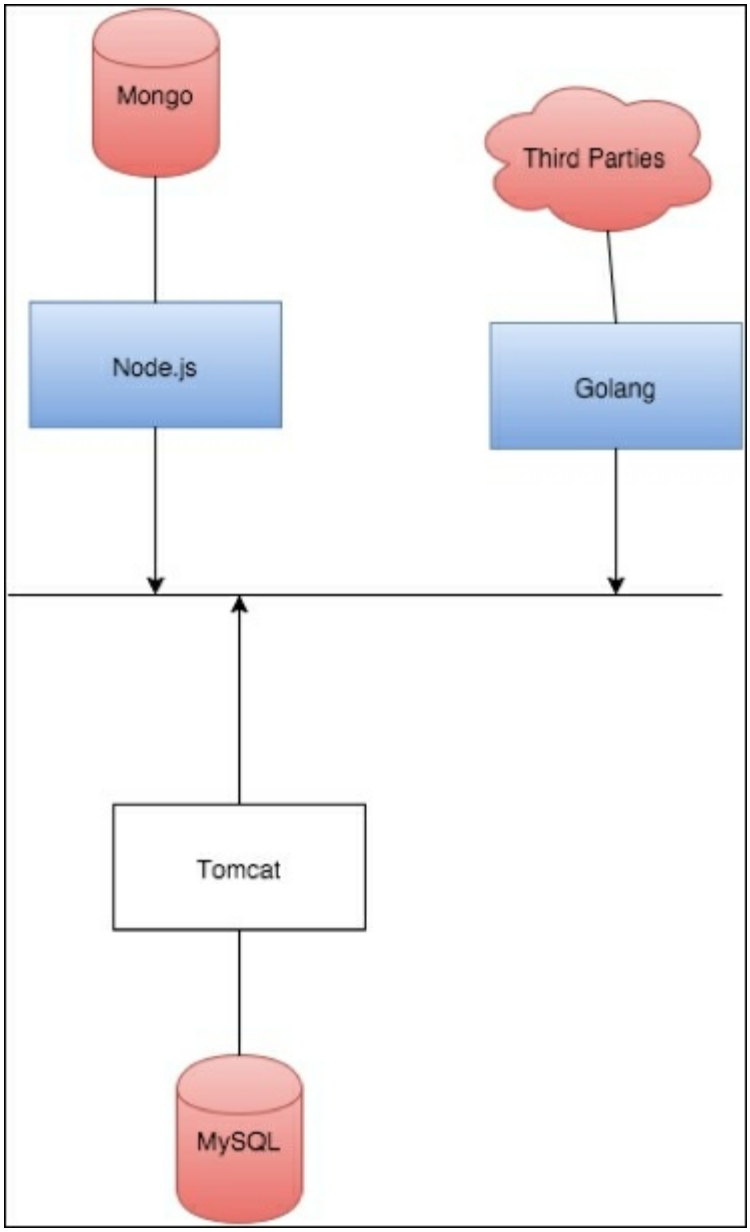
Django, is also a powerful framework on Python to write microservices. Being very similar to Ruby on Rails, it automates database migrations and makes the creation of **CRUD** (**Create Read Update Delete**) services an incredibly easy task.

Node.js took the advantage of a well-known language, JavaScript, to create a new server-side stack that is changing the way engineers create new software.

So, what is wrong in combining all of them? In all fairness, it is an advantage: *we can choose the right tool for the right job*.

Microservices-oriented architectures enable you to do it, as long as the integration technologies are standard. As you learned before, a microservice is a *small and independent piece of software that can operate by itself*.

The following diagram shows how the microservices hide data storage/gathering, having only the communication points in common—making them a good example of low coupling (one service implementation change won't interfere with any other service):



We have talked about performance earlier. There are always parts of our systems that are under more pressure than others. With modern multicore CPUs, parallel (concurrent) programming could solve some of these performance issues, however, Node.js is not a good language to parallelize tasks. We could choose to rewrite the microservice under pressure using a more appropriate language, such as Erlang, to manage concurrency in a more elegant way. It should take no more than two weeks to do it.

There is a downside to using multiple technologies on the same system: the developers and system administrators need to know all (or a few) of them. Companies that embraced microservices usually try to stick with one core technology (in this book, we will be using Node.js) and some auxiliary technologies (although we will be using Docker to manage the deployments, we could use Capistrano or Fabricator to manage the releases).

Replaceability

Replaceability is the ability to change one component of a system without interfering with how the system behaves.

When talking about software, replaceability comes along with low coupling. We should be writing our microservices in a way that the internal logic will not be exposed to the calling services so that the clients of a given service do not need to know about how it is implemented, just the interface. Let's take a look at the following example. It is written in Java as we only need to see the interface to identify the pitfalls:

```
public interface GeoIpService {
  /**
   * Checks if an IP is in the country given by an ISO code.
   **/
  boolean isIn(String ip, String isoCode) throws SOAPFaultException;
}
```

This interface, at first look, is self explanatory. It checks whether a given IP is in a given country and throws a `SOAPFaultException`, which is a big problem.

If we build the client that consumes this service, factoring into their logic, capture, and processing of the `SoapFaultException`, we are exposing internal implementation details to the outer world and making it hard to replace the `GeoIpService` interface. Also, the fact that we are creating a service related to a part of our application logic is an indication of the creation of a **bounded context**: a highly cohesive service or set of services that work together to achieve one purpose.

Independence

No matter how hard we try, the human brain is not designed to solve complex problems. The most efficient mode of functioning for the human brain is one thing at the time so that we *break down complex problems into smaller ones*. Microservices-oriented architectures should follow this approach: all the services should be independent and interact through the interface up to a point that they can be developed by different groups of engineers without any interaction, aside from agreeing the interfaces. This will enable a company adopting microservices to scale up, or down, the engineering teams, depending on the business needs, making the business agile in responding to peaks of demand or periods of quietness.

Why is replaceability important?

In a previous section, we talked about the right size of a microservice. As a general rule of thumb, a team should be able to rewrite and deploy a microservice in a sprint. The reason behind it is the **technical debt**.

I would define technical debt as the deviation from the original technical design to deliver the expected functionality within a planned deadline. Some of these sacrifices or wrong assumptions often lead to poorly written software that needs to be completely refactored or rewritten.

In the preceding example, the interface is exposing to the outer world the fact that we are using **SOAP** to call a web service, but we will need to change the code on the client side as a REST client has certainly nothing to do with SOAP exceptions.

Easy to deploy

Microservices should be easy to deploy.

Being software developers, we are well aware that a lot of things could go wrong, preventing a software from being deployed.

Microservices, as stated before, should be easy to deploy for a number of reasons, as stated in the following list:

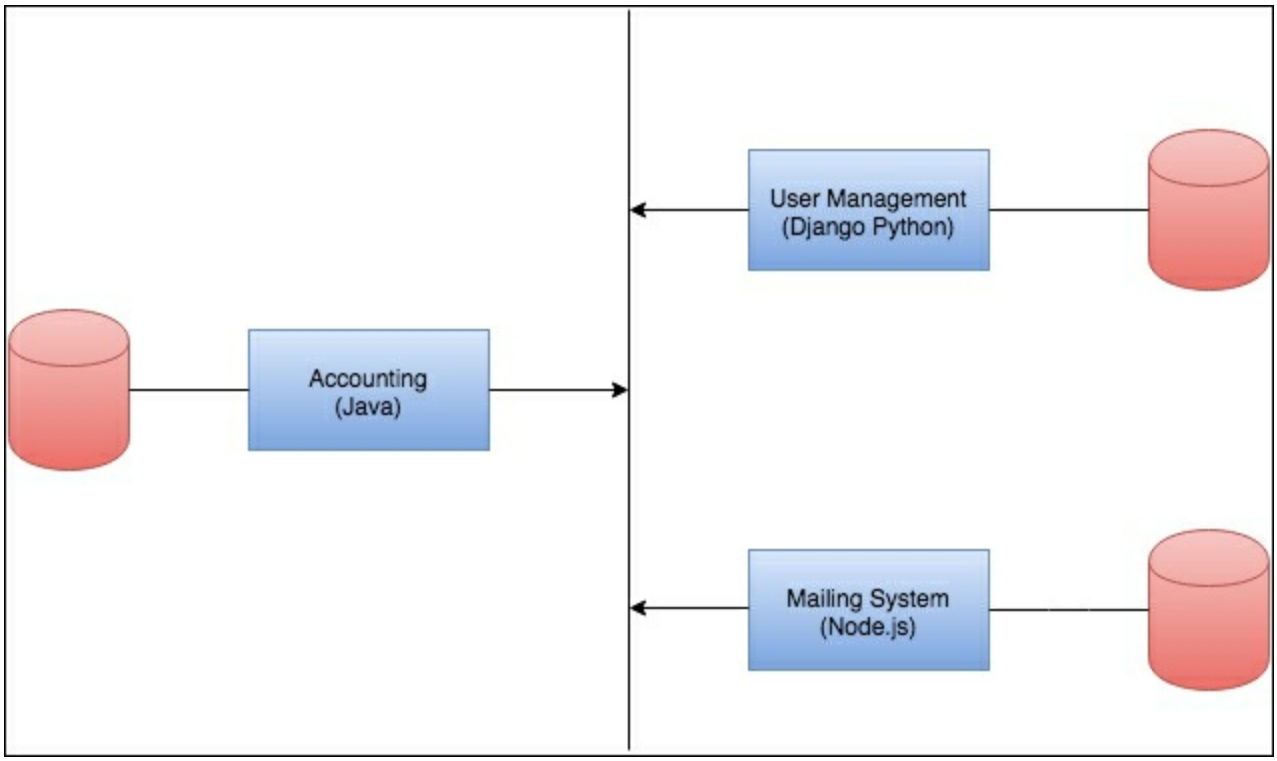
- Small amount of business logic (remember the *two weeks re-write from scratch* rule of thumb) leading into simpler deployments.
- Microservices are autonomous units of work, so upgrading a service is a contained problem on a complex system. No need to re-deploy the entire system.
- Infrastructure and configuration on microservices architectures should be automated as much as possible. Later in the book, we will learn how to use Docker to deploy microservices and what are the benefits over the traditional deployment techniques are.

SOA versus microservices

Service-Oriented Architectures (**SOA**) has been around for a number of years. SOA is a great principle to design software. They are self-contained components providing services to other components. As we agreed before, it is all about maintaining low coupling on the different modules of the system as if it was a puzzle so that we can replace the pieces without causing a big impact on the overall system.

In principle, SOA looks very similar to microservices architectures. So what is the difference?

Microservices are fine-grained SOA components. In other words, a single SOA component can be decomposed in a number of microservices that can work together in order to provide the same level of functionality:



Tip

Microservices are fine-grained SOA components. They are lightweight services with a narrow focus.

Another difference between microservices and SOA is the technologies used for interconnecting and writing the services.

J2EE is a technology stack that was designed to write SOA architectures as it enforced enterprise standards. Java Naming and Directory Interface, Enterprise Java Beans, and **Enterprise Service Bus** (**ESB**) were the ecosystems where SOA applications were built and maintained. Although ESB is a standard, very few engineers who graduated after 2005 have heard about ESB, even fewer have used

it, and nowadays the modern frameworks such as Ruby on Rails do not even consider such complex pieces of software.

On the other hand, microservices enforce the use of standards (such as HTTP) that are broadly known and broadly interoperable. We can choose the right language or tool to build a component (microservice) following one of the key benefits explained earlier in this chapter, in the *Technology heterogeneity* section.

Aside from the technology stack and the size of the services, there is an even bigger difference between SOA and microservices: the domain model. Earlier in this chapter, we have talked about decentralization. Decentralization of the governance, but, moreover, decentralization of the data. In a microservices-based software, every microservice should store its own data locally, isolating the domain models to a single service; whereas, on an SOA oriented-software, the data is usually stored in a few big databases and the services share the domain models.

Why Node.js?

A few years ago, I didn't believe in Node.js. To me, it was a trend more than a real tool to solve problems… JavaScript in the server? That didn't look right. In all fairness, I didn't even like JavaScript. Then, the modern frameworks such as jQuery or Angular.js came to the rescue. They solved one of the problems, which was the cross-browser compatibility. Where before we needed to factor in at least three different browsers, after jQuery all this logic was nicely encapsulated in a library so that we didn't need to worry about compatibility as long as we followed the jQuery documentation.

Then, JavaScript became more popular. Suddenly, all the internal tools were written with **Single-Page Application** (**SPA**) frameworks with a heavy usage of JavaScript, therefore, the majority of developers nowadays, one way or another, are proficient in JavaScript.

Then, someone decided to take JavaScript out of the browser, which was a great idea. Rhino, Node.js, and Nashorn are examples of runtimes that can execute standalone JavaScript. Some of them can even interact with the Java code, enabling the developer to import Java classes into a JavaScript program, which gives you the access to an endless set of frameworks already written in Java.

Let's focus on **Node.js**. Node.js is the perfect candidate for microservices-oriented architectures for a number of reasons, as stated in the following list:

- Easy to learn (although it can be hard to master)
- Easy to scale
- Highly testable
- Easy to deploy
- Dependency management through **npm**
- There are hundreds of libraries to integrate with the majority of standard protocols

These reasons, along with others that we will develop in the following chapters, make Node.js the perfect candidate for building solid microservices.

API aggregation

Seneca is the framework that I have chosen for development in the following chapters. One of the most attractive characteristics of Seneca is API aggregation.

API aggregation is an advanced technique to compose an interface by aggregating different functionalities (plugins, methods, and so on) to it.

Let's take a look at the following example:

```
var express = require('express');
var app = express();

app.get('/sayhello', function (req, res) {
  res.send('Hello World!');
});
app.get('/saygoodbye', function(req, res) {
  res.send('Bye bye!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('App listening at http://%s:%s', host, port);
});
```

The preceding example uses Express, a very popular web framework for Node.js. This framework is also built around the API aggregation technique. Let's take a look at the fourth and seventh lines. In these lines, the developer registers two methods that are to be executed when someone hits the URLs `/sayhello` and `/saygoodbye` with a GET request. In other words, the application is composed of different smaller and independent implementations that are exposed to the outer world on a single interface, in this case, an app listening on the `3000` port.

In the following chapters, I will explain why this property is important and how to take advantage of it when building (and scaling) microservices.

The future of Node.js

JavaScript was first designed to be a language executed in the web browser. For those who worked or studied, using C/C++ was very familiar and that was the key for its adoption as a standard for the dynamic manipulation of documents in Web 2.0. **Asynchronous JavaScript and XML** (**AJAX**) was the detonator for JavaScript growth. Different browsers had different implementations of the request objects so that the developers had a hard time to write a cross-browser code.

The lack of standards led to the creation of many frameworks that encapsulated the logic behind AJAX, making easy-to-write cross-browser scripts.

JavaScript is a script language. It was not designed to be object oriented, neither was it designed to be the language of choice for large applications as the code tends to get chaotic and it is hard to enforce standards across different companies on how the code should be laid out. Every single company where I worked has different *best practices* and some of them are even contradictory.

European Computer Manufacturers Association (**ECMA**) came to the rescue. **ECMAScript 6**, the next standard for ECMA languages (JavaScript, ActionScript, Rhino, and so on) introduces the concept of classes, inheritance, collections, and a number of interesting features that will make the development of JavaScript software easier and more standard than the actual V8 specification.

One of these features that I consider more interesting is the introduction of the **class** keyword that allows us to model our JavaScript software with objects.

At the moment, the majority of browsers support a large number of these features, but when it comes to Node.js, only a few of them are implemented by default and some of them are implemented by passing special flags to the interpreter (harmony flags).

In this book, I will try to avoid the ECMAScript 6 features, sticking to the V8 specification as it is widely known by the majority of developers and, once someone knows JavaScript V8, it is fairly easy to ramp up on ECMAScript 6.

Summary

In this chapter, we studied the key concepts around microservices, as well as the best practices to be followed when designing high-quality software components towards building robust and resilient software architectures that enable us to respond quickly to the business needs.

You have also learned the key benefits such as the possibility of using the right language for the right service (technology heterogeneity) on the microservices-oriented architectures as well as some of the pitfalls that could make our life harder, such as the overhead on the operational side caused by the distributed nature of the microservices-oriented architectures.

Finally, we discussed why Node.js is a great tool for building microservices, as well as how we could benefit from JavaScript to build high-quality software components through techniques like API aggregation.

In the following chapters, we will be developing the concepts discussed in this chapter, with code examples and further explanation about the topics I have learned over the years.

As explained before, we will focus on the V8 version of JavaScript, but I will provide some hints on how to easily write upgradeable components to embrace ECMAScript 6.

Chapter 2. Microservices in Node.js – Seneca and PM2 Alternatives

In this chapter, you will mainly learn about two frameworks, **Seneca** and **PM2**, and why they are important for building microservices. We will also get to know the alternatives to these frameworks in order to get a general understanding of what is going on in the Node.js ecosystem. In this chapter, we are going to focus on the following topics:

- **Need for Node.js**: In this section, we are going to justify the choice of Node.js as a framework to build our microservices-oriented software. We will walk through the software stack required to use this awesome technology.
- **Seneca – a microservices framework**: In this section, you will learn the basics of Seneca and why it is the right choice if we want to keep our software manageable. We will explain how to integrate Seneca with Express (the most popular web server in Node.js) in order to follow the industry standards.
- **PM2**: PM2 is the best choice to run Node.js applications. No matter what your problem in deploying your ecosystem of apps is, PM2 will always have a solution for it.

Need for Node.js

In the previous chapter, I mentioned that I wasn't a big fan of Node.js in the past. The reason for this was that I wasn't prepared to cope with the level of standardization that JavaScript was undergoing.

JavaScript in the browser was painful. Cross-browser compatibility was always a problem and the lack of standardization didn't help to ease the pain.

Then Node.js came and it was easy to create highly scalable applications due to its non-blocking nature (we will talk about it later in this chapter) and it was also very easy to learn as it was based on JavaScript, a well-known language.

Nowadays, Node.js is the preferred choice for a large number of companies across the world, as well as the number one choice for aspects that require a non-blocking nature in the server, such as web sockets.

In this book, we will primarily (but not only) use Seneca and PM2 as the frameworks for building and running microservices, but it does not mean that the alternatives are not good.

There are few alternatives in the market such as **restify** or **Express** for building applications and **forever** or **nodemon** to run them. However, I find Seneca and PM2 to be the most appropriate combination for building microservices for the following reasons:

- PM2 is extremely powerful regarding application deployments
- Seneca is not only a framework to build microservices, but it is also a paradigm that reshapes what we know about object-oriented software

We will be using Express in a few examples in the chapters of this book and we will also discuss how to integrate Seneca in Express as a middleware.

However, before that, let's discuss some concepts around Node.js that will help us to understand those frameworks.

Installing Node.js, npm, Seneca, and PM2

Node.js is fairly easy to install. Depending on your system, there is an installer available that makes the installation of Node.js and **npm** (**Node Package Manager**) a fairly simple task. Simply double-click on it and follow the instructions. At the time of writing this book, there are installers available for Windows and OSX.

However, the advanced users, especially DevOps engineers, will need to install Node.js and npm from the sources or binaries.

Note

Both Node.js and npm programs come bundled together in a single package that we can download for various platforms from the Node.js website (either sources or binaries):

https://nodejs.org/en/download/

For the Chef users, a popular configuration management software to build servers, there are few options available, but the most popular is the following recipe (for those unfamiliar with Chef, a recipe is basically a script to install or configure software in a server through Chef):

https://github.com/redguide/nodejs

At the time of writing this book, there are binaries available for Linux.

Learning npm

npm is a software that comes with Node.js and enables you to pull dependencies from the Internet without worrying about their management. It can also be used to maintain and update dependencies, as well as create projects from scratch.

As you probably know, every node app comes with a `package.json` file. This file describes the configuration of the project (dependencies, versions, common commands, and so on). Let's see the following example:

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "test project",
  "main": "index.js",
  "scripts": {
  "test": "grunt validate --verbose"
  },
  "author": "David Gonzalez",
  "license": "ISC"
}
```

The file itself is self-explanatory. There is an interesting section in the file—`scripts`.

In this section, we can specify the command that is used to run for different actions. In this case, if we run `npm test` from the terminal, npm will execute `grunt validate --verbose`.

Node applications are usually as easy to run as executing the following command:

```
node index.js
```

In the root of your project, consider that the bootstrapping file is `index.js`. If this is not the case, the best thing you can do is add a subsection in the `scripts` section in `package.json`, as follows:

```
"scripts": {
  "test": "grunt validate --verbose"
  "start": "node index.js"
},
```

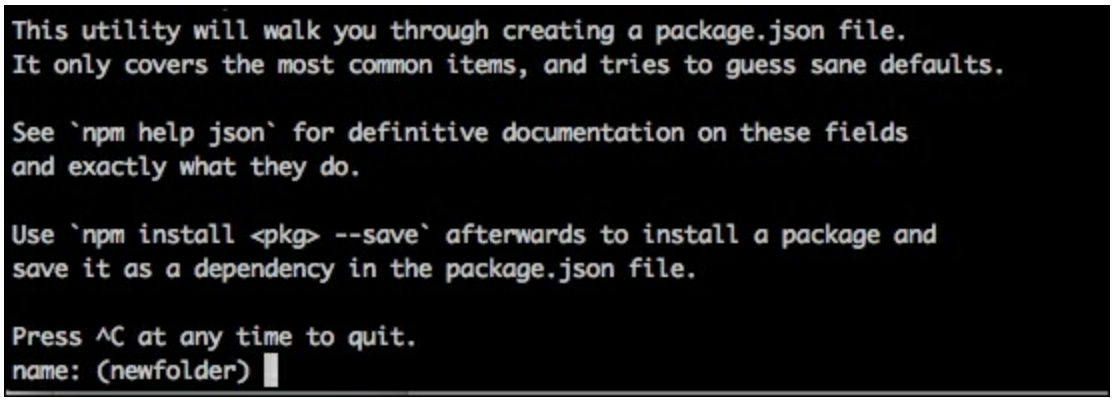
As you can see, now we have two commands executing the same program:

```
node index.js
npm start
```

The benefits of using `npm start` are quite obvious—uniformity. No matter how complex your application is, `npm start` will always run it (if you have configured the `scripts` section correctly).

Let's install Seneca and PM2 on a clean project.

First, execute `npm init` in a new folder from the terminal after installing Node.js. You should get a prompt similar to the following image:



npm will ask you for a few parameters to configure your project, and once you are done, it writes a `package.json` file with content similar to the preceding code.

Now we need to install the dependencies; npm will do that for us. Just run the following command:

```
npm install --save seneca
```

Now, if you inspect `package.json` again, you can see that there is a new section called

`dependencies` that contains an entry for Seneca:

```
"dependencies": {
  "seneca": "^0.7.1"
}
```

This means that from now on, our app can require the Seneca module and the `require()` function will be able to find it. There are a few variations of the `save` flag, as follows:

- `save`: This saves the dependency in the `dependencies` section. It is available through all the development life cycle.
- `save-dev`: This saves the dependency in the `devDependencies` section. It is only available in development and does not get deployed into production.
- `save-optional`: This adds a dependency (such as `save`), but lets npm continue if the dependency can't be found. It is up to the app to handle the lack of this dependency.

Let's continue with PM2. Although it can be used as a library, PM2 is mainly a command tool, like `ls` or `grep` in any Unix system. npm does a great job installing command-line tools:

```
npm install -g pm2
```

The `-g` flags instruct npm to globally install PM2, so it is available in the system, not in the app. This means that when the previous command finishes, `pm2` is available as a command in the console. If you run `pm2 help` in a terminal, you can see the help of PM2.

Our first program – Hello World

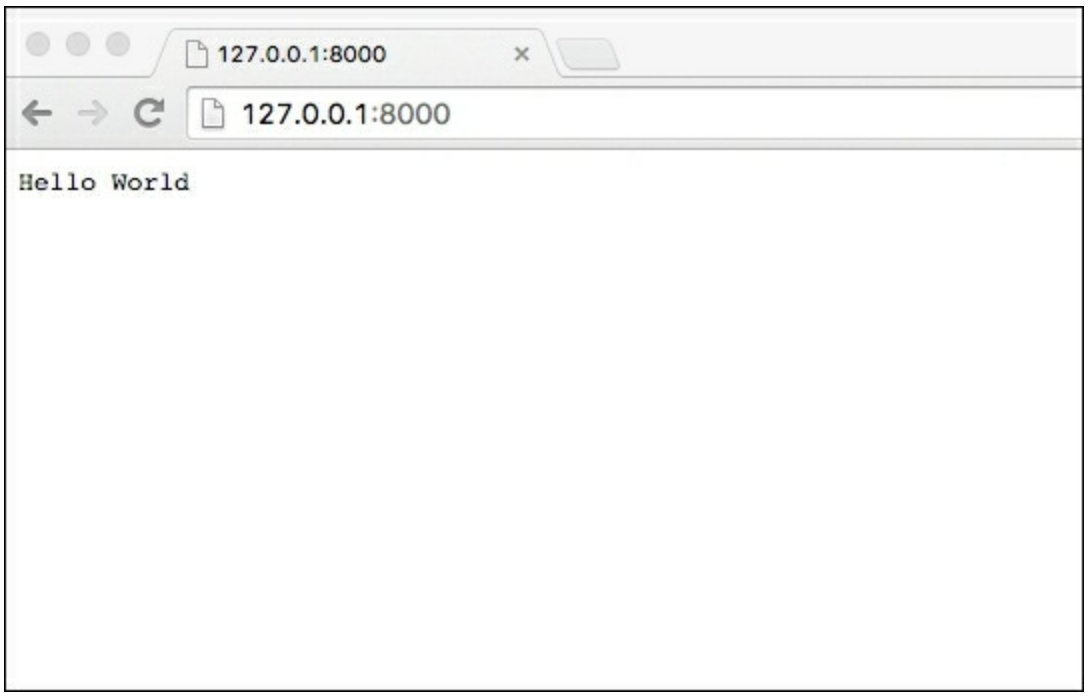
One of the most interesting concepts around Node.js is simplicity. You can learn Node.js in few days and master it in a few weeks, as long as you are familiar with JavaScript. Code in Node.js tends to be shorter and clearer than in other languages:

```
var http = require('http');

var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});

server.listen(8000);
```

The preceding code creates a server that listens on the `8000` port for requests. If you don't believe it, open a browser and type `http://127.0.0.1:8000` in the navigation bar, as shown in the following screenshot:



Let's explain the code:

- The first line loads the `http` module. Through the `require()` instruction, we ask the node to load the `http` module and assign the export of this module to the `http` variable. Exporting language elements is the way that Node.js has to expose functions and variables to the outer world from inside a module.
- The second construction in the script creates the HTTP server. The `http` module creates and exposes a method called `createServer()` that receives a function (remember JavaScript treats functions as first-level objects so that they can be passed as other functions arguments) as a

parameter that, in the Node.js world, is called **callback**. A callback is an action to be executed as a response to an event. In this case, the event is that the script receives an HTTP request. Node.js has a heavy usage of callbacks due to its thread model. Your application will always be executed on a single thread so that not blocking the application thread while waiting for operations to complete and prevents our application from looking stalled or hanged. Otherwise, your program won't be responsive. We'll come back to this in Chapter 4, *Writing Your First Microservice in Node.js*.

- In the next line, `server.listen(8000)` starts the server. From now on, every time our server receives a request, the callback on the `http.createServer()` function will be executed.

This is it. Simplicity is the key to Node.js programs. The code allows you to go to the point without writing tons of classes, methods, and config objects that complicate what, in the first instance, can be done much more simply: write a script that serves requests.

Node.js threading model

Programs written in Node.js are single-threaded. The impact of this is quite significant; in the previous example, if we have ten thousand concurrent requests, they will be queued and satisfied by the Node.js event loop (it will be further explained in Chapter 4, *Writing Your First Microservice in Node.js* and Chapter 6, *Testing and Documenting Node.js Microservices*) one by one.

At first glance, this sounds wrong. I mean, the modern CPUs can handle multiple parallel requests due to their multicore nature. So, what is the benefit of executing them in one thread?

The answer to this question is that Node.js was designed to handle asynchronous processing. This means that in the event of a slow operation such as reading a file, instead of blocking the thread, Node.js allows the thread to continue satisfying other events, and then the control process of the node will execute the method associated with the event, processing the response.

Sticking to the previous example, the `createServer()` method accepts a callback that will be executed in the event of an HTTP request, but meanwhile, the thread is free to keep executing other actions.

The catch in this model is what Node.js developers call the callback hell. The code gets complicated as every single action that is a response to a blocking action has to be processed on a callback, like in the previous example; the function used as a parameter to the `createServer()` method is a good example.

Modular organization best practices

The source code organization for big projects is always controversial. Different developers have different approaches to how to order the source code in order to keep the chaos away.

Some languages such as Java or C# organize the code in packages so that we can find source code files that are related inside a package. As an example, if we are writing a task manager software, inside the `com.taskmanager.dao` package we can expect to find classes that implement the **data access object** (**DAO**) pattern in order to access the database. In the same way, in the `com.taskmanager.dao.domain.model` package, we can find all the classes that represent model objects (usually tables) in our application.

This is a convention in Java and C#. If you are a C# developer, and you start working on an existing project, it only takes you a few days to get used to how the code is structured as the language enforces the organization of the source.

Javascript

JavaScript was first designed to be run inside the browser. The code was supposed to be embedded in HTML documents so that the **Document Object Model** (**DOM**) could be manipulated to create dynamic effects. Take a look at the following example:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title of the document</title>
</head>
<body>
  Hello <span id="world">Mundo</span>
  <script type="text/javascript">
  document.getElementById("world").innerText = 'World';
  </script>
</body>
</html>
```

As you can see, if you load this HTML on a browser, the text inside the `span` tag with the `id` as `world` is replaced when the page loads.

In JavaScript, there is no concept of dependency management. JavaScript can be segregated from the HTML into its own file, but there is no way (for now) to include a JavaScript file into another JavaScript file.

This leads to a big problem. When the project contains dozens of JavaScript files, the assets management become more of an art than an engineering effort.

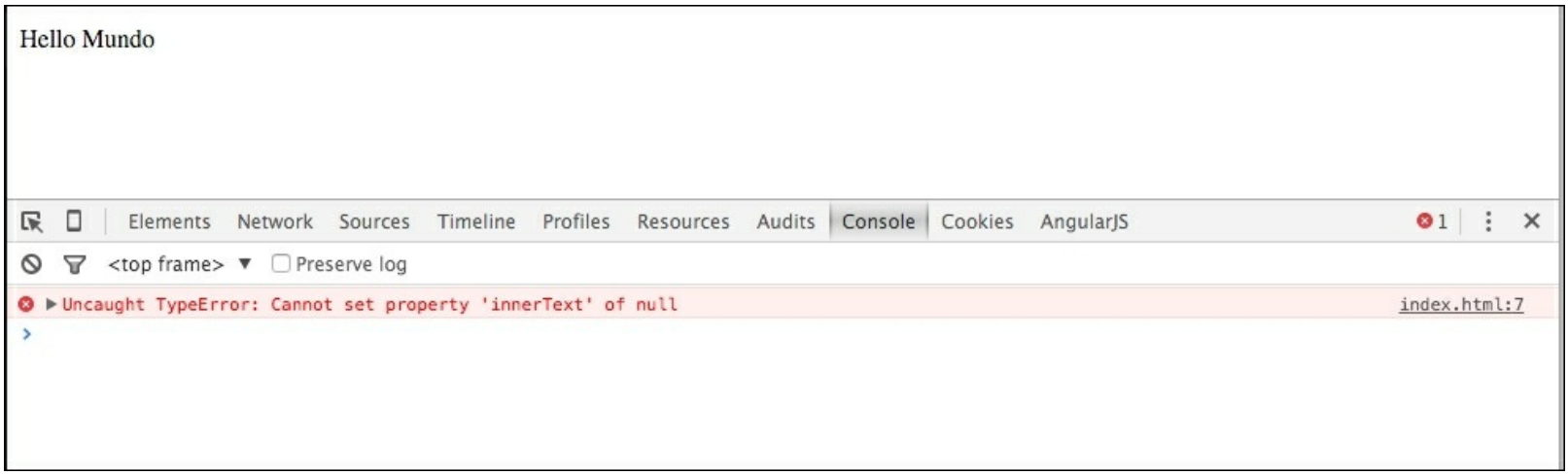
The order in which you import the JavaScript files becomes important as the browser executes the

JavaScript files as it finds them. Let's reorder the code in the previous example to demonstrate it, as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title of the document</title>
  <script type="text/javascript">
    document.getElementById("world").innerText = 'World';
  </script>
</head>
<body>
  Hello <span id="world">Mundo</span>

</body>
</html>
```

Now, save this HTML in an `index.html` file and try to load it in any browser, as shown in the following image:



In this case, I have used Chrome and the console shows an **Uncaught TypeError: Cannot set property 'innerText' of null** error in line 7.

Why is that happening?

As we explained earlier, the browser *executes* the code as it is found, and it turns out that when the browser executes the JavaScript, the `world` element does not exist yet.

Fortunately, Node.js has solved the dependency-loading problem using a very elegant and standard approach.

SOLID design principles

When talking about microservices, we always talk about modularity, and modularity always boils

down to the following (**SOLID**) design principles:

- **Single** responsibility principle
- **Open** for extension, closed for modification
- **Liskov** substitution
- **Interface** segregation
- **Dependency** inversion (inversion of control and dependency injection)

You want your code to be organized in modules. A module is an aggregation of code that does something simple, such as manipulating strings, and it does it well. The more functions (or classes, utilities, and so on) your module contains, the less cohesive it is, and we are trying to avoid that.

In Node.js, every JavaScript file is a module by default. We can also use folders as modules, but let's focus on files:

```
function contains(a, b) {
  return a.indexOf(b) > -1;
}

function stringToOrdinal(str) {
  var result = ""
  for (var i = 0, len = str.length; i < len; i++) {
    result += charToNumber(str[i]);
  }
  return result;
}

function charToNumber(char) {
  return char.charCodeAt(0) - 96;
}

module.exports = {
  contains: contains,
  stringToOrdinal: stringToOrdinal
}
```

The preceding code represents a valid module in Node.js. In this case, the module contains three functions, where two of them are exposed to the outside of the module.

In Node.js, this is done through the `module.exports` variable. Whatever you assign to this variable is going to be visible by the calling code so that we can simulate private content on a module, such as the `charToNumber()` function in this case.

So, if we want to use this module, we just need to `require()` it, as follows:

```
var stringManipulation = require("./string-manipulation");
console.log(stringManipulation.stringToOrdinal("aabb"));
```

This should output `1122`.

Let's go back to the SOLID principles and see how our module looks:

- **Single responsibility principle**: Our module only deals with strings
- **Open for extension, closed for modification**: We can add more functions, but the ones that we have are correct and they can be used to build new functions in the module
- **Liskov substitution**: We will skip this one, as the structure of the module is irrelevant to fulfil this principle
- **Interface segregation**: JavaScript is not a language that counts with an interface element such as Java or C#, but in this module, we exposed the interface, and the `module.exports` variable will act as a contract for the calling code and the change in our implementation won't affect how the module is being called
- **Dependency inversion**: Here is where we fail, not fully, but enough to reconsider our approach

In this case, we require the module, and the only way to interact with it is through the global scope. If, inside the module, we want to interact with data from outside, the only possible option is to create a global variable (or function) prior to requiring the module, and then assume that it is always going to be in there.

Global variables are a big problem in Node.js. As you are probably aware, in JavaScript, if you omit the `var` keyword when declaring a variable, it is automatically global.

This, coupled with the fact that intentional global variables create a data coupling between modules (coupling is what we want to avoid at any cost), is the reason to find a better approach to how to define the modules for our microservices (or in general).

Let's restructure the code as follows:

```
function init(options) {

  function charToNumber(char) {
    return char.charCodeAt(0) - 96;
  }

  function StringManipulation() {
  }

  var stringManipulation = new StringManipulation();

  stringManipulation.contains = function(a, b) {
    return a.indexOf(b) > -1;
  };

  stringManipulation.stringToOrdinal = function(str) {
    var result = ""
    for (var i = 0, len = str.length; i < len; i++) {
      result += charToNumber(str[i]);
    }
    return result;
  }
```

```
  return stringManipulation;
}

module.exports = init;
```

This looks a bit more complicated, but once you get used to it, the benefits are enormous:

- We can pass configuration parameters to the module (such as debugging information)
- Avoids the pollution of global scope as if everything is wrapped inside a function, and we enforce the *use strict* configuration (this avoids declarations without `var` with a compilation error)
- Parameterizing a module makes it easy to mock behaviors and data for testing

In this book, we are going to be writing a good amount of code to model systems from the microservices prospective. We will try to keep this pattern as much as we can so that we can see the benefits.

One of the library that we are going to be using to build microservices, Seneca, follows this pattern, as well as a large number of libraries that can be found on Internet.

Seneca – a microservices framework

Seneca is a framework for building microservices written by Richard Rodger, the founder and CTO of nearForm, a consultancy that helps other companies design and implement software using Node.js. Seneca is about simplicity, it connects services through a sophisticated pattern-matching interface that abstracts the transport from the code so that it is fairly easy to write highly scalable software.

Let's stop talking and see some examples:

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
})

seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
  var product = msg.left * msg.right
  respond( null, { answer: product } )
})

seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2}, console.log)
    seneca.act({role: 'math', cmd: 'product', left: 3, right: 4}, console.log)
```

As you can see, the code is self-explanatory:

- Seneca comes as a module, so the first thing that needs to be done is to `require()` it. Seneca package is wrapped in a function, so invoking the function initializes the library.
- Next two instructions are related to a concept explained in Chapter 1, *Microservices Architecture*: API composition. The `seneca.add()` method instructs Seneca to add a function that will be invoked with a set of patterns. For the first one, we specify an action that will take place when Seneca receives the `{role: math, cmd: sum}` command. For the second one, the pattern is `{role: math, cmd: product}`.
- The last line sends a command to Seneca that will be executed by the service that matches the pattern passed as the first parameter. In this case, it will match the first service as `role` and `cmd` match. The second call to act will match the second service.

Write the code in a file called `index.js` in the project that we created earlier in this chapter (remember that we installed Seneca and PM2), and run the following command:

```
node index.js
```

The output will be something similar to the following image:



We will talk about this output later in order to explain exactly what it means, but if you are used to enterprise applications, you can almost guess what is going on.

The last two lines are the responses from the two services: the first one executes `1+2` and the second one executes `3*4`.

The `null` output that shows up as the first word in the last two lines corresponds to a pattern that is widely used in JavaScript: the error first callback.

Let's explain it with a code example:

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
})

seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
  var product = msg.left * msg.right
  respond( null, { answer: product } )
})

seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2}, function(err, data) {
  if (err) {
    return console.error(err);
  }
  console.log(data);
});
seneca.act({role: 'math', cmd: 'product', left: 3, right: 4}, console.log);
```

The previous code rewrites the first invocation to Seneca with a more appropriate approach. Instead of dumping everything into the console, process the response from Seneca, which is a callback where the first parameter is the error, if one happened (`null` otherwise), and the second parameter is the data coming back from the microservice. This is why, in the first example, `null` was the first output into the console.

In the world of Node.js, it is very common to use callbacks. Callbacks are a way of indicating to the program that something has happened, without being blocked until the result is ready to be processed. Seneca is not an exception to this. It relies heavily on callbacks to process the response to service calls, which makes more sense when you think about microservices being deployed in different machines (in the previous example, everything runs in the same machine), especially because the network latency can be something to factor into the design of your software.

Inversion of control done right

Inversion of control is a must in modern software. It comes together with the dependency injection.

Inversion of control can be defined as *a technique to delegate the creation or call of components and methods so that your module does not need to know how to build the dependencies, which usually, are obtained through the dependency injection*.

Seneca does not really make use of the dependency injection, but it is the perfect example of inversion of control.

Let's take a look at the following code:

```
var seneca = require('seneca')();
seneca.add({component: 'greeter'}, function(msg, respond) {
  respond(null, {message: 'Hello ' + msg.name});
});
seneca.act({component: 'greeter', name: 'David'}, function(error, response) {
  if(error) return console.log(error);
  console.log(response.message);
});
```

This is the most basic Seneca example. From enterprise software's point of view, we can differentiate two components here: a producer (`Seneca.add()`) and a consumer (`Seneca.act()`). As mentioned earlier, Seneca does not have a dependency injection system as is, but Seneca is gracefully built around the inversion of control principle.

In the `Seneca.act()` function, we don't explicitly call the component that holds the business logic; instead of that, we ask Seneca to resolve the component for us through the use of an interface, in this case, a JSON message. This is inversion of control.

Seneca is quite flexible around it: no keywords (except for integrations) and no mandatory fields. It just has a combination of keywords and values that are used by a pattern matching engine called **Patrun**.

Pattern matching in Seneca

Pattern matching is one of the most flexible software patterns that you can use for microservices.

As opposed to network addresses or messages, patterns are fairly easy to extend. Let's explain it with the help of the following example:

```
var seneca = require('seneca')();
seneca.add({cmd: 'wordcount'}, function(msg, respond) {
  var length = msg.phrase.split(' ').length;
  respond(null, {words: length});
});

seneca.act({cmd: 'wordcount', phrase: 'Hello world this is Seneca'},
function(err, response) {
  console.log(response);
});
```

It is a service that counts the number of words in a sentence. As we have seen before, in the first line, we add the handler for the `wordcount` command, and in the second one, we send a request to Seneca to count the number of words in a phrase.

If you execute it, you should get something similar to the following image:



By now, you should be able to understand how it works and even make some modifications to it.

Let's extend the pattern. Now, we want to skip the short words, as follows:

```
var seneca = require('seneca')();

seneca.add({cmd: 'wordcount'}, function(msg, respond) {
  var length = msg.phrase.split(' ').length;
  respond(null, {words: length});
});

seneca.add({cmd: 'wordcount', skipShort: true}, function(msg, respond) {
  var words = msg.phrase.split(' ');
  var validWords = 0;
  for (var i = 0; i < words.length; i++) {
    if (words[i].length > 3) {
      validWords++;
    }
  }
  respond(null, {words: validWords});
});
```

```
seneca.act({cmd: 'wordcount', phrase: 'Hello world this is Seneca'},
function(err, response) {
  console.log(response);
});

seneca.act({cmd: 'wordcount', skipShort: true, phrase: 'Hello world this is
Seneca'}, function(err, response) {
  console.log(response);
});
```

As you can see, we have added another handler for the `wordcount` command with an extra `skipShort` parameter.

This handler now skips all the words with three or fewer characters. If you execute the preceding code, the output is similar to the following image:

```
➜ code  node wordcount.js
2015-11-01T13:50:05.889Z hrzpzs2mgt2n/1446385805876/3897/- INFO hello   Seneca/0.7.2/hrzpzs2mgt2n/1446385805876/3897/-
{ words: 5 }
{ words: 4 }
```

The first line, `{words: 5}`, corresponds to the first act call. The second line, `{words: 4}`, corresponds to the second call.

Patrun – a pattern-matching library

Patrun is also written by Richard Rodger. It is used by Seneca in order to execute the pattern matching and decide which service should respond to the call.

Patrun uses a **closest match** approach to resolve the calls. Let's see the following example:

```
{ x:1,     } -> A
{ x:1, y:1 } -> B
{ x:1, y:2 } -> C
```

In the preceding image, we can see three patterns. These are equivalent to `seneca.add()` from the example in the previous section.

In this case, we are registering three different combinations of *x* and *y* variables. Now, let's see how Patrun does the matching:

- `{x: 1} ->A`: This matches 100% with **A**
- `{x: 2} ->`: No match

- `{x:1, y:1} -> B`: 100% match with **B**; it also matches with **A**, but **B** is a better match—two out of two vs one out of one
- `{x:1, y:2} -> C`: 100% match with **C**; again, it also matches with **A**, but **C** is more concrete
- `{y: 1} ->`: No match

As you can see, Patrun (and Seneca) will always get the longest match. In this way, we can easily extend the functionality of the more abstract patterns by concreting the matching.

Reusing patterns

In the preceding example, in order to skip the words with fewer than three characters, we don't reuse the word count function.

In this case, it is quite hard to reuse the function as is; although the problem sounds very similar, the solution barely overlaps.

However, let's go back to the example where we add two numbers:

```
var seneca = require( 'seneca' )()

seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
  var sum = msg.left + msg.right
  respond(null, {answer: sum})
});

seneca.add({role: 'math', cmd: 'sum', integer: true}, function (msg, respond) {
  this.act({role: 'math', cmd: 'sum', left: Math.floor(msg.left), right:
Math.floor(msg.right)},respond);
});

seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5}, console.log)

seneca.act({role: 'math', cmd: 'sum', left: 1.5, right: 2.5, integer: true},
console.log)
```

As you can see, the code has changed a bit. Now, the pattern that accepts an integer relies on the base pattern to calculate the sum of the numbers.

Patrun always tries to match the closest and most concrete pattern that it can find with the following two dimensions:

- The longest chain of matches
- The order of the patterns

It will always try to find the best fit, and if there is an ambiguity, it will match the first pattern found.

In this way, we can rely on already-existing patterns to build new services.

Writing plugins

Plugins are an important part of applications based on Seneca. As we discussed in Chapter 1, *Microservices Architecture*, the API aggregation is the perfect way of building applications.

Node.js' most popular frameworks are built around this concept: small pieces of software that are combined to create a bigger system.

Seneca is also built around this; `Seneca.add()` principle adds a new piece to the puzzle so that the final API is a mixture of different small software pieces.

Seneca goes one step further and implements an interesting plugin system so that the common functionality can be modularized and abstracted into reusable components.

The following example is the minimal Seneca plugin:

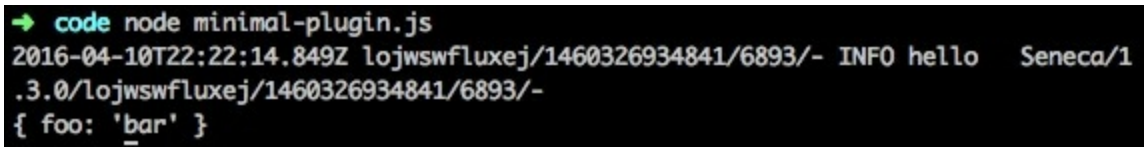
```
function minimal_plugin( options ) {
  console.log(options)
}

require( 'seneca' )()
  .use( minimal_plugin, {foo:'bar'} )
```

Write the code into a `minimal-plugin.js` file and execute it:

```
node minimal-plugin.js
```

The output of this execution should be something similar to the following image:



In Seneca, a plugin is loaded at the startup, but we don't see it as the default log level is INFO. This means that Seneca won't show any DEBUG level info. In order to see what Seneca is doing, we need to get more information, as follows:

```
node minimal-plugin.js -seneca.log.all
```

This produces a huge output. This is pretty much everything that is happening inside Seneca, which can be very useful to debug complicated situations, but in this case, what we want to do is show a list of plugins:

```
node minimal-plugin.js --seneca.log.all | grep plugin | grep DEFINE
```

It will produce something similar to the following image:

```
2015-11-01T20:43:38.969Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  basic        DEFINE  {}
2015-11-01T20:43:39.230Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  transport DEFINE  {}
2015-11-01T20:43:39.388Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  web         DEFINE  {}
2015-11-01T20:43:39.420Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  mem-store DEFINE  {}
2015-11-01T20:43:39.425Z 5kptab6ee6b4/1446410618929/4276/- DEBUG  plugin  minimal_plugin  DEFINE  {foo:bar}
```

Let's analyze the preceding output:

- `basic`: This plugin is included with the main Seneca module and provides a small set of basic utility action patterns.
- `transport`: This is the transport plugin. Up until now, we have only executed different services (quite small and concise) on the same machine, but what if we want to distribute them? This plugin will help us with that, and we will see how to do so in the following sections.
- `web`: In Chapter 1, *Microservices Architecture*, we mentioned that the microservices should advocate to keep the pipes that connect them under a standard that is widely used. Seneca uses TCP by default, but creating a RESTful API can be tricky. This plugin helps to do it, and we will see how to do this in the following section.
- `mem-store`: Seneca comes with a data abstraction layer so that we can handle the data storage in different places: Mongo, SQL databases, and so on. Out of the box, Seneca provides an in-memory storage so that it just works.
- `minimal_plugin`: This is our plugin. So, now we know that Seneca is able to load it.

The plugin we wrote does nothing. Now, it is time to write something useful:

```
function math( options ) {

  this.add({role:'math', cmd: 'sum'}, function( msg, respond ) {
    respond( null, { answer: msg.left + msg.right } )
  })

  this.add({role:'math', cmd: 'product'}, function( msg, respond ) {
    respond( null, { answer: msg.left * msg.right } )
  })

}

require( 'seneca' )()
  .use( math )
  .act( 'role:math,cmd:sum,left:1,right:2', console.log )
```

First of all, notice that in the last instruction, `act()` follows a different format. Instead of passing a dictionary, we pass a string with the same key values as the first argument, as we did with a dictionary. There is nothing wrong with it, but my preferred approach is to use the JSON objects (dictionaries), as it is a way of structuring the data without having syntax problems.

In the previous example, we can see how the code got structured as a plugin. If we execute it, we can

see that the output is similar to the following one:

```
➜ code  node math.js
2016-03-07T20:39:33.145Z kw4uoq06n1xg/1457383173137/2623/- INFO hello   Seneca/1.3.0/kw4uoq06n1xg/1457383173137/2623/-
null { answer: 3 }
```

One of the things you need to be careful about in Seneca is how to initialize your plugins. The function that wraps the plugin (in the preceding example, the `math()` function) is executed synchronously by design and it is called the **definition function**. If you remember from the previous chapter, Node.js apps are single-threaded.

To initialize a plugin, you add a special `init()` action pattern. This action pattern is called in sequence for each plugin. The `init()` function must call its respond callback without errors. If the plugin initialization fails, then Seneca exits the Node.js process. You want your microservices to fail fast (and scream loudly) when there's a problem. All plugins must complete initialization before any actions are executed.

Let's see an example of how to initialize a plugin in the following way:

```
function init(msg, respond) {
  console.log("plugin initialized!");
  console.log("expensive operation taking place now... DONE!");
  respond();
}

function math( options ) {

  this.add({role:'math', cmd: 'sum'}, function( msg, respond ) {
    respond( null, { answer: msg.left + msg.right } )
  })

  this.add({role:'math', cmd: 'product'}, function( msg, respond ) {
    respond( null, { answer: msg.left * msg.right } )
  })

  this.add({init: "math"}, init);
}
require( 'seneca' )()
  .use( math )
  .act( 'role:math,cmd:sum,left:1,right:2', console.log )
```

Then, after executing this file, the output should look very similar to the following image:

```
➜ code  node expensive.js
2016-03-07T20:40:25.351Z bv8phjhz4b92/1457383225343/2640/- INFO hello   Seneca/1.3.0/bv8phjhz4b92/1457383225343/2640/-
plugin initialized!
expensive operation taking place now... DONE!
null { answer: 3 }
```

As you can read from the output, the function that initializes the plugin was called.

Tip

The general rule in Node.js apps is to never block the thread. If you find yourself blocking the thread, you might need to rethink how to avoid it.

Web server integration

In Chapter 1, *Microservices Architecture*, we put a special emphasis on using standard technologies to communicate with your microservices.

Seneca, by default, uses a TCP transport layer that, although it uses TCP, is not easy to interact with, as the criteria to decide the method that gets executed is based on a payload sent from the client.

Let's dive into the most common use case: your service is called by JavaScript on a browser. Although it can be done, it would be much easier if Seneca exposed a REST API instead of the JSON dialog, which is perfect for communication between microservices (unless you have ultra-low latency requirements).

Seneca is not a web framework. It can be defined as a *general purpose microservices framework*, so it would not make too much sense to build it around a concrete case like the one exposed before.

Instead of that, Seneca was built in a way that makes the integration with other frameworks fairly easy.

Express is the first option when building web applications on Node.js. The amount of examples and documentation that can be found on Internet about Express makes the task of learning it fairly easy.

Seneca as Express middleware

Express was also built under the principle of API composition. Every piece of software in Express is called middleware, and they are chained in the code in order to process every request.

In this case, we are going to use **seneca-web** as a middleware for Express so that once we specify the configuration, all the URLs will follow a naming convention.

Let's consider the following example:

```
var seneca = require('seneca')()

seneca.add('role:api,cmd:bazinga',function(args,done){
  done(null,{bar:"Bazinga!"});
});
seneca.act('role:web',{use:{
  prefix: '/my-api',
  pin: {role:'api',cmd:'*'},
```

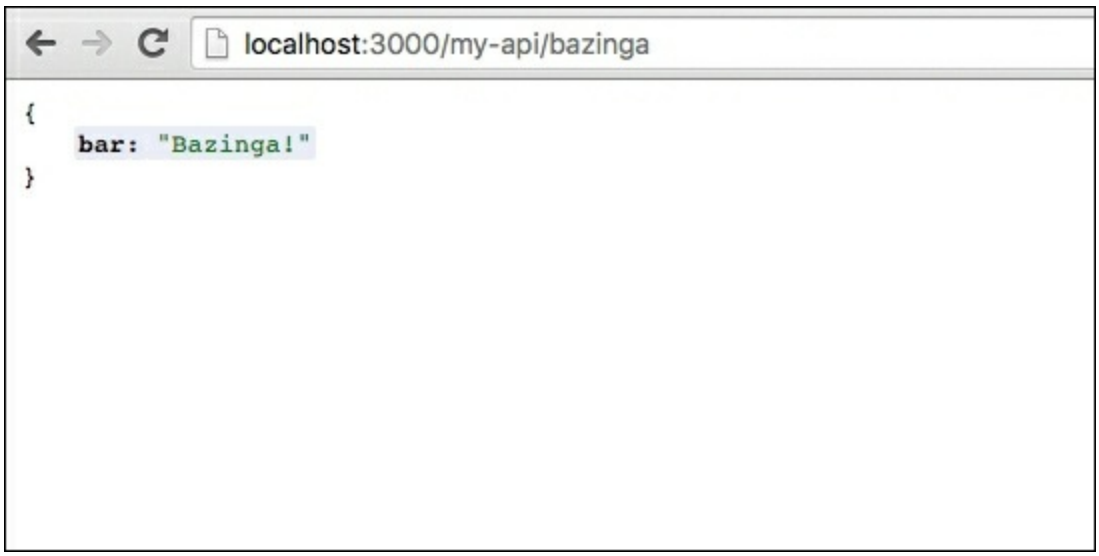
```
  map:{
    bazinga: {GET: true}
  }
}})
var express = require('express')
var app = express()
app.use( seneca.export('web') )
app.listen(3000)
```

This code is not as easy to understand as the previous examples, but I'll do my best to explain it:

- The second line adds a pattern to Seneca. We are pretty familiar with it as all the examples on this book do that.
- The third instruction, `seneca.act()`, is where the magic happens. We are mounting the patterns with the `role:api` pattern and any cmd pattern (`cmd:*`) to react to URLs under `/my-api`. In this example, the first `seneca.add()` will reply to the URL `/my-api/bazinga`, as `/my-api/` is specified by the `prefix` variable and `bazinga` by the `cmd` part of the `seneca.add()` command.
- `app.use(seneca.export('web'))` instructs Express to use seneca-web as middleware to execute actions based on the configuration rules.
- `app.listen(3000)` binds the port `3000` to Express.

If you remember from an earlier section in this chapter, `seneca.act()` takes a function as a second parameter. In this case, we are exposing configuration to be used by Express on how to map the incoming requests to Seneca actions.

Let's test it:



The preceding code is pretty dense, so let's explain it down to the code from the browser:

- Express receives a request that is handled by seneca-web.
- The seneca-web plugin was configured to use `/my-api/` as a prefix, which is being bound with

the keyword `pin` (refer to `seneca.act()` from the preceding code) to Seneca actions (`seneca.add()`) that contain the `role:api` pattern, plus any cmd pattern (`cmd:*`). In this case, `/my-api/bazinga` corresponds to the first (and only) `seneca.add()` command with the `{role: 'api', cmd: 'bazinga'}` pattern.

It takes a while to fully understand the integration between Seneca and Express, but once it is clear, the flexibility offered by the API composability pattern is limitless.

Express itself is big enough to be out of the scope of this book, but it is worth taking a look as it is a very popular framework.

Data storage

Seneca comes with a data-abstraction layer that allows you to interact with the data of your application in a generic way.

By default, Seneca comes with an in-memory plugin (as explained in the previous section), therefore, it works out of the box.

We are going to be using it for the majority of this book, as the different storage systems are completely out of scope and Seneca abstracts us from them.

Seneca provides a simple data abstraction layer (**Object-relational mapping** (**ORM**)) based on the following operations:

- **load**: This loads an entity by identifier
- **save**: This creates or updates (if you provide an identifier) an entity
- **list**: This lists entities matching a simple query
- **remove**: This deletes an entity by an identifier

Let's build a plugin that manages employees in the database:

```
module.exports = function(options) {
  this.add({role: 'employee', cmd: 'add'}, function(msg, respond){
    this.make('employee').data$(msg.data).save$(respond);
  });

  this.find({role: 'employee', cmd: 'get'}, function(msg, respond) {
    this.make('employee').load$(msg.id, respond);
  });
}
```

Remember that the database is, by default, in memory, so we don't need to worry about the table structure for now.

The first command adds an employee to the database. The second command recovers an employee from the database by `id`.

Note that all the ORM primitives in Seneca end up with the dollar symbol (`$`).

As you can see now, we have been abstracted from the data storage details. If the application changes in the future and we decide to use MongoDB as a data storage instead of an in-memory storage, the only thing we need to take care of is the plugin that deals with MongoDB.

Let's use our employee management plugin, as shown in the following code:

```
js
var seneca =
require('seneca')().use(require('seneca-entity)).use('employees-storage')
var employee = {
name: "David",
surname: "Gonzalez",
position: "Software Developer"
}

function add_employee() {
seneca.act({role: 'employee', cmd: 'add', data: employee}, function(err,
msg) {
console.log(msg);
});
}
add_employee();
```

In the preceding example, we add an employee to the in-memory database by invoking the pattern exposed in the plugin.

Along the book, we will see different examples about how to use the data abstraction layer, but the main focus will be on how to build microservices and not how to deal with the different data storages.

PM2 – a task runner for Node.js

PM2 is a production-process manager that helps to scale the Node.js up or down, as well as load balance the instances of the server. It also ensures that the processes are running constantly, tackling down one of the side effects of the thread model of Node.js: an uncaught exception kills the thread, which in turn kills your application.

Single-threaded applications and exceptions

As you learned before, Node.js applications are run in a single thread. This doesn't mean that Node.js is not concurrent, it only means that your application runs on a single thread, but everything else runs parallel.

This has an implication: *if an exception bubbles out without being handled, your application dies*.

The solution for this is making an intensive use of promises libraries such as **bluebird**; it adds handlers for success and failures so that if there is an error, the exception does not bubble out, killing your app.

However, there are some situations that we can't control, *we call them unrecoverable errors or bugs*. Eventually, your application will die due to a badly handled error. In languages such as Java, an exception is not a huge deal: the thread dies, but the application continues working.

In Node.js, it is a big problem. This problem was solved in the first instance using task runners such as **forever**.

Both of them are task runners that, when your application exits for some reason, rerun it again so it ensures the uptime.

Consider the following example:

```
➜ ~  forever helloWorld.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
Server running at http://127.0.0.1:8000/
```

The `helloWorld.js` application is now handled by forever, which will rerun it if the application dies. Let's kill it, as shown in the following image:

```
 4902 ttys000    0:00.33 node /usr/local/bin/forever helloWorld.js
 4903 ttys000    0:00.08 /usr/local/bin/node /Users/dgonzalez/helloWorld.js
```

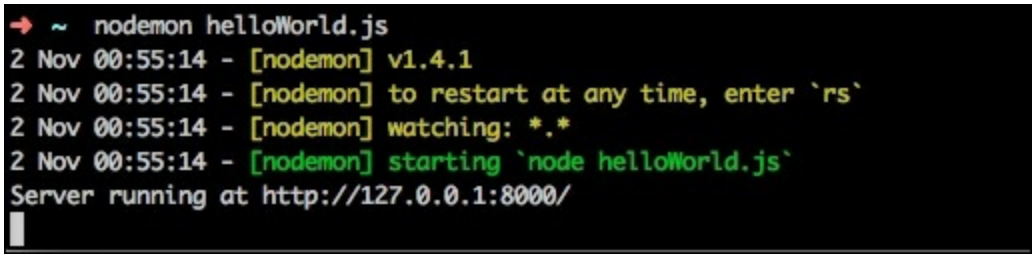
As you can see, forever has spawned a different process with the `4903` PID. Now, we issue a kill command (`kill -9 4093`) and that is the output from forever, as follows:



Although we have killed it, our application was respawned by forever without any downtime (at least, noticeable downtime).

As you can see, forever is pretty basic: it reruns the application as many times as it gets killed.

There is another package called **nodemon**, which is one of the most useful tools for developing Node.js applications. It reloads the application if it detects changes in the files that it monitors (by default, `*.*`):



Now, if we modify the `helloWorld.js` file, we can see how nodemon reloads the application. This is very interesting in order to avoid the edit/reload cycle and speed up the development.

Using PM2 – the industry-standard task runner

Although, forever looks very interesting, PM2 is a more advanced task runner than forever. With PM2, you can completely manage your application life cycle without any downtime, as well as scale your application up or down with a simple command.

PM2 also acts as a load balancer.

Let's consider the following example:

```
var http = require('http');

var server = http.createServer(function (request, response) {
  console.log('called!');
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});
server.listen(8000);
console.log("Server running at http://127.0.0.1:8000/");
```

This is a fairly simple application. Let's run it using PM2:

```
pm2 start helloWorld.js
```

This produces an output similar to the following image:



PM2 has registered an app named `helloWorld`. This app is running in the `fork` mode (that means, PM2 is not acting as a load balancer, it has just forked the app) and the PID of the operating system is `6858`.

Now, as the following screen suggests, we will run `pm2 show 0`, which shows the information relevant to the app with `id 0`, as shown in the following image:

```
➜ ~  pm2 show 0
Describing process with id 0 - name helloWorld
┌──────────────────┬──────────────────────────────────────────────────┐
│ status           │ online                                           │
│ name             │ helloWorld                                       │
│ id               │ 0                                                │
│ path             │ /Users/dgonzalez/helloWorld.js                   │
│ args             │                                                  │
│ exec cwd         │ /Users/dgonzalez                                 │
│ error log path   │ /Users/dgonzalez/.pm2/logs/helloWorld-error-0.log│
│ out log path     │ /Users/dgonzalez/.pm2/logs/helloWorld-out-0.log  │
│ pid path         │ /Users/dgonzalez/.pm2/pids/helloWorld-0.pid      │
│ mode             │ fork_mode                                        │
│ node v8 arguments│                                                  │
│ watch & reload   │ ✘                                                │
│ interpreter      │ node                                             │
│ restarts         │ 0                                                │
│ unstable restarts│ 0                                                │
│ uptime           │ 2m                                               │
│ created at       │ 2015-11-02T01:23:45.434Z                         │
└──────────────────┴──────────────────────────────────────────────────┘
Probes value
┌────────────┬─────────┐
│ Loop delay │ 0.94ms  │
└────────────┴─────────┘
```

With two commands, we have managed to run a simple application in a very sophisticated way.

From now on, PM2 will ensure that your app is always running so that if your application dies, PM2 will restart it again.

We can also monitor the number of apps PM2 is running:

```
pm2 monit
```

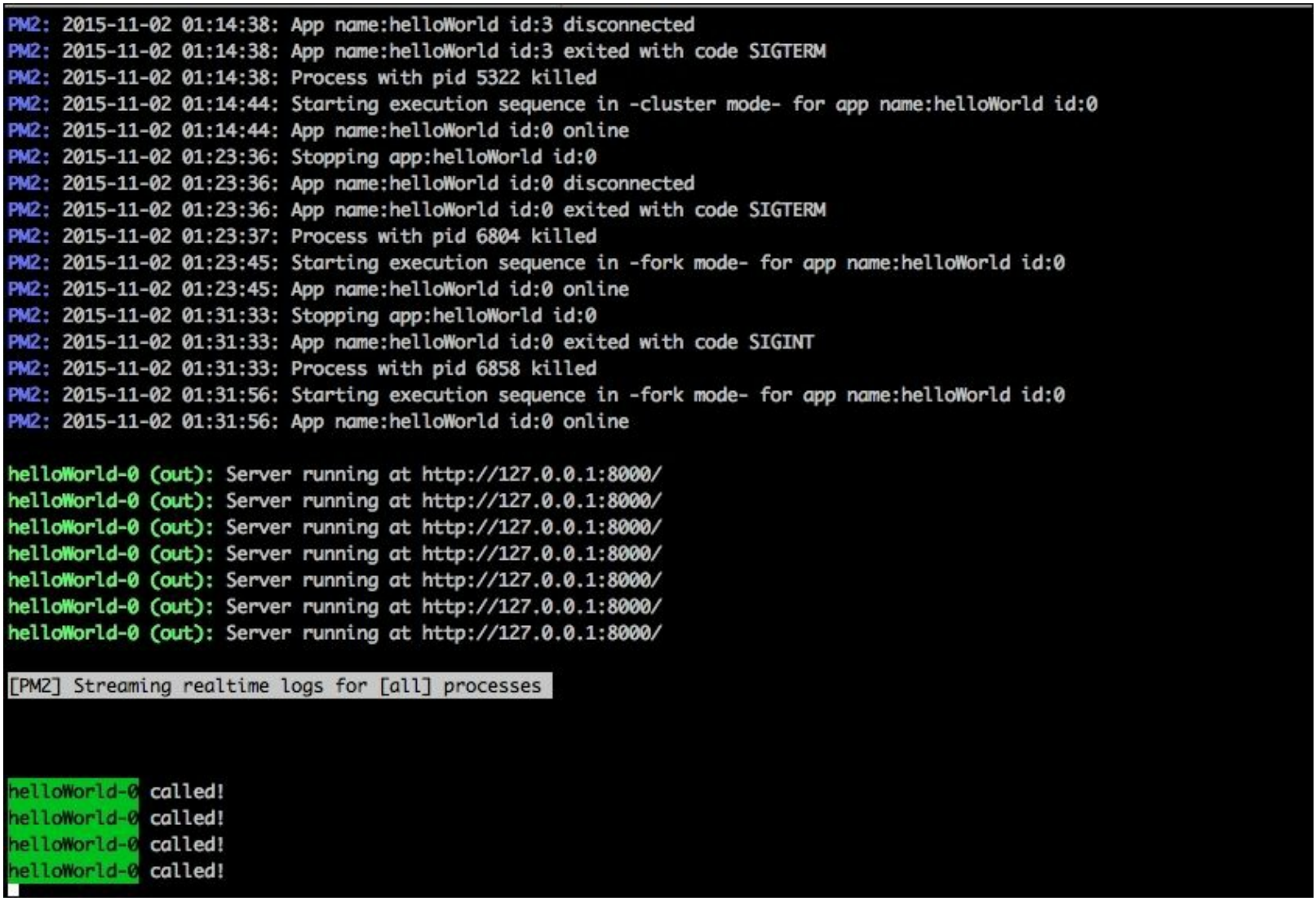
This shows the following output:

```
⊙ PM2 monitoring (To go further check out https://app.keymetrics.io)

 • helloWorld                [                              ] 0 %
[0] [fork_mode]              [|||||                         ] 27.340 MB
```

This is the PM2 monitor. In this case, it is a complete overkill as our system is only composed of one application, which runs in the fork mode.

We can also see the logs executing `pm2 logs` as shown in the following image:



As you can see, PM2 feels solid. With few commands, we have covered 90% of the monitoring necessities of our application. However, this is not everything.

PM2 also comes with an easy way to reload your applications without downtime:

```
pm2 reload all
```

This command ensures that your apps are restarted with zero downtime. PM2 will queue the incoming requests for you and reprocess them once your app is responsive again. There is a more fine-grained option where you can specify reloading only certain apps by specifying the app name:

```
pm2 reload helloWorld
```

For those who have been fighting for years with Apache, NGINX, PHP-FPM, and so on, this will sound very familiar.

Another interesting feature in PM2 is running your application in the cluster mode. In this mode, PM2

spawns a controller process and as many workers (your app) as you specify so that you can take the benefit of multicore CPUs with a single-thread technology such as Node.js.

Before doing this, we need to stop our running application:

```
pm2 stop all
```

This will result in the following output:



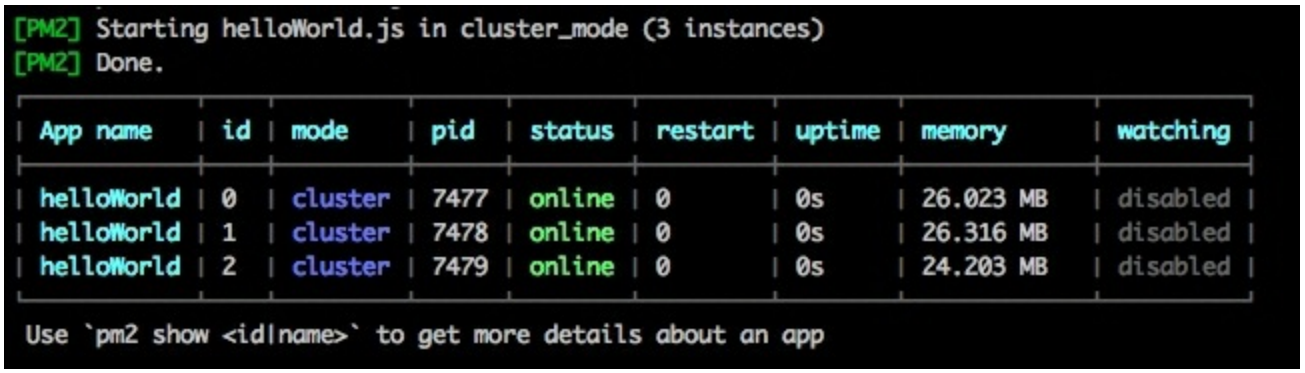
PM2 remembers the apps that were running, so before rerunning the app in the cluster mode, we need to inform PM2 to forget about your app, as follows:

```
pm2 delete all
```



We are ready to run our app in the cluster mode:

```
pm2 start helloWorld.js -i 3
```



PM2 is acting as a round-robin between the main process and the three workers so that they can cope with three requests at the same time. We can also scale down or up our number of workers:

```
pm2 scale helloWorld 2
```

This will result in two processes being run for the same app instead of three:



As you can see, with very little effort, we have managed to configure our app in order to be production ready.

Now, we can save the status of PM2 so that if we restart the server, and PM2 is running as a daemon, the apps will automatically start.

PM2 has a code API so that we can write a Node.js program to manage all the steps that we have been manually doing. It also has a way of configuring your services with a JSON file. We will discuss this in more depth in Chapter 6, *Testing and Documenting Node.js Microservices*, when we study how to use PM2 and Docker to deploy Node.js applications.

Summary

In this chapter, you learned the basics of Seneca and PM2 so that we will be able to build and run a microservices-oriented system in Chapter 4, *Writing Your First Microservice in Node.js*, of this book.

We have also demonstrated that a few of the concepts exposed in the previous chapter are actually helpful in solving real-world problems as well as making our life very easy.

In the next chapter, we will talk about how to split a monolithic application, a task for which we will need to know a few of the concepts developed during this chapter.

Chapter 3. From the Monolith to Microservices

In my professional life, I have worked in quite a few different companies, mainly in financial services, and all of the companies that I have worked for follow the same pattern as shown in the following:

1. A company is set up by a couple of people with good domain knowledge: insurance, payments, credit cards, and so on.
2. The company grows, demanding new business requirements that need to be satisfied quickly (regulation, big customers demanding silly things, and so on), which are built in a hurry with little to no planning.
3. The company experiences another phase of growing, where the business transactions are clearly defined and poorly modelled by a hard-to-maintain monolithic software.
4. The company increases the headcount that drives into growing pains and loss of efficiency due to restrictions imposed on how the software was built in the first instance.

This chapter is not only about how to avoid the previous flow (uncontrolled organic growth), but it is also about how to model a new system using microservices. This chapter is the soul of this book, as I will try to synthetize my experience in a few pages, setting up the principles to be followed in Chapter 4, *Writing Your First Microservice in Node.js*, where we will be building a full system based on microservices using the lessons learned in the previous chapters.

First, there was the monolith

A huge percentage (my estimate is around 90%) of the modern enterprise software is built following a monolithic approach.

Large software components that run in a single container and have a well-defined development life cycle, which goes completely against the agile principles, deliver early and deliver often (https://en.wikipedia.org/wiki/Release_early,_release_often), as follows:

- **Deliver early**: The sooner you fail, the easier it is to recover. If you are working for two years on a software component and then it is released, there is a huge risk of deviation from the original requirements, which are usually wrong and change every few days.
- **Deliver often**: Delivering often, the stakeholders are aware of the progress and can see the changes reflected quickly in the software. Errors can be fixed in a few days and improvements are identified easily.

Companies build big software components instead of smaller ones that work together as it is the natural thing to do, as shown in the following:

1. The developer has a new requirement.
2. He builds a new method on an existing class on the service layer.
3. The method is exposed on the API via HTTP, SOAP, or any other protocol.

Now, multiply it by the number of developers in your company, and you will obtain something called **organic growth**. Organic growth is a type of *uncontrolled and unplanned* growth on software systems under business pressure without an adequate long-term planning, and it is bad.

How to tackle organic growth?

The first thing required to tackle organic growth is to make sure that the business and IT are aligned in the company. Usually, in big companies, IT is not seen as a core part of the business.

Organizations outsource their IT systems, keeping the price in mind, but not the quality so that the partners building these software components are focused on one thing: *deliver on time* and according to the specifications even if they are incorrect.

This produces a less-than-ideal ecosystem to respond to the business needs with a working solution for an existing problem. IT is lead by people who barely understand how the systems are built, and usually overlook the complexity of software development.

Fortunately, this is a changing tendency as IT systems become the driver of 99% of the businesses around the world, but we need to get smarter about how we build them.

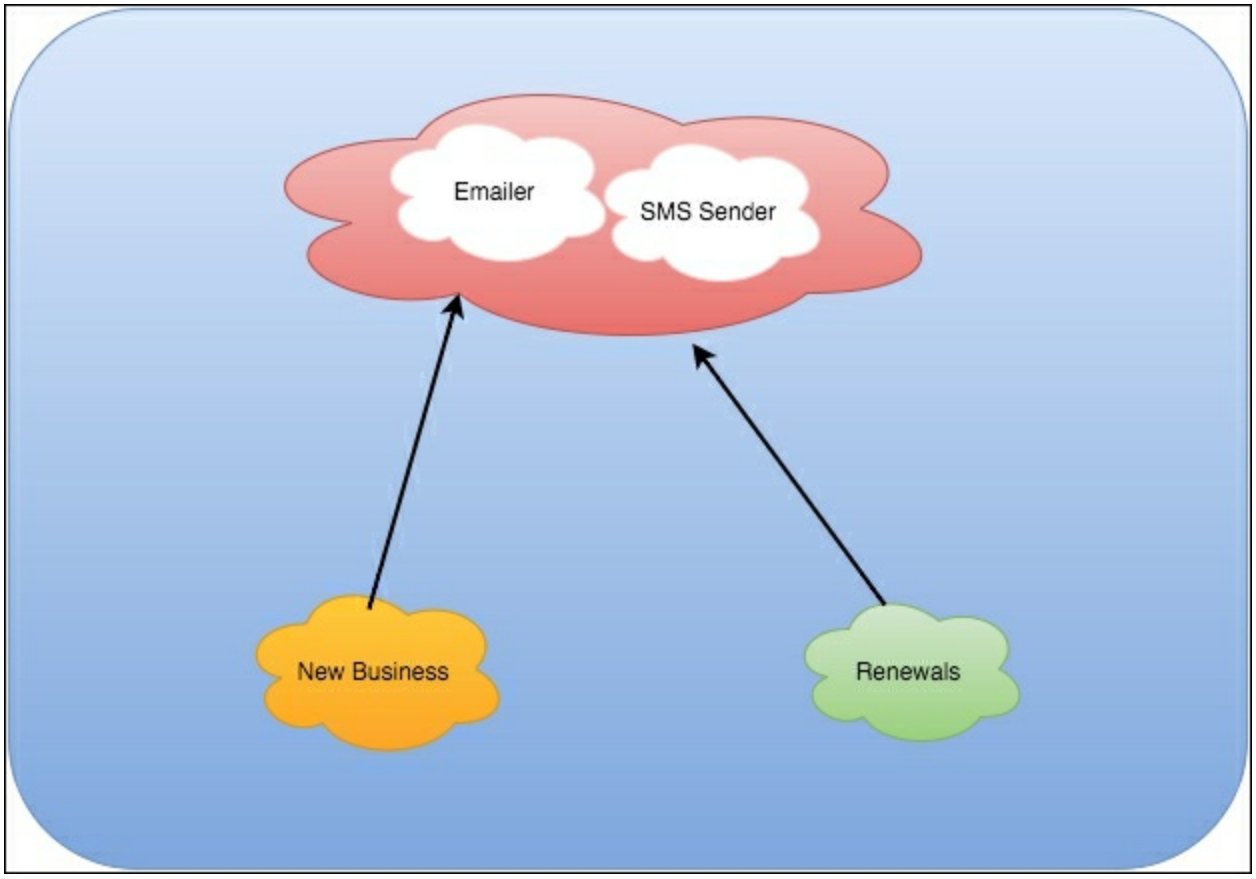
The first measure to tackle organic growth is aligning IT and business stakeholders to work together: educating the nontechnical stakeholders is the key to success.

If we go back to the few big releases schema. Can we do it better?

Of course we can. Divide the work into manageable software artifacts that model a single, well-defined business activity and give it an entity.

It does not need to be a microservice at this stage, but keeping the logic inside of a separated, well-defined, easily testable, and decoupled module will give us a huge advantage for future changes in the application.

Let's consider the following example:



In this insurance system, you can see that someone was in a hurry. SMS and e-mail sender, although both are communication channels, they have a very different nature and you probably want them to act in different ways.

The calling services are grouped into the following two high-level entities:

- **New Business**: The new customers that receive an e-mail when they sign up
- **Renewals**: The existing customers that receive an SMS when the insurance policy is ready to be renewed

At some point, the system needed to send SMSs and e-mails and someone created the communication service entity that handles all the third-party communications.

It looks like a good idea in the beginning. SMS or e-mail, at the end of the day, is only a channel, the communication mechanism will be 90% same and we can reuse plenty of functionality.

What happens if we suddenly want to integrate a third-party service that handles all the physical post?

What happens if we want to add a newsletter that goes out to the customers once a week with information that we consider interesting for our customers?

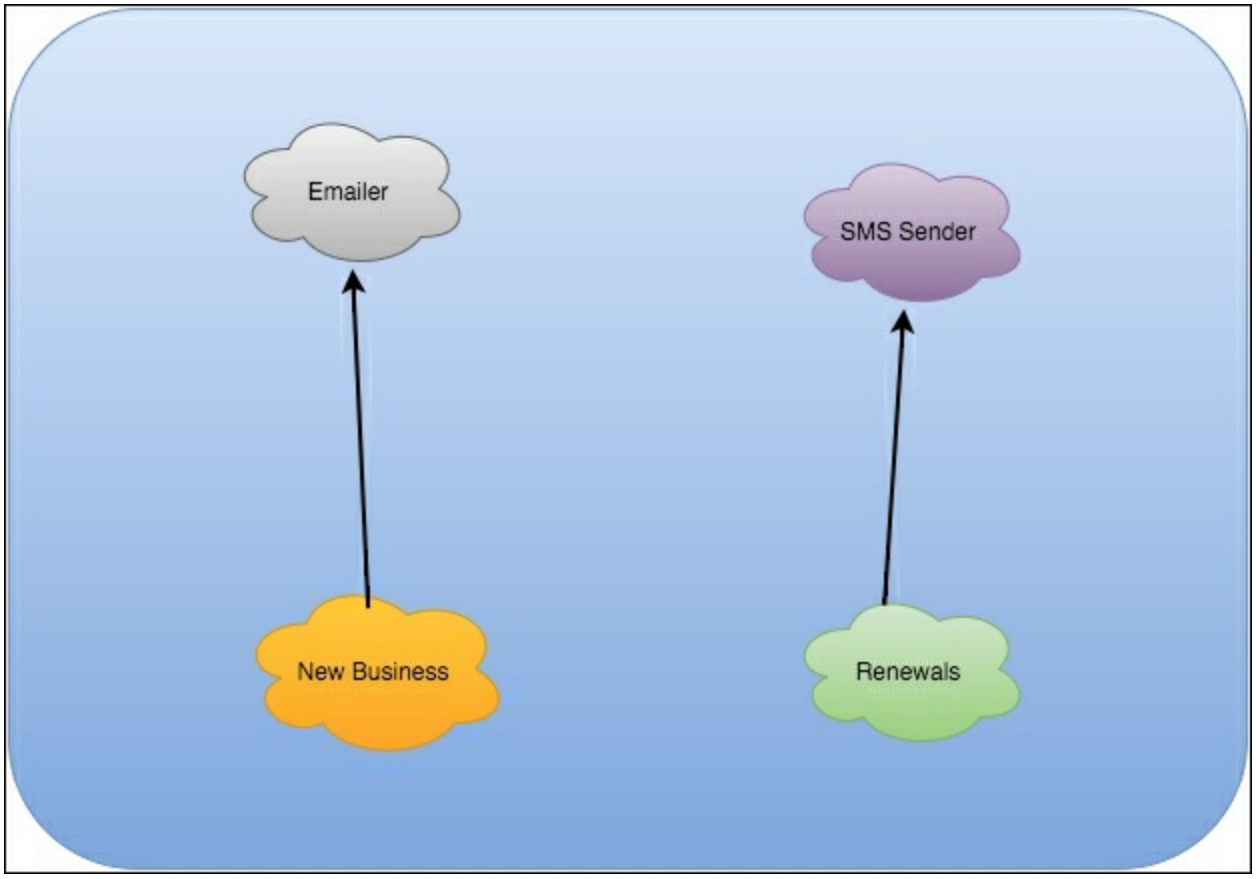
The service will grow out of control and it will become more difficult to test, release, and ensure that the changes in the SMS code won't affect sending the e-mail in any form.

This is organic growth and, in this case, it is related to a law called **Conway's Law**, which states the following:

> *Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.*

In this case, we are falling into a trap. We are trying to model the communication on a single software component that is probably too big and complex to react quickly to new business needs.

Let's take a look at the following diagram:



Now, we have encapsulated every communication channel on its own service (which, later on, will be deployed as a microservice) and we will do the same for future communication channels.

This is the first step to beat organic growth: create fine-grained services with well-defined boundaries and a single responsibility—*do something small, but do it well*.

How abstract is too abstract?

Our brain can't handle complicated mechanisms. The abstraction capacity is one of the most recent human intelligence acquisitions.

In the example from the previous section, I've given something for good, which will upset half of the programmers in the world: *eradicating the abstraction of our system*.

The abstraction capacity is something that we learn along the years and, unlike intelligence, it can be trained. Not everyone can reach the same level of abstraction, and if we mix the specific and complex domain knowledge required by some industries with a high-level of abstraction, we have the perfect recipe for a disaster.

When building software, one of the golden rules that I always tried to follow (and try is the correct word, as I always find huge opposition to it) is to avoid premature abstraction.

How many times did you find yourself in a corner with a simple set of requirements: *build a program to solve X*. However, your team goes off and anticipates all the possible variations of *X*, without even knowing if they are plausible. Then, once the software is in production, one of the stakeholders comes with a variation of *X* that you could have never imagined (as the requirements were not even correct) and now, getting this variation to work will cost you a few days and a massive refactor.

The way to avoid this problem is simple: *avoid abstraction without at least three use cases*.

Do not factor in the possibility of sending the data through different types of channels as it might not happen and you are compromising the current feature with unnecessary abstractions. Once you have at least one other communication channel, it is time to start thinking about how these two software components can be designed better, and when the third use case shows up, refactor.

Remember that when building microservices, they should be small enough to be rewritten on a single sprint (around two weeks), so the benefits of having a working prototype in such a short period of time is worth the risk of having to rewrite it once the requirements are more concrete: something to show to the stakeholders is the quickest way to nail down the requirements.

Seneca is great in this regard as, through pattern matching, we can extend the API of a given microservice without affecting the existing calling code: our service is open for extension, but closed for modification (the SOLID principles) as we are adding functionality without affecting the existing one. We will see more complete examples of this behavior in Chapter 4, *Writing Your First Microservice in Node.js*.

Then the microservices appeared

Microservices are here to stay. Nowadays, the companies give more importance to the quality of the software. As stated in the previous section, deliver early and deliver often are the key to succeed in software development.

Microservices are helping us to satisfy business needs as quickly as possible through modularity and specialization. Small pieces of software that can easily be versioned and upgraded within a few days and they are easy to test as they have a clear and small purpose (specialization) and are written in such a way that they are isolated from the rest of the system (modularization).

Unfortunately, it is not common to find the situation as described previously. Usually, big software systems are not built in a way that modularization or specialization are easy to identify. The general rule is to build a big software component that does everything and the modularization is poor, so we need to start from the very basics.

Let's start by writing some code, as shown in the following:

```
module.exports = function(options) {

  var init = {}

  /**
   * Sends one SMS
   */
  init.sendSMS = function(destination, content) {
    // Code to send SMS
  }

  /**
   * Reads the pending list of SMS.
   */
  init.readPendingSMS = function() {
    // code to receive SMS
    return listOfSms;
  }

  /**
   * Sends an email.
   */
  init.sendEmail = function(subject, content) {
    // code to send emails
  }

  /**
   * Gets a list of pending emails.
   */
  init.readPendingEmails = function() {
    // code to read the pending emails
    return listOfEmails;
```

```
  }

  /**
   * This code marks an email as read so it does not get
   * fetch again by the readPendingEmails function.
   */
  init.markEmailAsRead = function(messageId) {
    // code to mark a message as read.
  }

  /**
   * This function queues a document to be printed and
   * sent by post.
   */
  init.queuePost = function(document) {
    // code to queue post
  }

  return init;
}
```

As you can see, this module can be easily called **communications service** and it will be fairly easy to guess what it is doing. It manages the e-mail, SMS, and post communications.

This is probably too much. This service is deemed to grow out of control, as people will keep adding methods related to communications. This is the key problem of monolithic software: the bounded context spans across different areas, affecting the quality of our software from both functional and maintenance point of view.

If you are a software developer, a red flag will be raised straightaway: the cohesion of this module is quite poor.

Although it could have worked for a while, we are now changing our mindset. We are building small, scalable, and autonomous components that can be isolated. The cohesion in this case is bad as the module is doing too many different things: e-mail, SMS, and post.

What happens if we add another communication channel such as Twitter and Facebook notifications?

The service grows out of control. Instead of having small functional software components, you end up with a gigantic module that will be difficult to refactor, test, and modify. Let's take a look at the following SOLID design principles, explained in Chapter 2, *Microservices in Node.js – Seneca and PM2 Alternatives*:

- **Single-responsibility principle**: The module does too many things.
- **Open for extension, closed for modification**: The module will need to be modified to add new functionalities and probably change the common code.
- **Liskov Substitution**: We will skip this one again.
- **Interface segregation**: We don't have any interface specified in the module, just the implementation of an arbitrary set of functions.

- **Dependency injection:** There is no dependency injection. The module needs to be built by the calling code.

Things get more complicated if we don't have tests.

Therefore, let's split it into various small modules using Seneca.

First, the e-mail module (`email.js`) will be as follows:

```
module.exports = function (options) {

  /**
   * Sends an email.
   */
  this.add({channel: 'email', action: 'send'}, function(msg, respond) {
    // Code to send an email.
    respond(null, {...});
  });

  /**
   * Gets a list of pending emails.
   */
  this.add({channel: 'email', action: 'pending'}, function(msg, respond) {
    // Code to read pending email.
    respond(null, {...});
  });

  /**
   * Marks a message as read.
   */
  this.add({channel: 'email', action: 'read'}, function(msg, respond) {
    // Code to mark a message as read.
    respond(null, {...});
  });
}
```

The SMS module (`sms.js`) will be as follows:

```
module.exports = function (options) {

  /**
   * Sends an email.
   */
  this.add({channel: 'sms', action: 'send'}, function(msg, respond) {
    // Code to send a sms.
    respond(null, {...});
  });

  /**
   * Receives the pending SMS.
   */
  this.add({channel: 'sms', action: 'pending'}, function(msg, respond) {
    // Code to read pending sms.
```

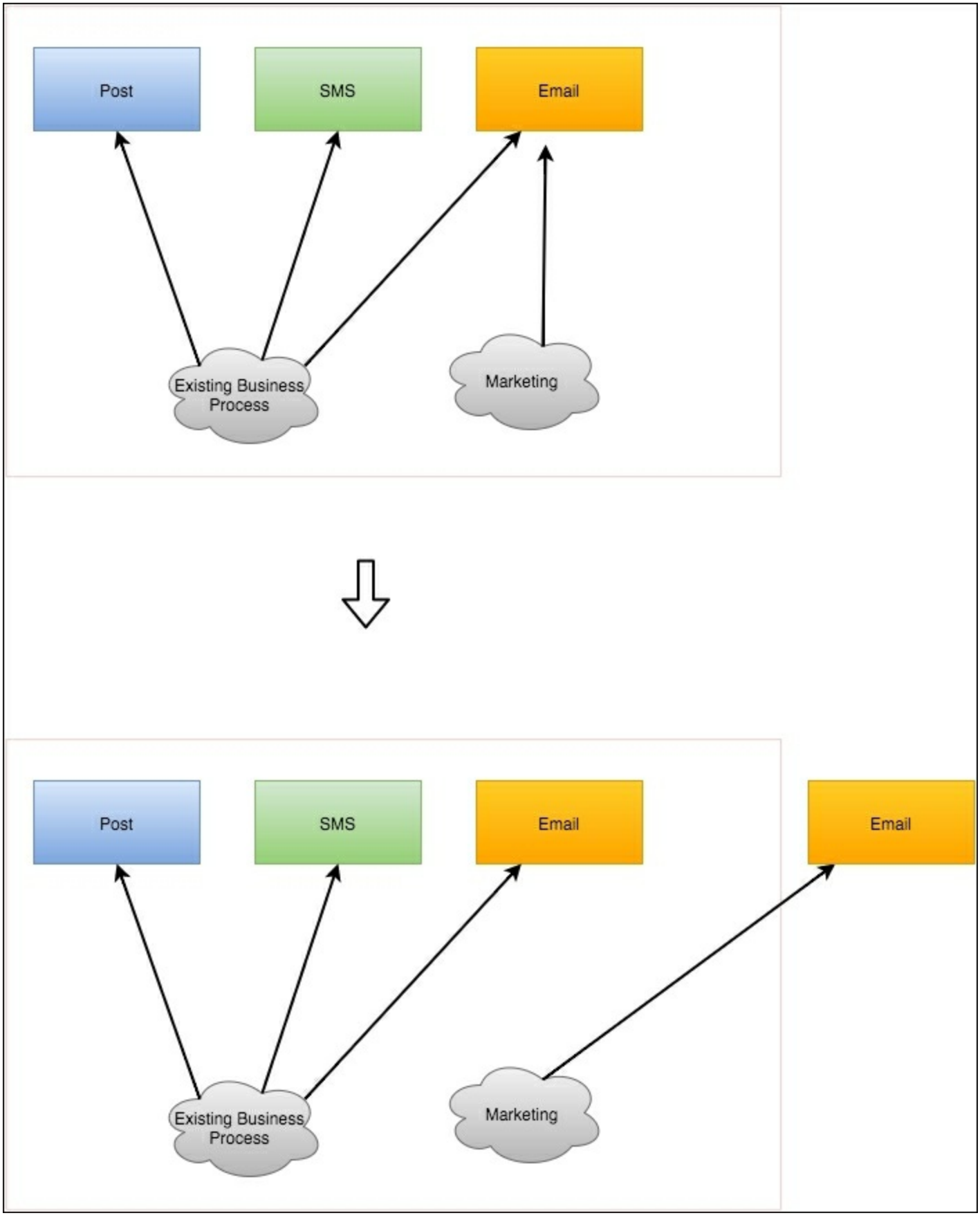
```
    respond(null, {...});
  });
}
```

Finally, the post module (`post.js`) will be as follows:

```
module.exports = function (options) {

  /**
   * Queues a post message for printing and sending.
   */
  this.add({channel: 'post', action: 'queue'}, function(msg, respond) {
    // Code to queue a post message.
    respond(null, {...});
  });
}
```

The following diagram shows the new structure of modules:



Now, we have three modules. Each one of these modules does one specific thing without interfering with each other; we have created high-cohesion modules.

Let's run the preceding code, as follows:

```
var seneca = require("seneca")()
      .use("email")
      .use("sms")
      .use("post");

seneca.listen({port: 1932, host: "10.0.0.7"});
```

As simple as that, we have created a server with the IP `10.0.0.7` bound that listens on the `1932` port for incoming requests. As you can see, we haven't referenced any file, we just referenced the module by name; Seneca will do the rest.

Let's run it and verify that Seneca has loaded the plugins:

```
node index.js --seneca.log.all | grep DEFINE
```

This command will output something similar to the following lines:

```
2015-11-12T22:57:24.720Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin basic      DEFINE {}
2015-11-12T22:57:24.790Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin transport DEFINE {}
2015-11-12T22:57:24.832Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin web       DEFINE {}
2015-11-12T22:57:24.847Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin mem-store DEFINE {}
2015-11-12T22:57:24.860Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin email      DEFINE {}
2015-11-12T22:57:24.872Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin sms       DEFINE {}
2015-11-12T22:57:24.882Z bml3vuxw8qjv/1447369044687/7355/- DEBUG plugin post      DEFINE {}
```

If you remember from Chapter 2, *Microservices in Node.js – Seneca and PM2 Alternatives*, Seneca loads a few plugins by default: `basic`, `transport`, `web`, and `mem-store`, which allow Seneca to work out of the box without being hassled with the configuration. Obviously, as we will see in Chapter 4, *Writing Your First Microservice in Node.js*, that the configuration is necessary as, for example, `mem-store` will only store data in the memory without persisting it between executions.

Aside from the standard plugins, we can see that Seneca has loaded three extra plugins: `email`, `sms`, and `post`, which are the plugins that we have created.

As you can see, the services written in Seneca are quite easy to understand once you know how the framework works. In this case, I have written the code in the form of a plugin so that it can be used by different instances of Seneca on different machines, as Seneca has a transparent transport mechanism that allows us to quickly redeploy and scale parts of our monolithic app as microservices, as follows:

- The new version can be easily tested, as changes on the e-mail functionality will only affect sending the e-mail.
- It is easy to scale. As we will see in the next chapter, replicating a service is as easy as configuring a new server and pointing our Seneca client to it.
- It is also easy to maintain, as the software is easier to understand and modify.

Disadvantages

With microservices, we solve the biggest problems in modern enterprise, but that does not mean that they are problem free. Microservices often lead to different types of problems that are not easy to foresee.

The first and most concerning one is the operational overhead that could chew up the benefits obtained from using microservices. When you are designing a system, you should always have one question in mind: how to automate this? Automation is the key to tackling this problem.

The second disadvantage with microservices is nonuniformity on the applications. A team might consider something a good practice that could be banned in another team (especially around exception handling), which adds an extra layer of isolation between teams that probably does not do well for the communication of your engineers within the team.

Lastly, but not less important, microservices introduce a bigger communication complexity that could lead to security problems. Instead of having to control a single application and its communication with the outer world, we are now facing a number of servers that communicate with each other.

Splitting the monolith

Consider that the marketing department of your company has decided to run an aggressive e-mail campaign that is going to require peaks of capacity that could harm the normal day-to-day process of sending e-mail. Under stress, the e-mails will be delayed and that could cause us problems.

Luckily, we have built our system as explained in the previous section. Small Seneca modules in the form of a high-cohesion and low-coupled plugins.

Then, the solution to achieve it is simple: deploy the e-mail service (`email.js`) on more than one machine:

```
var seneca = require("seneca")().use("email");
seneca.listen({port: 1932, host: "new-email-service-ip"});
```

Also, create a Seneca client pointing to it, as follows:

```
var seneca = require("seneca")()
      .use("email")
      .use("sms")
      .use("post");
seneca.listen({port: 1932, host: "10.0.0.7"});

// interact with the existing email service using "seneca"

var senecaEmail = require("seneca").client({host: "new-email-service-ip", port:
1932});

// interact with the new email service using "senecaEmail"
```

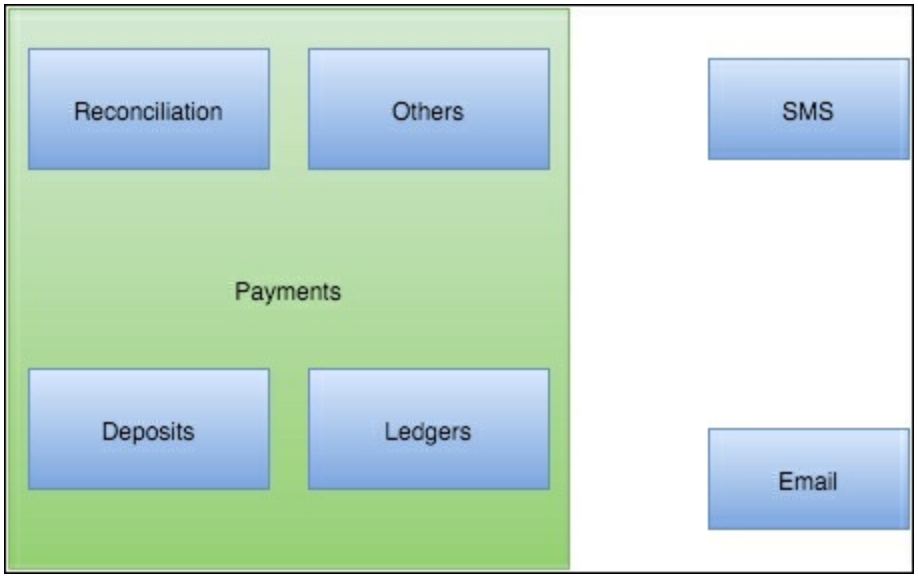
From now on, the `senecaEmail` variable will contact the remote service when calling `act` and we would have achieved our goal: *scale up our first microservice*.

Problems splitting the monolith – it is all about the data

Data storage could be problematic. If your application has grown out of control for a number of years, the database would have done the same, and by now, the organic growth will make it hard to deal with significant changes in the database.

Microservices should look after their own data. Keeping the data local to the service is one of the keys to ensure that the system remains flexible as it evolves, but it might not be always possible. As an example, financial services suffer especially from one of the main weak points of microservices-oriented architectures: *the lack of transactionality*. When a software component deals with money, it needs to ensure that the data remains consistent and not eventually consistent after every single operation. If a customer deposits money in a financial company, the software that holds the account balance needs to be consistent with the money held in the bank, otherwise, the reconciliation of the accounts will fail. Not only that, if your company is a regulated entity, it could cause serious problems for the continuity of the business.

The general rule of thumb, when working with microservices and financial systems, is to keep a not-so-microservice that deals with all the money and creates microservices for the auxiliary modules of the system such as e-mailing, SMS, user registration, and so on, as shown in the following image:



As you can see in the preceding picture, the fact that payments will be a big microservice instead of smaller services, it only has implications in the operational side, there is nothing preventing us from modularizing the application as seen before. The fact that withdrawing money from an ATM has to be an atomic operation (either succeed or fail without intermediate status) should not dictate how we organize the code in our application, allowing us to modularize the services, but spanning the transaction scope across all of them.

Organizational alignment

In a company where the software is built based on microservices, every single stakeholder needs to be involved in decision making.

Microservices are a huge paradigm shift. Usually, large organizations tend to build software in a very old fashioned manner. Big releases every few months that require days to complete the **quality assurance** (**QA**) phase and few hours to deploy.

When a company chooses to implement a microservices-oriented architecture, the methodology changes completely: small teams work on small features that are built, tested, and deployed on their own. The teams do one thing (one microservice, or more realistic, a few of them) and they do it well (they master the domain and technical knowledge required to build the software).

These are what usually called cross-functional teams. A unit of work of few people that have the required knowledge to build high-quality software components.

It is also important to flag that the team has to master the domain knowledge needed to understand the business requirements.

Here is where the majority of the companies where I have worked in my professional life fail (in my opinion). Developers are considered brick stackers that magically understand the business flows without being exposed to them before. If one developer delivers *X* amount of work in one week, ten developers will deliver *10X*. This is wrong.

People in cross-functional teams that build the microservices have to master (not only know) the domain-specific knowledge in order to be efficient and factor the Conway's Law and its implications into the system for changing how the business processes work.

When talking about organizational alignment in microservices, autonomy is the key. The teams need to be autonomous in order to be agile while building the microservices, which implies keeping the technical authority within the team, as follows:

- Languages used
- Code standards
- Patterns used to solve problems
- Tools chosen to build, test, debug, and deploy the software

This is an important part, as this is where we need to define how the company builds software and where the engineering problems may be introduced.

As an example, we can look into the coding standards, as shown in the following list:

- Do we want to keep the same coding standards across the teams?
- Do we want each team to have their own coding standards?

In general, I am always in favor of the 80% rule: *80% of perfection is more than enough for 100% of the use cases*. It means that loosening up the coding standards (it can be applied to other areas) and allowing some level of imperfection/personalization, helps to reduce the friction between teams and also allows the engineers to quickly catch up with the very few important rules to follow such as logging strategies or exception handling.

If your coding standards are too complicated, there will be friction when a team tries to push a code into a microservice out of their usual scope (remember, teams own the services, but every team can contribute to them).

Summary

In this chapter, we discussed the principles of building monolithic applications oriented to be split as microservices, depending on the business needs. As you have learned, the **Atomicity**, **Consistency**, **Isolation**, **Durability** (**ACID**) design principles are concepts that we need to have in mind in order to build high quality software.

You have also learned that we cannot assume that we are going to be able to design a system from scratch, so we need to be smart about how we build the new parts of the system and how we refactor the existing ones so that we achieve the level of flexibility required to satisfy the business needs and be resilient.

We also have given a small introduction about monolithic designed databases and how they are the biggest pain points when splitting a monolithic software into microservices, as it is usually required to shut down the system for a few hours in order to split the data into local databases. This subject could well be an entire book as new trends with NoSQL databases are changing the game of data storage.

Finally, we discussed how to align the teams of engineers in our company in order to be efficient while keeping the flexibility and resilience needed to be able to be agile, as well as how the Conway's Law impacts the conversion of monolithic systems into microservices-oriented architectures.

In the next chapter, we will apply all the principles discussed in the first three chapters, as well as a big dose of common sense to build a full working system based on microservices.

Chapter 4. Writing Your First Microservice in Node.js

We have been learning about how to build robust microservices-oriented software, and now it is time to put all the concepts to practice. In this chapter, we are going to build a microservices-oriented e-commerce using Seneca and some other frameworks that are going to allow us to write a software that will benefit from the particularities of the microservices.

Micromerce – the big picture

It covers the following:

- Writing microservices
- Sizing microservices
- Creating APIs
- Integrating Seneca with Express
- Storing data using Seneca

In this chapter, we are going to write a full (nearly) simplistic e-commerce solution based on microservices. Full means full from the conceptual point of view, but for obvious reasons, it won't be full (as production ready) as it could take us a few books to handle all the possible flows.

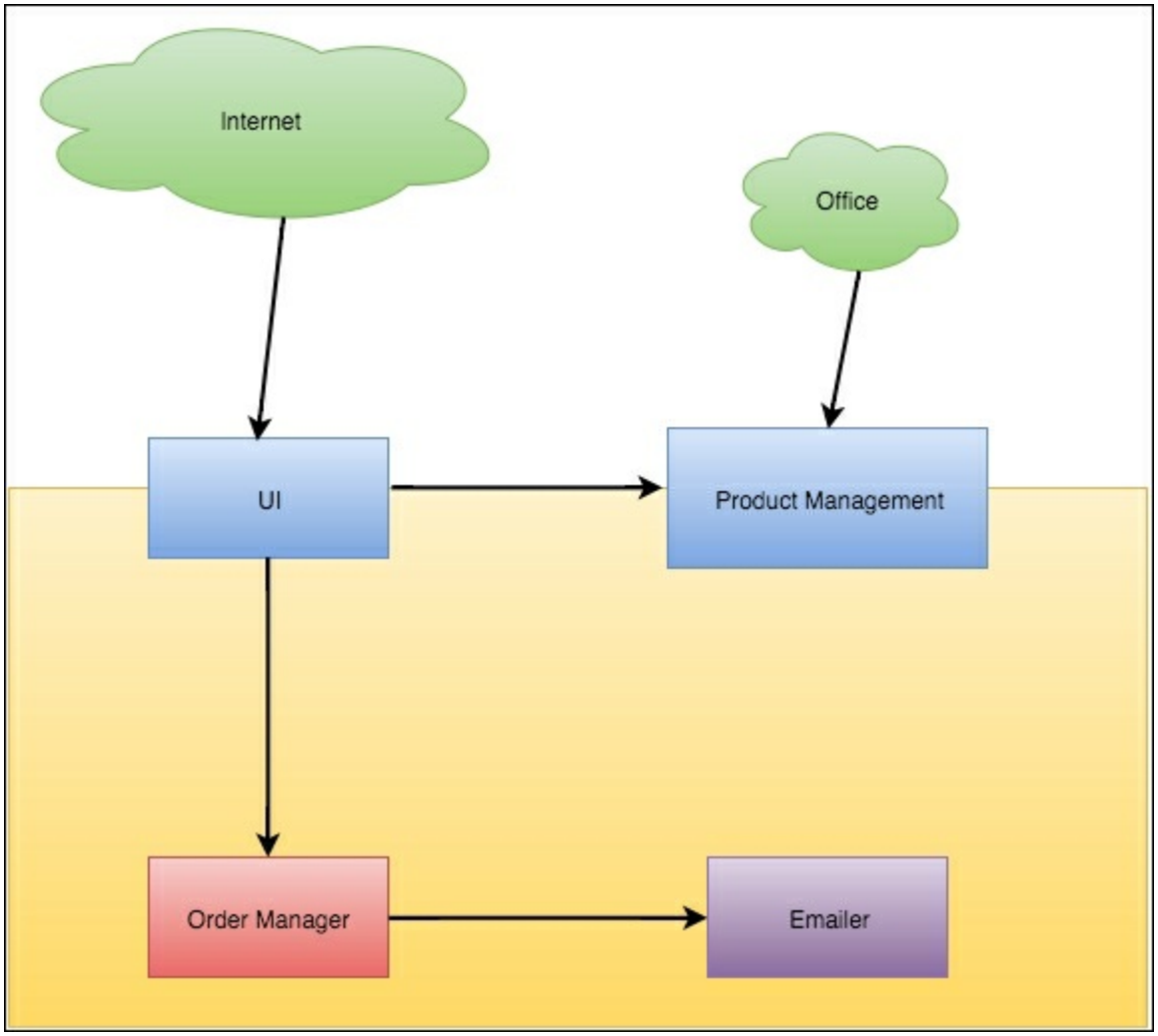
We won't go deep in to the UI, as it is not related to the subject of this book. What we will do instead is a microservice that will aggregate all the other microservices, creating a frontend API to be consumed by a **Single-Page Application** (**SPA**), built with any of the modern JavaScript frameworks.

In this chapter, we are going to develop the following four microservices:

- **Product Manager**: This microservice will be responsible for adding, editing, and removing products from our database, as well as serving products to the customers. This microservice will be partially public for a potential admin site to add/remove products.
- **Order Manager**: This microservice will be responsible for managing the order and billing.
- **Emailer**: This microservice will be responsible for delivering e-mails to the customers.
- **UI**: This microservice will expose the feature from the other microservices to a potential SPA, but we will only build the JSON interface.

Building the four preceding microservices, we will develop the concepts discussed in the previous chapters so that, by the end of this chapter, we will be able to identify the most common pitfalls going forward. Keep in mind that the objective of this book is not converting you into a microservices or Node.js expert, but to give you the tools required to learn by yourself, as well as make you aware of the best design principles and the most common pitfalls.

Let's take a look at the deployment diagram:



This diagram shows how our company (the yellow square) hides some of our microservices from the real world and exposes some others to different networks, as follows:

- **UI** will be exposed to the Internet. Everybody will be able to hit this endpoint.
- **Product Management** will manage the products in our e-commerce. It will have the following two interfaces:
  - A Seneca endpoint from where the UI will extract data
  - A JSON API from where the office of our company will be able to create, update, and delete products
- **Emailer** will be our communication channel with our customers. We will use this microservice to explain the good points of Seneca, and we will also give an example of the eventual consistency and system degradation when a microservice fails.
- **Order Manager**: This microservice will let us handle the orders for our customers. With this microservice, we will discuss how to handle the fact that the data is local to each microservice, instead of being global to the system. You can't just go to the database to recover the product name or price, it needs to be recovered from other microservice.

Tip

As you can see, there is no user or staff management, but with these four microservices, we will be

able to develop the core concepts of microservices architectures. Seneca comes with a very powerful data and transport plugin system that makes it easy to use Seneca with different data storages and transport systems.

For all our microservices, we are going to use MongoDB as the storage. Seneca comes with an out-of-the-box in-memory database plugin that allows you to start coding straightaway, but the storage is transient: it does not persist the data between calls.

Product Manager – the two-faced core

Product Manager is the core of our system. I know what you are thinking: microservices should be small (micro) and distributed (no central point), but you need to set the conceptual centre somewhere, otherwise you will end up with a fragmented system and traceability problems (we will talk about it later).

Building a dual API with Seneca is fairly easy, as it comes with a quite straightforward integration with Express. Express is going to be used to expose some capabilities of the UI such as editing products, adding products, deleting products, and so on. It is a very convenient framework, easy to learn, and it integrates well with Seneca. It is also a de-facto standard on Node.js for web apps, so it makes it easy to find information about the possible problems.

It is going to also have a private part exposed through Seneca TCP (the default plugin in Seneca) so that our internal network of microservices (specifically, the UI) will be able to access the list of products in our catalogue.

Product Manager is going to be small and cohesioned (it will only manage products), as well as scalable, but it will hold all the knowledge required to deal with products in our e-commerce.

First thing we need to do is to define our Product Manager microservice, as follows:

- This is going to have a function to retrieve all the products in the database. This is probably a bad idea in a production system (as it probably would require pagination), but it works for our example.
- This should have one function that fetches all the products for a given category. It is similar to the previous one, it would need pagination in a production-ready system.
- This should have a function to retrieve products by identifier (`id`).
- This should have one function that allows us to add products to the database (in this case MongoDB). This function will use the Seneca data abstraction to decouple our microservice from the storage: we will be able to (in theory) switch Mongo to a different database without too much hassle (in theory again).
- This should have one function to remove products. Again, using Seneca data abstraction.
- This should have one function to edit products.

Our product will be a data structure having four fields: **name**, **category**, **description**, and **price**. As you can see, it is a bit simplistic, but it will help us to understand the complicated world of microservices.

Our Product Management microservice is going to use MongoDB (https://www.mongodb.org/). Mongo is a document-oriented schema-less database that allows an enormous flexibility to store data such as products (that, at the end of the day, are documents). It is also a good choice for Node.js as it stores JSON objects, which is a standard, created for JavaScript (**JSON** stands for **JavaScript Object Notation**), so that looks like the perfect pairing.

There is a lot of useful information on the MongoDB website if you want to learn more about it.

Let's start coding our functions.

Fetching products

To fetch products, we go to the database and dump the full list of products straight to the interface. In this case, we won't create any pagination mechanism, but in general, paginating data is a good practice to avoid database (or applications, but mainly database) performance problems.

Let's see the following code:

```
/**
 * Fetch the list of all the products.
 */
seneca.add({area: "product", action: "fetch"}, function(args, done) {
  var products = this.make("products");
  products.list$({}, done);
});
```

We already have a pattern in Seneca that returns all the data in our database.

The `products.list$()` function will receive the following two parameters:

- The query criteria
- A function that receives an error and result object (remember the error-first callback approach)

Seneca uses the `$` symbol to identify the key functions such as `list$`, `save$`, and so on. Regarding the naming of the properties of your objects, as long as you use alphanumeric identifiers, your naming will be collision free.

We are passing the `done` function from the `seneca.add()` method to the `list$` method. This works as Seneca follows the callback with error-first approach. In other words, we are creating a shortcut for the following code:

```
seneca.add({area: "product", action: "fetch"}, function(args, done) {
  var products = this.make("products");
  products.list$({}, function(err, result) {
    done(err, result);
  });
});
```

Fetching by category

Fetching by category is very similar to fetching the full list of products. The only difference is that now the Seneca action will take a parameter to filter the products by category.

Let's see the code:

```
/**
 * Fetch the list of products by category.
 */
seneca.add({area: "product", action: "fetch", criteria: "byCategory"},
function(args, done) {
  var products = this.make("products");
  products.list$({category: args.category}, done);
});
```

One of the first questions that most advanced developers will now have in their mind is that *isn't this a perfect scenario for an injection attack?* Well, Seneca is smart enough to prevent it, so we don't need to worry about it any more than avoid concatenating strings with user input.

As you can see, the only significant difference is the parameter passed called `category`, which gets delegated into Seneca data abstraction layer that will generate the appropriate query, depending on the storage we use. This is extremely powerful when talking about microservices. If you remember, in the previous chapters, we always talked about coupling as if it was the root of all evils, and now we can assure it is, and Seneca handles it in a very elegant way. In this case, the framework provides a contract that the different storage plugins have to satisfy in order to work. In the preceding example, `list$` is part of this contract. If you use the Seneca storage wisely, switching your microservice over to a new database engine (have you ever been tempted to move a part of your data over MongoDB?) is a matter of configuration.

Fetching by ID

Fetching a product by ID is one of the most necessary methods, and it is also a tricky one. Not tricky from the coding point of view, as shown in the following:

```
/**
 * Fetch a product by id.
 */
seneca.add({area: "product", action: "fetch", criteria: "byId"}, function(args,
done) {
  var product = this.make("products");
  product.load$(args.id, done);
});
```

The tricky part is how `id` is generated. The generation of `id` is one of the contact points with the database. Mongo creates a hash to represent a synthetic ID; whereas, MySQL usually creates an integer that auto-increments to uniquely identify each record. Given that, if we want to switch MongoDB to MySQL in one of our apps, the first problem that we need to solve is how to map a hash that looks something similar to the following into an ordinal number:

```
e777d434a849760a1303b7f9f989e33a
```

In 99% of the cases, this is fine, but we need to be careful, especially when storing IDs as, if you recall from the previous chapters, the data should be local to each microservice, which could imply that changing the data type of the ID of one entity, requires changing the referenced ID in all the other databases.

Adding a product

Adding a product is trivial. We just need to create the data and save it in the database:

```
/**
 * Adds a product.
 */
seneca.add({area: "product", action: "add"}, function(args, done) {
  var products = this.make("products");
  products.category = args.category;
  products.name = args.name;
  products.description = args.description;
  products.category = args.category;
  products.price = args.price
  products.save$(function(err, product) {
    done(err, products.data$(false));
  });
});
```

In this method, we are using a helper from Seneca, `products.data$(false)`. This helper will allow us to retrieve the data of the entity without all the metadata about namespace (zone), entity name, and base name that we are not interested in when the data is returned to the calling method.

Removing a product

The removal of a product is usually done by `id`: We target the specific data that we want to remove by the primary key and then remove it, as follows:

```
/**
 * Removes a product by id.
 */
seneca.add({area: "product", action: "remove"}, function(args, done) {
  var product = this.make("products");
  product.remove$(args.id, function(err) {
  done(err, null);
  });
});
```

In this case, we don't return anything aside from an error if something goes wrong, so the endpoint that calls this action can assume that a non-errored response is a success.

Editing a product

We need to provide an action to edit products. The code for doing that is as follows:

```
/**
 * Edits a product fetching it by id first.
 */
seneca.edit({area: "product", action: "edit"}, function(args, done) {
  seneca.act({area: "product", action: "fetch", criteria: "byId", id: args.id},
function(err, result) {
  result.data$(
  {
    name: args.name,
    category: args.category,
    description: args.description,
    price: args.price
  }
  );
  result.save$(function(err, product){
    done(product.data$(false));
    });
  });
});
```

Here is an interesting scenario. Before editing a product, we need to fetch it by ID, and we have already done that. So, what we are doing here is relying on the already existing action to retrieve a product by ID, copying the data across, and saving it.

This is a nice way for code reuse introduced by Seneca, where you can delegate a call from one action to another and work in the wrapper action with the result.

Wiring everything up

As we agreed earlier, the product manager is going to have two faces: one that will be exposed to other microservices using the Seneca transport over TCP and a second one exposed through Express (a Node.js library to create web apps) in the REST way.

Let's wire everything together:

```
var plugin = function(options) {
  var seneca = this;

  /**
   * Fetch the list of all the products.
   */
  seneca.add({area: "product", action: "fetch"}, function(args, done) {
    var products = this.make("products");
    products.list$({}, done);
  });

  /**
   * Fetch the list of products by category.
   */
  seneca.add({area: "product", action: "fetch", criteria: "byCategory"},
function(args, done) {
    var products = this.make("products");
    products.list$({category: args.category}, done);
  });

  /**
   * Fetch a product by id.
   */
  seneca.add({area: "product", action: "fetch", criteria: "byId"},
function(args, done) {
    var product = this.make("products");
    product.load$(args.id, done);
  });

  /**
   * Adds a product.
   */
  seneca.add({area: "product", action: "add"}, function(args, done) {
    var products = this.make("products");
    products.category = args.category;
    products.name = args.name;
    products.description = args.description;
    products.category = args.category;
    products.price = args.price
    products.save$(function(err, product) {
      done(err, products.data$(false));
    });
  });
```

```
  /**
   * Removes a product by id.
   */
  seneca.add({area: "product", action: "remove"}, function(args, done) {
    var product = this.make("products");
    product.remove$(args.id, function(err) {
      done(err, null);
    });
  });

  /**
   * Edits a product fetching it by id first.
   */
  seneca.add({area: "product", action: "edit"}, function(args, done) {
    seneca.act({area: "product", action: "fetch", criteria: "byId", id:
args.id}, function(err, result) {
      result.data$(
        {
          name: args.name,
          category: args.category,
          description: args.description,
          price: args.price
        }
      );
      result.save$(function(err, product){
        done(err, product.data$(false));
      });
    });
  });
}
module.exports = plugin;

var seneca = require("seneca")();
seneca.use(plugin);
seneca.use("mongo-store", {
  name: "seneca",
  host: "127.0.0.1",
  port: "27017"
});

seneca.ready(function(err){

  seneca.act('role:web',{use:{
    prefix: '/products',
    pin: {area:'product',action:'*'},
    map:{
    fetch: {GET:true},
    edit: {GET:false,POST:true},
    delete: {GET: false, DELETE: true}
    }
  }});
  var express = require('express');
  var app = express();
  app.use(require("body-parser").json());
```

```
  // This is how you integrate Seneca with Express
  app.use( seneca.export('web') );

  app.listen(3000);

});
```

Now let's explain the code:

We have created a Seneca plugin. This plugin can be reused across different microservices. This plugin contains all the definitions of methods needed by our microservice that we have previously described.

The preceding code describes the following two sections:

- The first few lines connect to Mongo. In this case, we are specifying that Mongo is a local database. We are doing that through the use of a plugin called mongo-store —https://github.com/rjrodger/seneca-mongo-store, written by Richard Rodger, the author of Seneca.
- The second part is new to us. It might sound familiar if you have used JQuery before, but basically what the `seneca.ready()` callback is doing is taking care of the fact that Seneca might not have connected to Mongo before the calls start flowing into its API. The `seneca.ready()` callback is where the code for integrating Express with Seneca lives.

The following is the `package.json` configuration of our app:

```
{
  "name": "Product Manager",
  "version": "1.0.0",
  "description": "Product Management sub-system",
  "main": "index.js",
  "keywords": [
    "microservices",
    "products"
  ],
  "author": "David Gonzalez",
  "license": "ISC",
  "dependencies": {
  "body-parser": "^1.14.1",
  "debug": "^2.2.0",
  "express": "^4.13.3",
  "seneca": "^0.8.0",
  "seneca-mongo-store": "^0.2.0",
  "type-is": "^1.6.10"
  }
}
```

Here we control all the libraries needed for our microservice to run, as well as the configuration.

Integrating with Express – how to create a REST API

Integrating with Express is quite straightforward. Let's take a look at the code:

```
seneca.act('role:web',{use:{
  prefix: '/products',
  pin: {area:'product',action:'*'},
  map:{
  fetch: {GET:true},
  edit: {PUT:true},
  delete: {GET: false, DELETE: true}
  }
}});
var express = require('express');
var app = express();
app.use(require("body-parser").json());

// This is how you integrate Seneca with Express
app.use( seneca.export('web') );

app.listen(3000);
```

This code snippet, as we've seen in the preceding section, provides the following three REST endpoints:

```
/products/fetch

/products/edit

/products/delete
```

Let's explain how.

First, what we do is tell Seneca to execute the `role:web` action, indicating the configuration. This configuration specifies to use a `/products` prefix for all the URLs, and it pins the action with a matching `{area: "product", action: "*"}` pattern. This is also new for us, but it is a nice way to specify to Seneca that whatever action it executes in the URL, it will have implicit `area: "product"` of the handler. This means that `/products/fetch` endpoint will correspond to the `{area: 'products', action: 'fetch'}` pattern. This could be a bit difficult, but once you get used to it, it is actually really powerful. It does not force `use` to fully couple our actions with our URLs by conventions.

In the configuration, the attribute map specifies the HTTP actions that can be executed over an endpoint: fetch will allow `GET`, edit will allow `PUT`, and delete will only allow `DELETE`. This way, we can control the semantics of the application.

Everything else is probably familiar to you. Create an Express app and specify using the following two plugins:

- The JSON body parser
- The Seneca web plugin

This is all. Now, if we add a new action to our Seneca list of actions in order to expose it through the API, the only thing that needs to be done is to modify the map attribute to allow HTTP methods.

Although we have built a very simplistic microservice, it captures a big portion of the common patterns that you find when creating a **CRUD** (**Create Read Update Delete**) application. We have also created a small REST API out of a Seneca application with little to no effort. All we need to do now is configure the infrastructure (MongoDB) and we are ready to deploy our microservice.

The e-mailer – a common problem

E-mailing is something that every company needs to do. We need to communicate with our customers in order to send notifications, bills, or registration e-mails.

In the companies where I've worked before, e-mailing always presented a problem such as e-mails not being delivered, or being delivered twice, with the wrong content to the wrong customer, and so on. It looks terrifying that something as simple as sending an e-mail could be this complicated to manage.

In general, e-mail communication is the first candidate to write a microservice. Think about it:

- E-mail does one thing
- E-mail does it well
- E-mail keeps its own data

It is also a good example of how the *Conway's law* kicks into our systems without being noticed. We design our systems modeling the existing communication in our company as we are constrained by it.

How to send e-mails

Back to the basics. How do we send e-mails? I am not talking about which network protocol we use for sending the e-mail or what are the minimum acceptable headers?

I am talking about what we need to send an e-mail from the business point of view:

- A title
- The content
- A destination address

That is everything. We could have gone far, talking about acknowledgements, secure e-mail, BCCs, and so on. However, we are following the lean methodology: start with the minimum viable product and build up from it until you achieve the desired result.

I can't remember a project where the e-mail sending wasn't a controversial part. The product chosen to deliver e-mails ends up tightly coupled to the system and it is really hard to replace it seamlessly. However, microservices are here to rescue us.

Defining the interface

As I mentioned before, although it sounds easy, sending corporate e-mails could end up being a mess. Therefore, the first thing we need to clear is our minimum requirements:

- How do we render the e-mail?
  - Does rendering the email belongs to the bound context of the email manipulation?
  - Do we create another microservice to render e-mails?
  - Do we use a third party to manage the e-mails?
- Do we store the already sent e-mails for auditing purposes?

For this microservice, we are going to use Mandrill. Mandrill is a company that allows us to send corporate e-mails, track the already sent e-mails, and create e-mail templates that can be edited online.

Our microservice is going to look as shown in the following code:

```
var plugin = function(options) {
  var seneca = this;
  /**
   * Sends an email using a template email.
   */
  seneca.add({area: "email", action: "send", template: "*"}, function(args,
done) {
// TODO: More code to come.
  });

  /**
   * Sends an email including the content.
   */
  seneca.add({area: "email", action: "send"}, function(args, done) {
// TODO: More code to come.
  });
};
```

We have two patterns: one that makes use of templates and the other that sends the content contained in the request.

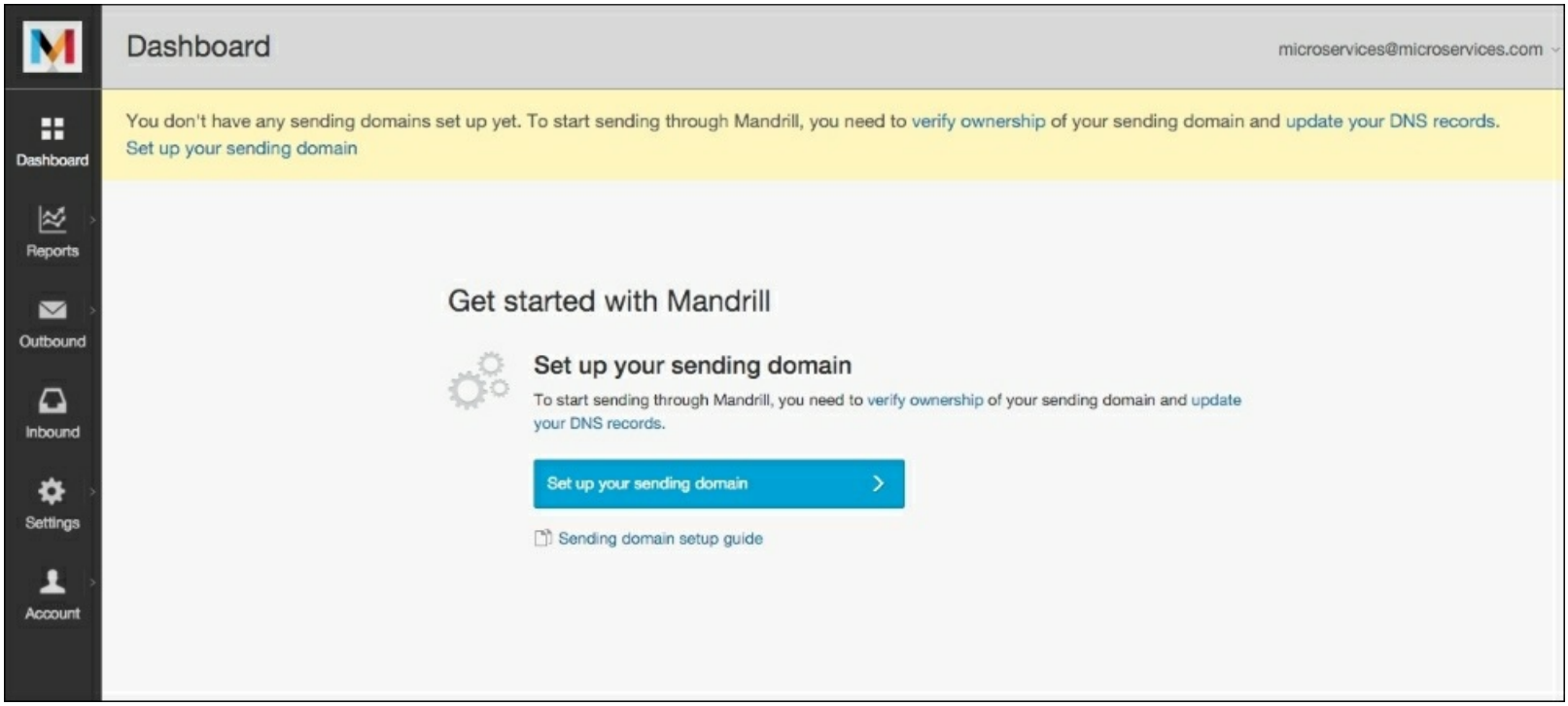
As you can see, everything that we have defined here is information related to e-mailing. There is no bleeding from the Mandrill terminology into what the other microservices see in our e-mail sending. The only compromise that we are making is the templating. We are delegating the template rendering to the e-mail sender, but it is not a big deal, as even if we walk away from Mandrill, we will need to render the content somehow.

We will come back to the code later.

Setting up Mandrill

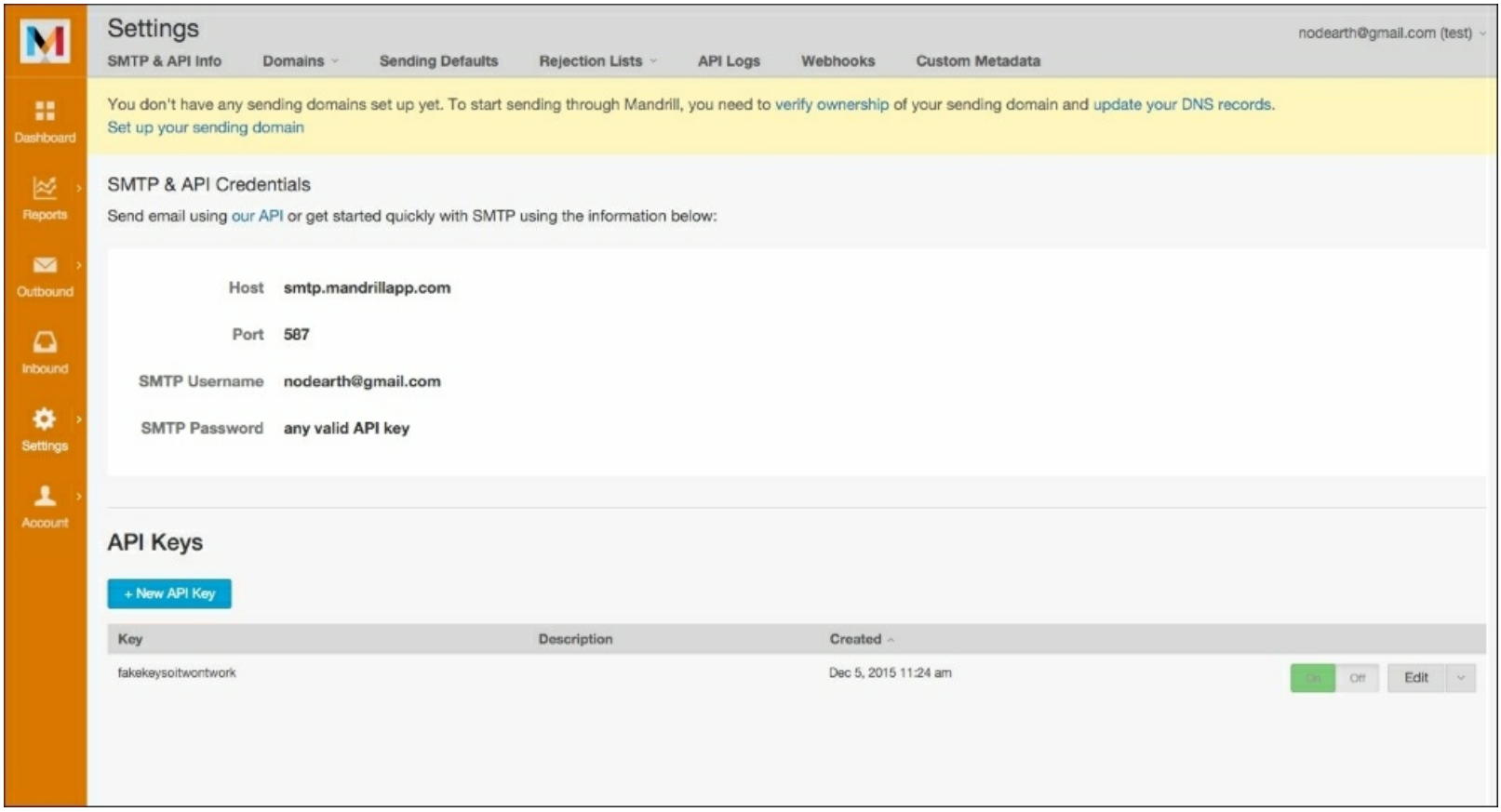
Mandrill is fairly easy to use and shouldn't be a problem to set up. However, we are going to use the test mode so that we can assure that the e-mails are not going to be delivered and we can access the API for all our needs.

The first thing we need to do is create an account on Mandrill. Just register with your e-mail at https://mandrillapp.com, and you should be able to access to it, as shown in the following screenshot:



Now we have created an account that we need to enter into the test mode. In order to do it, just click on your e-mail at the top-right corner and select the **Turn on the test mode** option from the menu. The Mandrill menu on the left will turn orange now.

Next, we need to create an API key. This key is the login information to be used by the Mandrill API. Just click on **Settings** and **SMTP & API Info** and add a new key (don't forget the checkbox to mark the key as test key). It should look like the following screenshot now:



The key is everything you need for now. Let's test the API:

```
var mandrill = require("mandrill-api/mandrill");
var mandrillClient = new mandrill.Mandrill("<YOUR-KEY-HERE>");

mandrillClient.users.info({}, function(result){
  console.log(result);
}, function(e){
  console.log(e);
});
```

With these few lines, we have managed to test that Mandrill is up and running and we have a valid key. The output of this program should be something very similar to the following JSON:

```
{ username: 'youremail@yourdomain.com',
  created_at: '2015-12-05 10:55:59.02874',
  public_id: 'yourpublicid',
  reputation: 33,
  hourly_quota: 25,
  backlog: 0,
  stats:
   { today:
      { sent: 0,
        hard_bounces: 0,
        soft_bounces: 0,
        rejects: 0,
        complaints: 0,
<...continues...>
```

Hands on – integrating Mandrill in your microservice

Everything is ready now. We have a working key and our interface. The only thing left is to create the code. We are going to use a small part of the Mandrill API, but if you want to make use of other features, you can find a better description here: https://mandrillapp.com/api/docs/

Let's take a look at the following code:

```
/**
   * Sends an email including the content.
   */
  seneca.add({area: "email", action: "send"}, function(args, done) {
    console.log(args);
    var message = {
      "html": args.content,
      "subject": args.subject,
      "to": [{
        "email": args.to,
        "name": args.toName,
        "type": "to"
      }],
      "from_email": "info@micromerce.com",
      "from_name": "Micromerce"
    }
    mandrillClient.messages.send({"message": message}, function(result) {
      done(null, {status: result.status});
    }, function(e) {
      done({code: e.name}, null);
    });
  });
```

This first method sends messages without using a template. We just get the HTML content (and a few other parameters) from our application and deliver it through Mandrill.

As you can see, we only have two contact points with the outer world: the parameters passed in and the return of our actions. Both of them have a clear contract that has nothing to do with Mandrill, but what about the data?

At the error, we are returning `e.name`, assuming that it is a code. At some point, someone will end up branching the flow depending on this **error code**. Here, we have something called data coupling; our software components don't depend on the contract, but they do depend on the content sent across.

Now, the question is: how do we fix it? *We can't*. At least not in an easy way. We need to assume that our microservice is not perfect, it has a flaw. If we switch provider for e-mailing, we are going to need to revisit the calling code to check potential couplings.

In the world of software, in every single project that I've worked on before, there was always a big push trying to make the code as generic as possible, trying to guess the future, which usually could be

as bad as assuming that your microservice won't be perfect. There is something that always attracted my attention: we put a large amount of effort in to perfection, but we pretty much ignore the fact that we are going to fail and we do can nothing about it. Software fails often and we need to be prepared for that.

Later, we will see a pattern to factor human nature into the microservices: **the circuit breaker**.

Don't be surprised if Mandrill rejects the e-mails due to the *unsigned* reason. This is due to the fact that they couldn't validate the domain from where we are sending the e-mail (in this case, a dummy domain that does not exist). If we want Mandrill to actually process the e-mails (even though we are in test mode), we just need to verify our domain by adding some configuration to it.

Note

More information can be found in the Mandrill documentation here:

https://mandrillapp.com/api/docs/

The second method to send e-mails is send an e-mail from a template. In this case, Mandrill provides a flexible API:

- It provides per-recipient variables in case we send the e-mail to a list of customers
- It has global variables
- It allows content replacement (we can replace a full section)

For convenience, we are going to just use global variables as we are limited on space in this book.

Let's take a look at the following code:

```
  /**
   * Sends an email using a template email.
   */
  seneca.add({area: "email", action: "send", template: "*"}, function(args,
done) {
    console.log("sending");
    var message = {
      "subject": args.subject,
      "to": [{
        "email": args.to,
        "name": args.toName,
        "type": "to"
      }],
      "from_email": "info@micromerce.com",
      "from_name": "Micromerce",
      "global_merge_vars": args.vars,
    }
    mandrillClient.messages.sendTemplate(
      {"template_name": args.template, "template_content": {}, "message":
message},
    function(result) {
```

```
    done(null, {status: result.status});
  }, function(e) {
    done({code: e.name}, null);
  });
});
```

Now we can create our templates in Mandrill (and let someone else to manage them) and we are able to use them to send e-mails. Again, we are specializing. Our system specializes in sending e-mails and you leave the creation of the e-mails to someone else (maybe someone from the marketing team who knows how to talk to customers).

Let's analyze this microservice:

- **Data is stored locally**: Not really (it is stored in Mandrill), but from the design point of view, it is
- **Our microservice is well cohesioned**: It sends only e-mails; it does one thing, and does it well
- **The size of the microservice is correct**: It can be understood in a few minutes, it does not have unnecessary abstractions and can be rewritten fairly easily

When we talked about the SOLID design principles earlier, we always skipped L, which stands for **Liskov Substitution**. Basically, this means that the software has to be semantically correct. For example, if we write an object-oriented program that handles one abstract class, the program has to be able to handle all the subclasses.

Coming back to Node.js, if our service is able to handle sending a plain e-mail, it should be easy to extend and add capabilities without modifying the existing ones.

Think about it from the day-to-day production operations point of view; if a new feature is added to your system, the last thing you want to do is retest the existing functionalities or even worse, deliver the feature to production, introducing a bug that no one was aware of.

Let's create a use case. We want to send the same e-mail to two recipients. Although Mandrill API allows the calling code to do it, we haven't factored in a potential CC.

Therefore, we are going to add a new action in Seneca that allows us to do it, as follows:

```
/**
 * Sends an email including the content.
 */
seneca.add({area: "email", action: "send", cc: "*"}, function(args, done) {
  var message = {
    "html": args.content,
    "subject": args.subject,
    "to": [{
      "email": args.to,
      "name": args.toName,
      "type": "to"
    },{
      "email": args.cc,
```

```
        "name": args.ccName,
        "type": "cc"
      }],
      "from_email": "info@micromerce.com",
      "from_name": "Micromerce"
    }
    mandrillClient.messages.send({"message": message}, function(result) {
      done(null, {status: result.status});
    }, function(e) {
      done({code: e.name}, null);
    });
  });
```

We have instructed Seneca to take the calls that include `cc` in the list of parameters and send them using a Mandrill CC in the send API. If we want to use it, the following signature of the calling code will change:

```
seneca.act({area: "email", action: "send", subject: "The Subject", to:
"test@test.com", toName: "Test Testingtong"}, function(err, result){
// More code here
});
```

The signature will change to the following code:

```
seneca.act({area: "email", action: "send", subject: "The Subject", to:
"test@test.com", toName: "Test Testingtong", cc: "test2@test.com", ccName: "Test
2"}, function(err, result){
// More code here
});
```

If you remember correctly, the pattern matching tries to match the most concrete input so that if an action matches with more parameters than another one, the call will be directed to it.

Here is where Seneca shines: We can call it **polymorphism of actions**, as we can define different versions of the same action with different parameters that end up doing slightly different things and enabling us to reutilize the code if we are 100% sure that this is the right thing to do (remember, microservices enforce the share-nothing approach: repeating the code might not be as bad as coupling two actions).

Here is the `package.json` for the e-mailer microservice:

```
{
  "name": "emailing",
  "version": "1.0.0",
  "description": "Emailing sub-system",
  "main": "index.js",
  "keywords": [
  "microservices",
  "emailing"
  ],
  "author": "David Gonzalez",
  "license": "ISC",
```

```
  "dependencies": {
  "mandrill-api": "^1.0.45",
  "seneca": "^0.8.0"
  }
}
```

The fallback strategy

When you design a system, usually we think about replaceability of the existing components; for example, when using a persistence technology in Java, we tend to lean towards standards (**JPA**) so that we can replace the underlying implementation without too much effort.

Microservices take the same approach, but they isolate the problem instead of working towards an easy replaceability. If you read the preceding code, inside the Seneca actions, we have done nothing to hide the fact that we are using Mandrill to send the e-mails.

As I mentioned before, e-mailing is something that, although seems simple, always ends up giving problems.

Imagine that we want to replace Mandrill for a plain SMTP server such as Gmail. We don't need to do anything special, we just change the implementation and roll out the new version of our microservice.

The process is as simple as applying the following code:

```
var nodemailer = require('nodemailer');
var seneca = require("seneca")();
var transporter = nodemailer.createTransport({
  service: 'Gmail',
  auth: {
    user: 'info@micromerce.com',
    pass: 'verysecurepassword'
  }
});

/**
 * Sends an email including the content.
 */
seneca.add({area: "email", action: "send"}, function(args, done) {
  var mailOptions = {
    from: 'Micromerce Info <info@micromerce.com>',
    to: args.to,
    subject: args.subject,
    html: args.body
  };
  transporter.sendMail(mailOptions, function(error, info){
    if(error){
      done({code: e}, null);
    }
    done(null, {status: "sent"});
  });
});
```

For the outer world, our simplest version of the e-mail sender is now using SMTP through Gmail to deliver our e-mails.

As we will see later in the book, delivering a new version of the same interface in a microservice network is fairly easy; as long as we respect the interface, the implementation should be irrelevant.

We could even roll out one server with this new version and send some traffic to it in order to validate our implementation without affecting all the customers (in other words, contain the failure).

We have seen how to write an e-mail sender in this section. We have worked through a few examples on how our microservice can be adapted quickly for new requirements as soon as the business requires new capabilities or we decide that our vendor is not good enough to cope with our technical requirements.

The order manager

The order manager is a microservice that processes the orders that the customer places through the UI. As you probably remember, we are not going to create a sophisticated single-page application with a modern visual framework, as it is out of the scope of this book, but we are going to provide the JSON interface in order to be able to build the front end later.

Order manager introduces an interesting problem: this microservice needs access to the information about products, such as name, price, availability, and so on. However, it is stored in the product manager microservice, so how do we do that?

Well, the answer for this question might look simple, but requires a bit of thinking.

Defining the microservice – how to gather non-local data

Our microservice will need to do the following three things:

- Recover orders
- Create orders
- Delete existing orders

When recovering an order, the option is going to be simple. Recover the order by the primary key. We could extend it to recover orders by different criteria, such as price, date, and so on, but we are going to keep it simple as we want to focus on microservices.

When deleting existing orders, the option is also clear: use the ID to delete orders. Again, we could choose a more advanced deletion criteria, but we want to keep it simple.

The problem arises when we are trying to create orders. Creating an order in our small microservice architecture means sending an e-mail to the customer, specifying that we are processing their order, along with the details of the order, as follows:

- Number of products
- Price per product
- Total price
- Order ID (in case the customer needs to troubleshoot problems with the order)

How do we recover the product details?

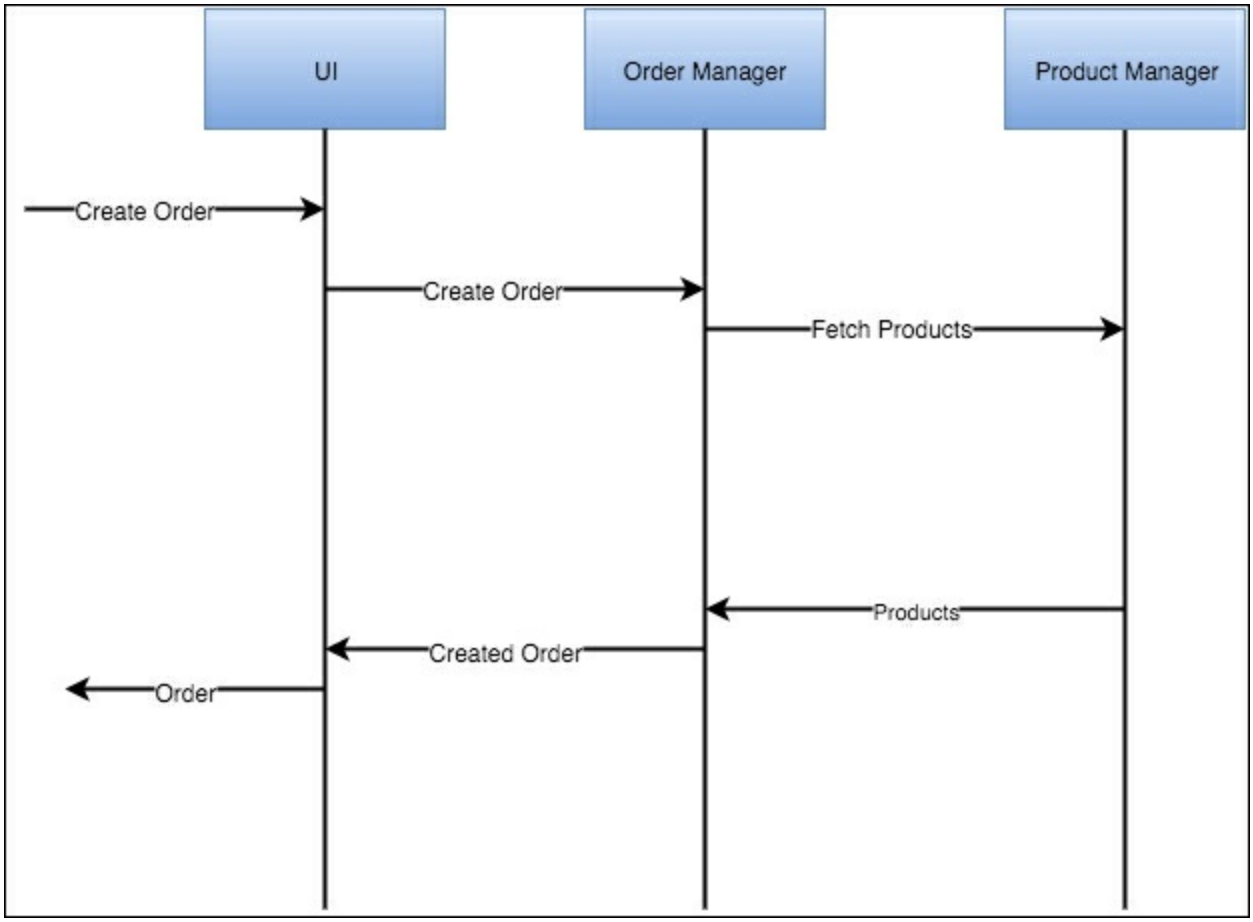
If you see our diagram shown in the *Micromerce – the big picture* section of this chapter, order manager will only be called from the UI, which will be responsible to recover the product name, its price, and so on. We could adopt the following two strategies here:

- Order manager calls product manager and gets the details
- UI calls product manager and delegates the data to the order manager

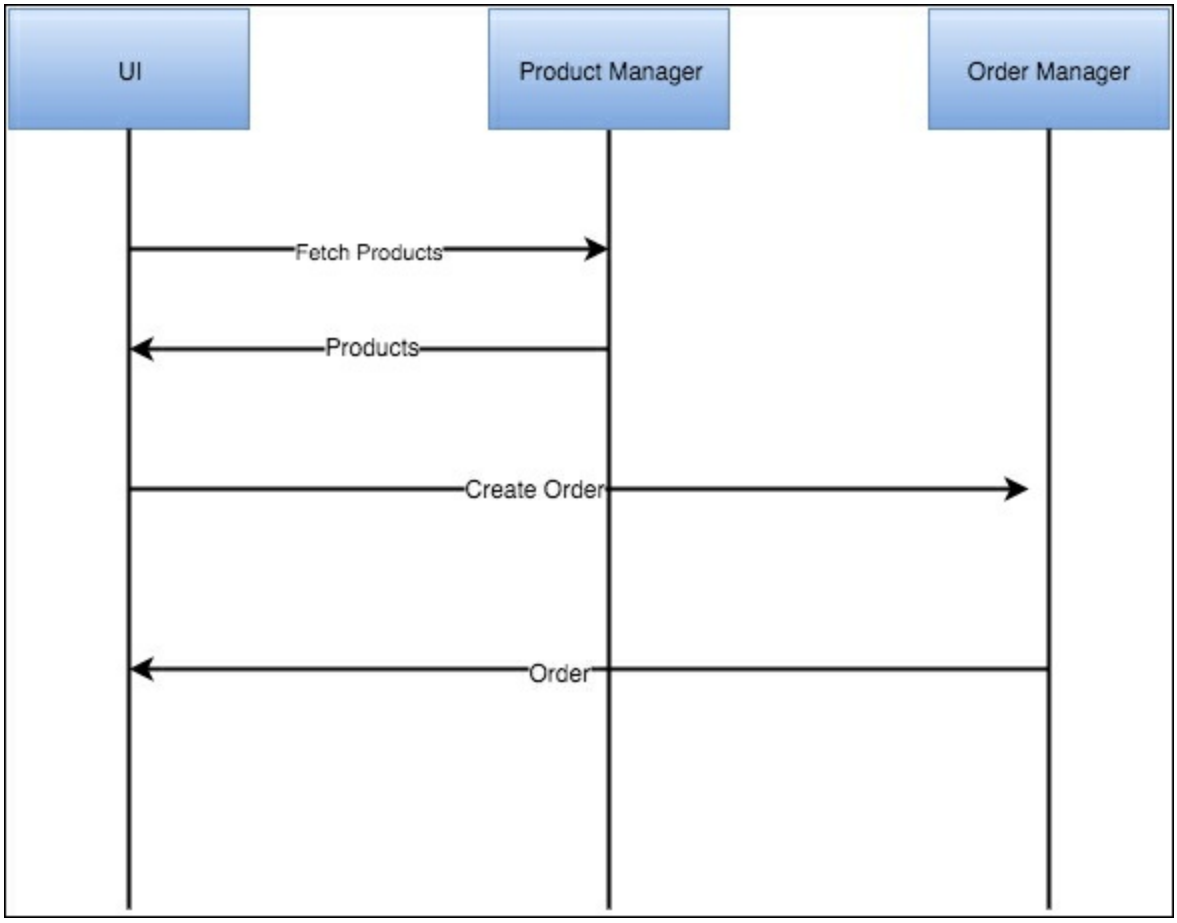
Both options are totally valid, but in this case, we are going for the second: UI will gather the information needed to generate an order and it will only call the order manager when all the data required is available.

Now to answer the question: why?

A simple reason: failure tolerance. Let's take a look at the following sequence diagram of the two options:



The diagram for the second option is shown as follows:



In the first view, there is a big difference: the depth of the call; whereas in the first example, we have two levels of depth (UI calls the order manager, which calls the product manager). In the second example, we have only one level of depth. There are a few immediate effects in our architecture, as follows:

- When something goes wrong, if we only have one level of depth, we don't need to check in too many places.
- We are more resilient. If something goes wrong, it is the UI of the microservice that notices it, returning the appropriate HTTP code, in this case, without having to translate the errors that occurred a few levels above the client-facing microservice.
- It is easier to deploy and test. Not much easier, but we don't need to juggle around, we can see straight away if the product manager is reached from the UI, instead of having to go through the order manager.

The fact that we are using this architecture instead of the two-level depth does not mean that it isn't appropriate for another situation: the network topology is something that you need to plan ahead if you are creating a microservices-oriented architecture, as it is one of the hardest aspects to change.

In some cases, if we want to be extremely flexible, we can use a messaging queue with publisher/subscriber technology where our microservices can subscribe to different types of messages and emit others to be consumed by a different service, but it could complicate the infrastructure that we need to put in place to avoid single point of failures.

The order manager – the code

Let's take a look at the code for the order manager:

```
var plugin = function(options) {
  var seneca = this;

  seneca.add({area: "orders", action: "fetch"}, function(args, done) {
    var orders = this.make("orders");
    orders.list$({id: args.id}, done);
  });

  seneca.add({area: "orders", action: "delete"}, function(args, done) {
    var orders = this.make("orders");
    orders.remove$({id: args.id}, function(err) {
        done(err, null);
    });
  });
}
module.exports = plugin;
```

As you can see, there is nothing complicated about the code. The only interesting point is the missing code from the create action.

Calling remote services

Until now, we have assumed that all our microservices run in the same machine, but that is far from ideal. In the real world, microservices are distributed and we need to use some sort of transport protocol to carry the message from one service to another.

Seneca, as well as nearForm, the company behind Seneca, has sorted this problem for us and the open source community around it.

As a modular system, Seneca has embedded the concept of plugin. By default, Seneca comes with a bundled plugin to use TCP as the protocol, but it is not hard to create a new transport plugin.

Note

While writing this book, I created one by myself: https://github.com/dgonzalez/seneca-nservicebus-transport/

With this plugin, we could route the Seneca messages through NServiceBus (a .NET-based Enterprise Bus), changing the configuration of our client and server.

Let's see how to configure Seneca to point to a different machine:

```
var senecaEmailer = require("seneca")().client({host: "192.168.0.2", port:
8080});
```

By default, Seneca will use the default plugin for transport, which as we've seen in Chapter 2, *Microservices in Node.js – Seneca and PM2 Alternatives*, is `tcp`, and we have specified it to point to the `192.168.0.2` host on the `8080` port.

As simple as that, from now on, when we execute an act command on `senecaEmailer`, the transport will send the message across to the e-mailer and receives the response.

Let's see the rest of the code:

```
seneca.add({area: "orders", action: "create"}, function(args, done) {
  var products = args.products;
  var total = 0.0;
  products.forEach(function(product){
    total += product.price;
  });
  var orders = this.make("orders");
  orders.total = total;
  orders.customer_email = args.email;
  orders.customer_name = args.name;
  orders.save$(function(err, order) {
    var pattern = {
      area: "email",
      action: "send",
      template: "new_order",
      to: args.email,
      toName: args.name,
      vars: {
        // ... vars for rendering the template including the products ...
      }
    }
    senecaEmailer.act(pattern, done);
  });
});
```

As you can see, we are receiving a list of products with all the data needed and passing them to the e-mailer to render the e-mail.

If we change the host where the e-mailer lives, the only change that we need to do here is the configuration of the `senecaEmailer` variable.

Even if we change the nature of the channel (we could potentially even write a plugin to send the data over Twitter, for example), the plugin should look after the particularities of it and be transparent for the application.

Resilience over perfection

In the example from the preceding section, we built a microservice that calls another microservice in order to resolve the call that it receives. However, the following points need to be kept in mind:

- What happens if the e-mailer is down?

- What happens if the configuration is wrong and the e-mailer is not working on the correct port?

We could be throwing *what ifs* for few pages.

Humans are imperfect and so are the things that they build, and software is not an exception. Humans are also bad at recognizing the potential problems in logical flows, and software tends to be a complex system.

In other languages, playing with exceptions is almost something normal, but in JavaScript, exceptions are a big deal:

- If an exception bubbles out in a web app in Java, it kills the current stack of calls and Tomcat (or the container that you use) returns an error to the client
- If an exception bubbles out in a Node.js app, the application is killed as we only have one thread executing the app

As you can see, pretty much every single callback in Node.js has a first parameter that is an error.

When talking about microservices, this error is especially important. You want to be resilient. The fact that an e-mail has failed sending does not mean that the order cannot be processed, but the e-mail could be manually sent later by someone reprocessing the data. This is what we call eventual consistency; we factor into our system the fact that at some point our system is going to crash.

In this case, if there is a problem sending the e-mail, but we could store the order in the database, the calling code, in this case the UI, should have enough information to decide whether the customer gets a fatal message or just a warning:

Your order is ready to be processed, however it might take us two days to send you the e-mail with the order details. Thanks for your patience.

Usually, the fact that our application will keep working even if we cannot complete a request, it is usually more business than technical decision. This is an important detail, as when building microservices, *Conway's law* is pushing us, the technical people, to model the existing business process and partial success maps perfectly to the human nature. If you can't complete a task, create a reminder in Evernote (or a similar tool) and come back to it once the blocker is resolved.

This reads much better than the following:

Something happened about something, but we can't tell you more (which is what my mind reads sometimes when I get a general failure in some websites).

We call this way of handling errors system degradation: it might not be 100% functional, but it will still work even though its few features are not available, instead of a general failure.

If you think for a second, how many times a web service call has rolled back a full transaction in your big corporate system only because it couldn't reach a third-party service that might not even be

important?

In this section, we built a microservice that uses another microservice to resolve a request from a customer: order manager uses e-mailer to complete the request. We have also talked about resilience and how important it is in our architecture in order to provide the best service.

The UI – API aggregation

Until now, we have built independent microservices. They had a specific purpose and dealt with one specific part of our system: e-mail sending, product management, and order processing, but now we are building a microservice whose only purpose is to facilitate the communication between microservices.

Now we are going to build a microservice that interacts with others and is the front-facing façade to the customer.

When I was planning the contents of this chapter, a service like this one wasn't in it. However, after thinking about it, this chapter wouldn't have been the same without showing a few concepts around API aggregation that are fairly easy to show in a frontend microservice.

Need for frontend microservice

Think about scalability. When dealing with HTTP traffic, there is a pyramid of traffic. There are more hits in the frontend than in the backend. Usually, in order to reach the backend, the frontend needs to process the following few requests from the frontend:

- Read a form
- Validate it
- Manage the PRG pattern (https://en.wikipedia.org/wiki/Post/Redirect/Get)

As you can see, there is a lot of logic that needs to be processed by the frontend, so that it is not hard to see capacity problems if the software is busy. If we are using a microservice, and we are using it in the right way, scaling up or down should be an automatic process that can be triggered with a couple of clicks (or commands).

The code

Until now, we have pretty much always tested the code in a single server. This is fine for testing, but when we are building microservices, we want them to be distributed. Therefore, in order to achieve it, we need to indicate to Seneca how to reach the services:

```
var senecaEmailer = require("seneca")().client({
  host: "192.168.0.2",
  port: 8080
});
var senecaProductManager = require("seneca")().client({
  host: "192.168.0.3",
  port: 8080
});
var senecaOrderProcessor = require("seneca")().client({
  host: "192.168.0.4",
  port: 8080
});
```

What we have done is create three Seneca instances. They are like communication pipes between servers.

Let's explain the code:

Seneca, by default, uses the transport plugin TCP. It means that Seneca will be listening to the `/act` URL on the server. As an example, when we create `senecaEmailer`, the URL where Seneca will be pointing to is `http://192.168.0.2:8080/act`.

We can actually verify it with curl. If we execute the following command line, replacing `<valid Seneca pattern>` by a valid Seneca command, we should get a response from the server in the JSON format, which would be the second parameter in the `done` function for the action:

```
curl -d '<valid Seneca pattern>' -v http://192.168.0.2:8080/act
```

Tip

Seneca's default transport plugin is TCP. If we don't specify any other, Seneca will use it to reach other servers and listen to calls.

Let's see an easy example:

```
var seneca = require("seneca")();
seneca.add({cmd: "test"}, function(args, done) {
  done(null, {response: "Hello World!"});
});

seneca.listen({port: 3000});
```

If we run this program, we can see the following output from the terminal:

```
2015-12-14T01:23:48.944Z asrwzxwx2u4e/1450920228931/7488/- INFO hello Seneca/0.8.0/asrwzxwx2u4e/1450920228931/7488/-
2015-12-14T01:23:49.102Z asrwzxwx2u4e/1450920228931/7488/- INFO listen {port:3000}
```

It means that Seneca is listening to the port `3000`. Let's test it:

```
curl -d '{"cmd": "test"}' -v http://127.0.0.1:3000/act
```

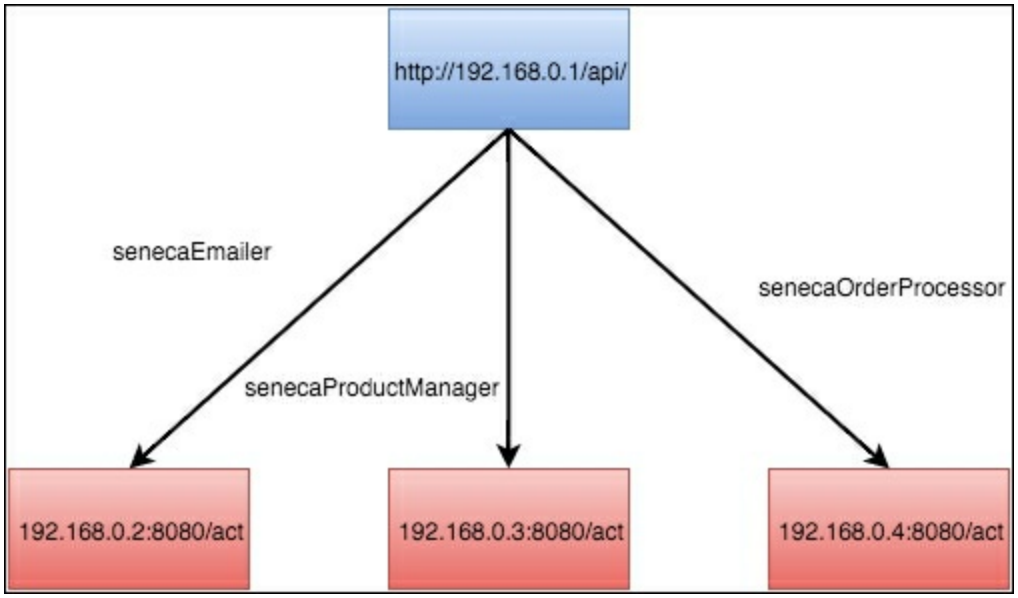
This should print something very similar to the following code in the terminal:

```
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> POST /act HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 15
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 15 out of 15 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json
< Cache-Control: private, max-age=0, no-cache, no-store
< Content-Length: 27
< seneca-id: i45q1ayb0wl1
< seneca-kind: res
< seneca-origin: curl/7.43.0
< seneca-accept: asrwzxwx2u4e/1450920228931/7488/-
< seneca-track:
< seneca-time-client-sent: 0
< seneca-time-listen-recv: 0
< seneca-time-listen-sent: 0
< Date: Thu, 14 Dec 2015 01:26:15 GMT
< Connection: keep-alive
<
* Connection #0 to host 127.0.0.1 left intact
{"response":"Hello World!"}%
```

The preceding code is the TCP/IP dialog between our terminal and Seneca server with the result of the response in the last line.

So, what we achieved earlier on having three different instances of Seneca is configuring our network of microservices; Seneca will transport the messages across the network for us.

The following flow diagram describes how a single API can hide multiple Seneca servers in the backend with different microservices (different Seneca instances, basically):



Now, let's take a look at the skeleton of the microservice:

```
var express = require("express");
var bodyParser = require('body-parser');
var senecaEmailer = require("seneca")().client({
  host: "192.168.0.2",
  port: 8080
});
var senecaProductManager = require("seneca")().client({
  host: "192.168.0.3",
  port: 8080
});
var senecaOrderProcessor = require("seneca")().client({
  host: "192.168.0.4",
  port: 8080
});

function api(options) {
  var seneca = this;

  /**
   * Gets the full list of products
   */
  seneca.add({area: "ui", action: "products"}, function(args, done) {
    // More code to come
  });
  /**
   * Get a product by id
   */
  seneca.add({area: "ui", action: "productbyid"}, function(args, done) {
    // More code to come
  });

  /**
   * Creates an order
```

```
   */
  seneca.add({area: "ui", action: "createorder"}, function(args, done) {
    // More code to come
  });

  this.add("init:api", function(msg, respond){
    seneca.act('role:web',{ use: {
      prefix: '/api',
      pin:  'area:ui,action:*',
      map: {
        products:  {GET:true}
        productbyid: {GET:true, suffix:'/:id'}
        createorder: {POST:true}
      }
    }}, respond)
  });
}
module.exports = api;
var seneca = require("seneca")();
seneca.use(api);

var app = require("express")();
app.use( require("body-parser").json());
app.use(seneca.export("web"));
app.listen(3000);
```

We have actually left the functionality that calls other microservices for later discussion. Now we are going to focus on how the code is articulated:

- We are creating a new plugin. The plugin is called `api` (the name of the function for wrapping the plugin is `api`).
- The plugin has to perform the following three actions:
  - List all the products
  - Get a product by ID
  - Create an order
- These three actions will call to two different microservices: Product Manager and Order Manager. We will come back to this topic later.

Tip

Seneca can be seamlessly integrated with Express in order to provide web capabilities to Seneca microservices.

Until here, everything is well known, but what about the initialization function of the plugin?

At first look, it looks like dark magic:

```
  this.add("init:api", function(msg, respond){
    seneca.act('role:web',{ use: {
      prefix: '/api',
      pin:  'area:ui,action:*',
      map: {
```

```
        products:  {GET:true}
        productbyid: {GET:true, suffix:'/:id'}
        createorder: {POST:true}
      }
    }}, respond)
  });
```

Let's explain it:

1. Seneca will call the `init: <plugin-name>` action in order to initialize the plugin.
2. Through the `prefix` argument, we are listening to URLs under the `/api` path.
3. We are instructing Seneca to map the URLs to action by pinning a base common argument. In this case, all our `seneca.add(..)` contains an argument called `area` with the `ui` value. We are also asking Seneca to route calls that contain the `action` argument (no matter the value, that is why we use the `*`) so that it would ignore calls that don't specify the `action` argument.

The following argument (`map`) specifies the methods allowed in the matching.

How is the argument matching done?

The `area` argument is implicit as we have pinned it with the `ui` value.

The `action` argument needs to be present.

The URL must start with `/api` as we specified a prefix.

So, with this information, `/api/products` will correspond to the `{area: "ui", action: "products"}` action. In the same way, `/api/createorder` will correspond to the `{area: "ui", action:"createorder"}` action.

The `Productbyid` argument is a bit special.

Tip

The Seneca `pin` keyword is used to assume that the calling code has a pair of argument-value so that it makes the code easier to understand, but be careful, implicit values can have bad effects to the readability.

Now, although it is not simple, this looks much easier.

Let's go back to the Seneca actions that are going to provide the functionality:

```
  /**
   * Gets the full list of products.
   */
  seneca.add({area: "ui", action: "products"}, function(args, done) {
    senecaProductManager.act({area: "product", action: "fetch"}, function(err,
result) {
```

```
      done(err, result);
    });
  });

  /**
   * Get a product by id.
   */
  seneca.add({area: "ui", action: "productbyid"}, function(args, done) {
    senecaProductManager.act({area: "product", action: "fetch", criteria:
"byId", id: args.id}, function(err, result) {
      done(err, result);
    });
  });

  /**
   * Creates an order to buy a single prodct.
   */
  seneca.add({area: "ui", action: "createorder"}, function(args, done) {
    senecaProductManager.act({area: "product", action: "fetch", criteria:
"byId", id: args.id}, function(err, product) {
      if(err) done(err, null);
      senecaOrderProcessor.act(area: "orders", action: "create", products:
[product], email: args.email, name: args.name, function(err, order) {
        done(err, order);
      });
    });
  });
```

Note

Warning! In the services written in this chapter, there is no data validation performed in order to make the concepts around the design of microservices clear. You should always validate the incoming data from untrusted systems (such as customers input).

We are actually using everything that we've discussed in the previous chapters, but we are taking a step forward in the Seneca semantics.

We have created an API with a very limited set of functionalities, but through them, we are aggregating the functionality of different microservices into one.

A detail to take into account is the amount of nested calls in the create order action (the last one). In this case, we are creating orders out of only a product to simplify the code, but if we are nesting too many calls for non-blocking actions waiting for the response in a callback, we will end up having a pyramid of code that makes your program difficult to read.

The solution for it would be to refactor how the data is fetched and/or reorganize the anonymous functions, avoiding inlining.

Another solution is the usage of promises libraries such as Q or Bluebird (http://bluebirdjs.com/) that allow us to chain the flow of the methods through promises:

```
myFunction().then(function() {
// Code here
}).then(function(){
// More code here
}).catch(function(error){
// Handle the error.
});
```

In this way, instead of building a sea of callbacks, we are nicely chaining the calls to the methods and adding error handlers to avoid the exceptions from bubbling up.

As you can see, we are using the UI as a central point of communication for all the microservices, except for the mailer, and we have a really good reason for it.

Service degradation – when the failure is not a disaster

Microservices are great, and we have demonstrated that by writing a small system in a few hundred lines of code that is fairly easy to understand.

They are also great as they allow us to react in the event of a failure:

- What happens if the e-mailer microservice stops working?
- What happens if the order processor stops working?
- Can we recover from the situation?
- What does the customer see?

These questions, on a monolithic system, are nonsense. The e-mailer probably would be a part of the application. The failure on sending an e-mail implies a general error, unless it is specifically handled. Same with the order processor.

However, what about our microservices-oriented architecture?

The fact that the e-mailer has failed to deliver a few e-mails does not prevent the orders from being processed, even though the customers aren't getting the e-mails. This is what we call performance or service degradation; the system might be slower, but some functionalities will still work.

Tip

Service degradation is the ability of a system to lose a feature without suffering a general failure.

What about the order manager? Well...we can still make the products-related calls work, but we won't be able to process any order…which might still be a good thing.

The fact that the order manager is responsible for sending the e-mail instead of the UI microservice is not coincidental; we only want to send the e-mail with the acknowledgement of a sale on the success event, and we don't want to send the success e-mail in any other case.

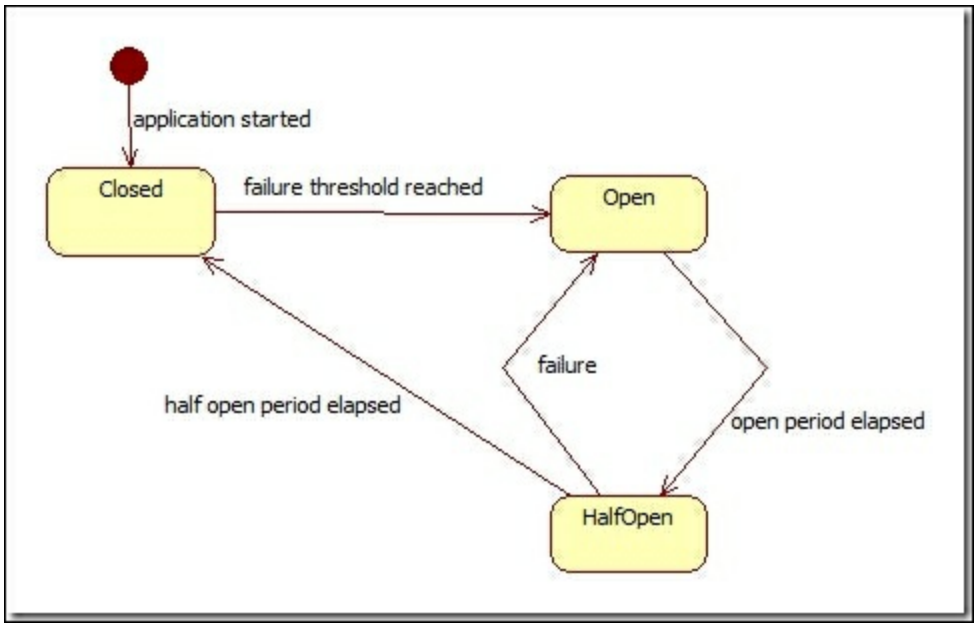
Circuit breakers

In the previous section, we talked about system degradation in the event of a failure, but everybody who has worked in IT for a number of years knows that a system does not fail suddenly in most cases of failures.

The most common event is a timeout; the server is busy for a period of time, which makes the request to fail, giving our customers a terrible user experience.

How can we solve this particular problem?

We can solve this problem with a circuit breaker, as shown in the following image:



A circuit breaker is a mechanism to prevent requests from reaching an unstable server that could cause our application to misbehave.

As you can see in the preceding schema, the circuit breaker has the following three statuses:

- **Closed**: The circuit is closed; the requests reach their destination.
- **Open**: The circuit is open; the requests don't get past the circuit breaker and the client gets an error. The system will retry the communication after a time period.
- **HalfOpen**: The circuit tests the service again, and if there is no error reaching it, the requests can flow again and the circuit breaker is **Closed**.

With this simple mechanism, we can prevent the errors to cascade through our system, avoiding catastrophic failures.

Ideally, the circuit breaker should be asynchronous. This means that even if there are no requests, every few seconds/milliseconds, the system should be trying to re-establish the connection to the

faulty service in order to continue the normal operation.

Tip

Failure is a common denominator in the human nature: better be prepared for it.

Circuit breakers are also an ideal place to alert the support engineers. Depending on the nature of our system, the fact that a given service cannot be reached could mean a serious issue. Can you imagine a bank that is unable to reach the SMS service to send two-factor authentication codes? No matter how hard we try, it will always happen at some point. So, be prepared for it.

Note

There is a very inspiring article from Martin Fowler (one of the big names in microservices) about circuit breakers at http://martinfowler.com/bliki/CircuitBreaker.html.

Seneca – a simple puzzle that makes our lives easier

Seneca is great. It enables the developers to take a simple and small idea and translate it into a piece of code with a connection point that does not make any assumption, just facts. An action has a clear input and provides you the interface to give an answer for it through a callback.

How many times have you found your team struggling with the class structure of an application just to reuse code in *a nice way*?

Seneca focuses on **simplicity**. The fact that we are not modeling objects, but just parts of systems using small portions of code that are extremely cohesive and idempotent to objects makes our life much easier.

Another way how Seneca makes our life easy is through the **plugability**.

If you review the code that we have been writing in this book, the first thing that will be spotted is how convenient the plugins are.

They provide the right level of encapsulation for a bunch of actions (Does it look similar to a class?) that are somehow related to each other.

I always try not to over-engineer solutions. It is really easy to fall into premature abstraction, preparing the code for a future that we don't know whether it it is going to happen in the majority of the cases.

We don't realize how long we spend maintaining features that have been overdesigned and need to be tested every time someone changes the code around them.

Seneca avoids (or at least discourages) this type of designs. Think about Seneca actions as a small piece of paper (like a post-it), where you need to write what happened last week. You need to be smart about what to fit in there, and possibly, split it into another post-it if the content gets to dense.

Another point where Seneca is good is in configurability. As we have seen before, Seneca comes with a number of integrations for data storage and transport.

An important side of Seneca is the transport protocol. As we know by now, the default transport is carried over TCP, but can we use a message queue to do it? The structure is shown as follows:



Yes, we can. It is already done and maintained.

Note

The following URL is a plugin for Seneca that allows it to send messages over RabbitMQ instead of HTTP:

https://github.com/senecajs/seneca-rabbitmq-transport

If you look into the code of the plugin (it looks really complex, but it is not), you can spot where the magic happens in few seconds:

```
  seneca.add({role: 'transport', hook: 'listen', type: 'rabbitmq'},
hook_listen_rabbitmq)
  seneca.add({role: 'transport', hook: 'client', type: 'rabbitmq'},
hook_client_rabbitmq)
```

Seneca is using Seneca actions to delegate the transport of the message. Although it looks a bit recursive, it is brilliant!

Once you understand how Seneca and the transport protocol chosen work, you are immediately qualified to write a transport plugin for Seneca.

Note

When I started learning about Seneca in order to write this book, I also wrote a transport plugin to use NServiceBus (http://particular.net/).

NServiceBus is an interesting idea, it allows you to connect a number of storages and AMPQ-compliant systems and use them as clients. For example, we could be writing messages in a SQL Server table and consuming them from a queue once they get routed through NServiceBus, having immediate auditing capabilities on the history of the messages.

With such flexibility, we could potentially write a plugin that uses pretty much anything as a transport protocol.

Seneca and promises

All our code from the previous chapters is relying on callbacks. Callbacks are good as far as your code does not nest them on more than three levels.

However, there is an even better way of managing the asynchronous nature of JavaScript: **promises**.

Take a look at the following code:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>promise demo</title>
<script src="https://code.jquery.com/jquery-1.10.2.js"></script>
</head>
<body>

<button>Go</button>
<p>Ready...</p>
<div></div>
<div></div>
<div></div>
<div></div>

<script>
var effect = function() {
  return $( "div" ).fadeIn( 800 ).delay( 1200 ).fadeOut();
};

$( "button" ).on( "click", function() {
  $( "p" ).append( " Started... " );

  $.when( effect() ).done(function() {
  $( "p" ).append( " Finished! " );
  });
});
</script>

</body>
</html>
```

The preceding code is an example of the JQuery fragment using promises.

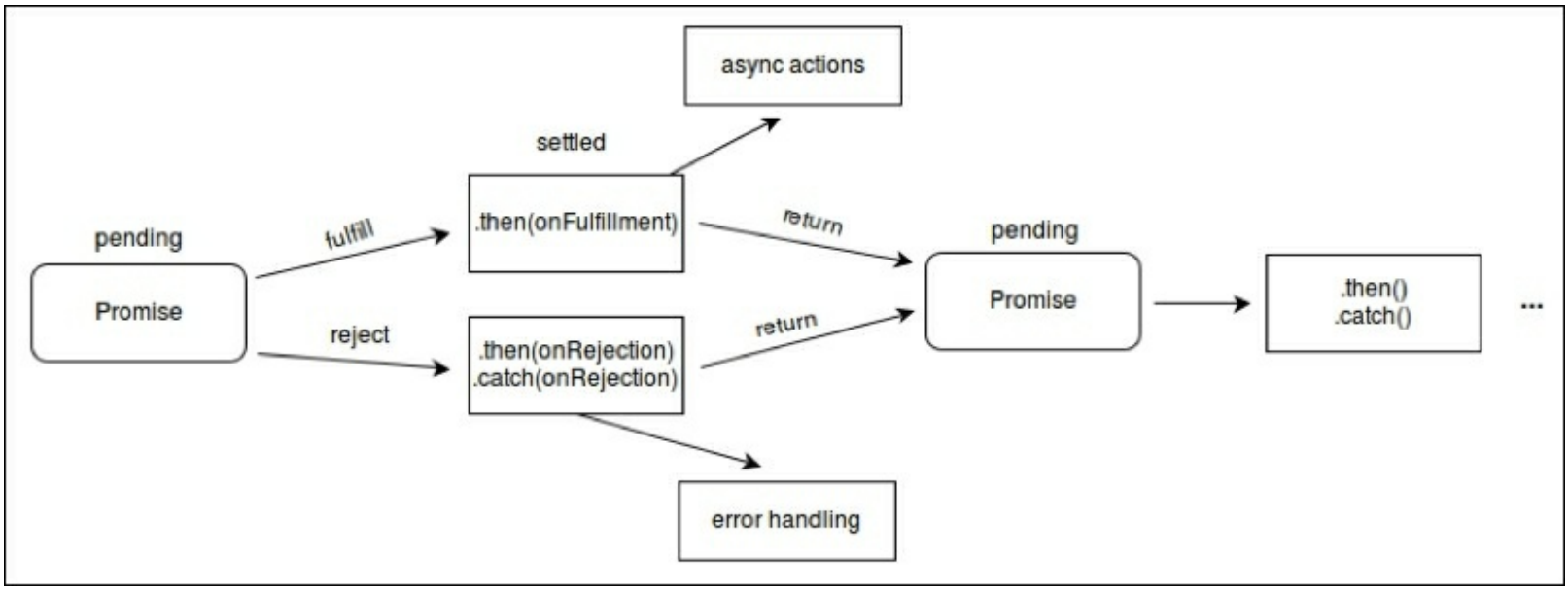
A promise, by its definition is:

A declaration or assurance that one will do something or that a particular thing will happen.

That is exactly it. If you see the preceding code, `$.when`, returns a promise. We don't know how long the effect function will take, but we can guarantee that once it is ready, the function inside of `done` will be executed. It looks very similar to callbacks, but take a look at the following code:

```
callhttp(url1, data1).then(function(result1) {
   // result1 is available here
   return callhttp(url2, data2);
}).then(function(result2) {
   // only result2 is available here
   return callhttp(url3, data3);
}).then(function(result3) {
   // all three are done now, final result is in result3
});
```

Don't try to execute it, it is just a hypothetical example, but what we are doing in there is chain promises; and that makes the code vertical instead of ending up in a pyramid-shaped program, which is a lot harder to read, as shown in the following diagram:



Seneca, by default, is not a promise-oriented framework, but (there is always a but) using Bluebird, one of the most famous promises libraries in JavaScript, we can *promisify* Seneca, as follows:

```
var Promise = require('bluebird');
var seneca = require('seneca')();

// Promisify the .act() method; to learn more about this technique see:
// http://bluebirdjs.com/docs/features.html#promisification-on-steroids
var act = Promise.promisify(seneca.act, seneca);

// Return no error and a success message to illustrate a resolved promise
seneca.add({cmd: 'resolve'}, function (args, done) {
  done(null, {message: "Yay, I've been resolved!"});
});

// Return an error to force a rejected promise
seneca.add({cmd: 'reject'}, function (args, done) {
  done(new Error("D'oh! I've been rejected."));
});
```

```
// Use the new promisified act() with no callback
act({cmd: 'resolve'})
  .then(function (result) {
  // result will be {message: "Yay, I've been resolved!"} since
  // its guaranteed to resolve
  })
  .catch(function (err) {
  // Catch any error as usual if it was rejected
  });

act({cmd: 'reject'})
  .then(function (result) {
  // Never reaches here since we throw an error on purpose
  })
  .catch(function (err) {
  // err will be set with message "D'oh! I've been rejected."
  });
```

There are two important details in the preceding code:

```
var act = Promise.promisify(seneca.act, seneca);
```

This creates a promisified version of the `act` function and its use, as follows:

```
act({cmd: 'reject'})
  .then(function (result) {
  // Never reaches here since we throw an error on purpose
  })
  .catch(function (err) {
  // err will be set with message "D'oh! I've been rejected."
  });
```

An important detail in this last fragment; instead of receiving a callback with the following two parameters:

- An error
- The results

We are chaining the following two methods:

- **Then**: This is executed when the promise is resolved
- **Catch**: This is executed if there is an error while resolving the promise

This type of constructions allows us to write the following code:

```
act({cmd: 'timeout'})
  .then(function (result) {
  // Never reaches here since the gate executer times out
  })
  .catch(function (err) {
  // err will be set with a timeout error thrown by the gate executer
  });
```

This code is handling something that we have never talked about before: the gate executor timeouts. It happens when Seneca cannot reach the destination in some situations, and it can be easily handled with a promise as shown earlier. The `then` part would never be executed as the function will only be called when there is an error.

There are a few well-consolidated options in the market now for promises in JavaScript. Nowadays, my preferred choice would be Bluebird (https://github.com/petkaantonov/bluebird) because of its simplicity. Q is another option used by AngularJS (one of the most popular **SPA** frameworks), but for day-to-day use, it looks more complicated than Bluebird.

Debugging

Debugging a Node.js application is very similar to debugging any other application. IDEs like **WebStorm** or **IntelliJ** provide a traditional debugger where you can install breakpoints and stop the execution whenever the application hits the given line.

This is perfect if you buy a license for one of the IDEs, but there is a free alternative that will have a very similiar result for the users of Google Chrome, **node-inspector**.

Node-inspector is an npm package that pretty much enables the Chrome debugger to debug Node.js applications.

Let's see how it works:

1. First of all, we need to install node-inspector:

   ```
   npm install -g node-inspector
   ```

   This should add a command to our system called `node-inspector`. If we execute it, we get the following output:

   That means our debug server has started.
2. Now we need to run a node application with a special flag to indicate that it needs to be debugged.

   Let's take a simple Seneca act as an example:

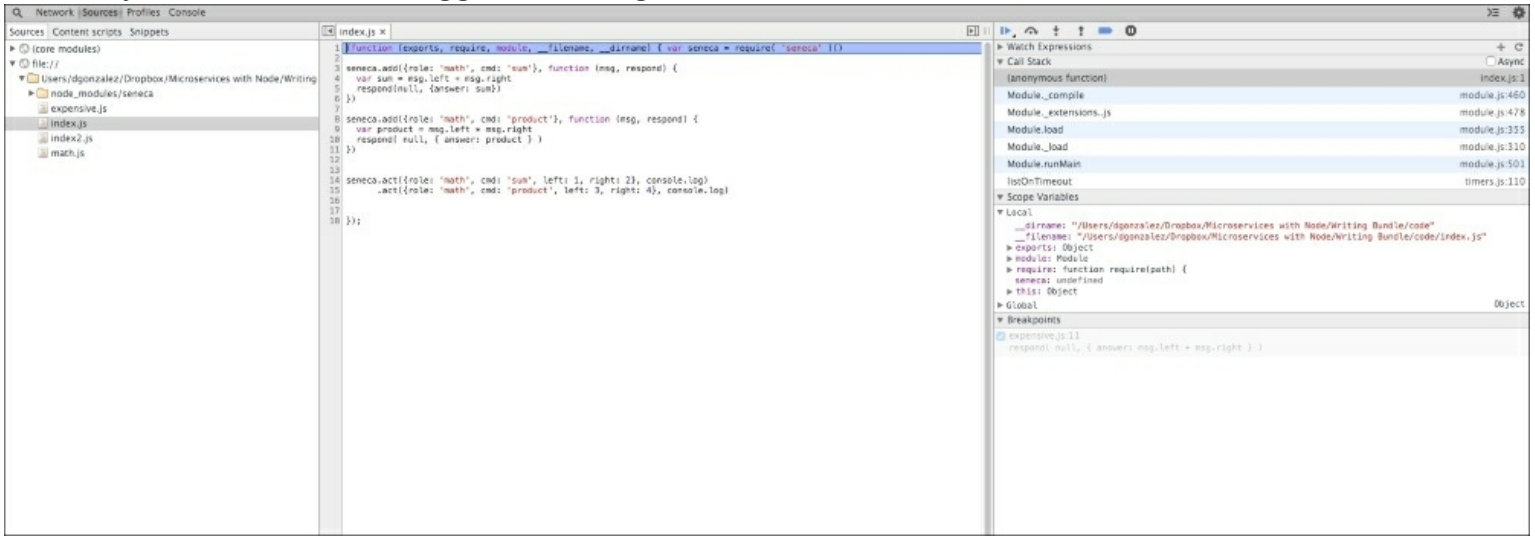
   ```
   var seneca = require( 'seneca' )()
   seneca.add({role: 'math', cmd: 'sum'}, function (msg, respond) {
     var sum = msg.left + msg.right
     respond(null, {answer: sum})
   })

   seneca.add({role: 'math', cmd: 'product'}, function (msg, respond) {
     var product = msg.left * msg.right
     respond( null, { answer: product } )
   })

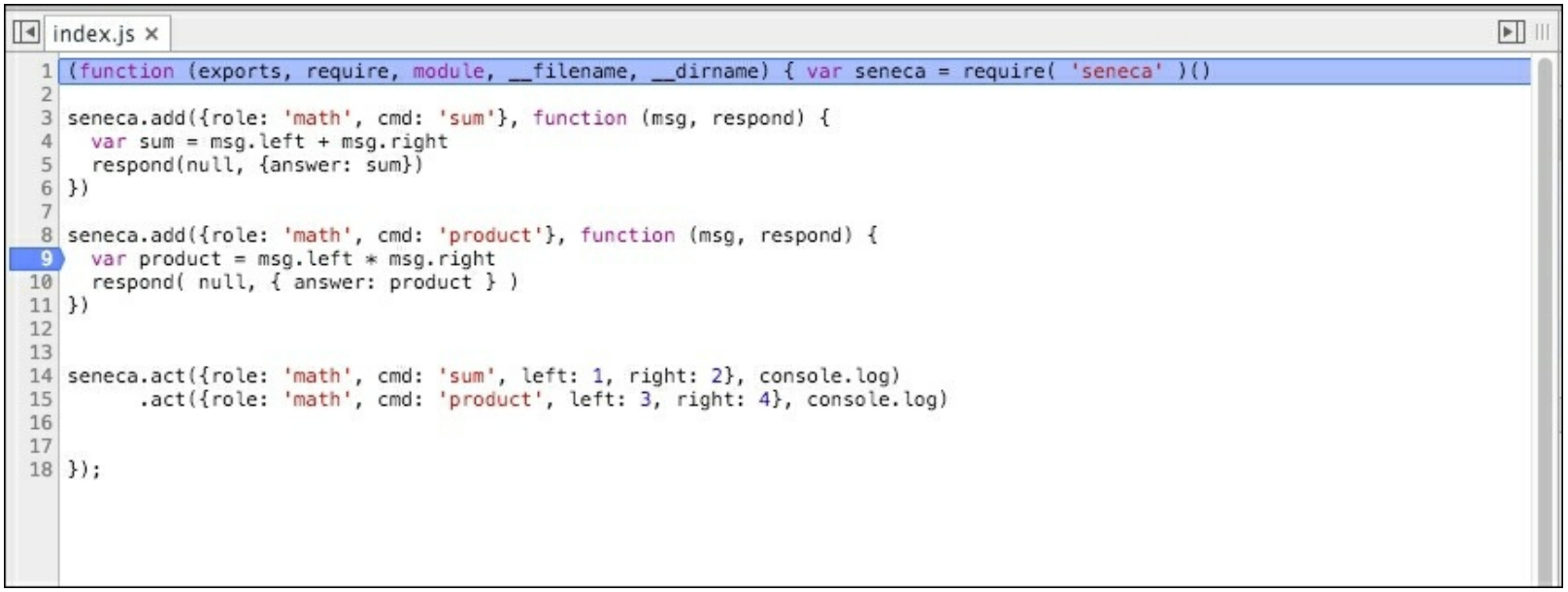

   seneca.act({role: 'math', cmd: 'sum', left: 1, right: 2}, console.log)
   seneca.act({role: 'math', cmd: 'product', left: 3, right: 4}, console.log)
   ```
3. Now, in order to run it on the debug mode, execute the following command:

```
node index.js --debug-brk
```

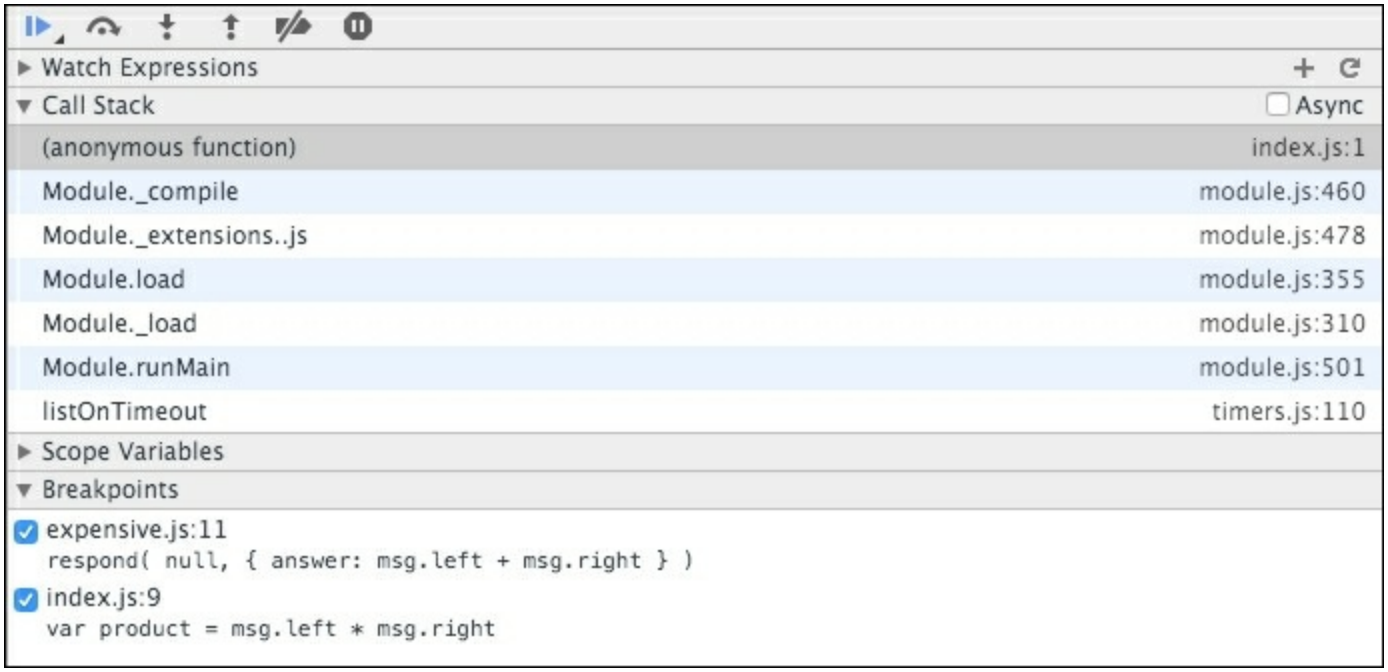
The way to access the debugger is through the URL `http://127.0.0.1:8080/?port=5858`:



I am sure this image is very familiar to every developer in the world: it is the Chrome debugger showing our code. As you can see in the first line, the one highlighted in blue, the application stopped in the first instruction so that we can place the breakpoints by clicking the line numbers, as shown in the following image:



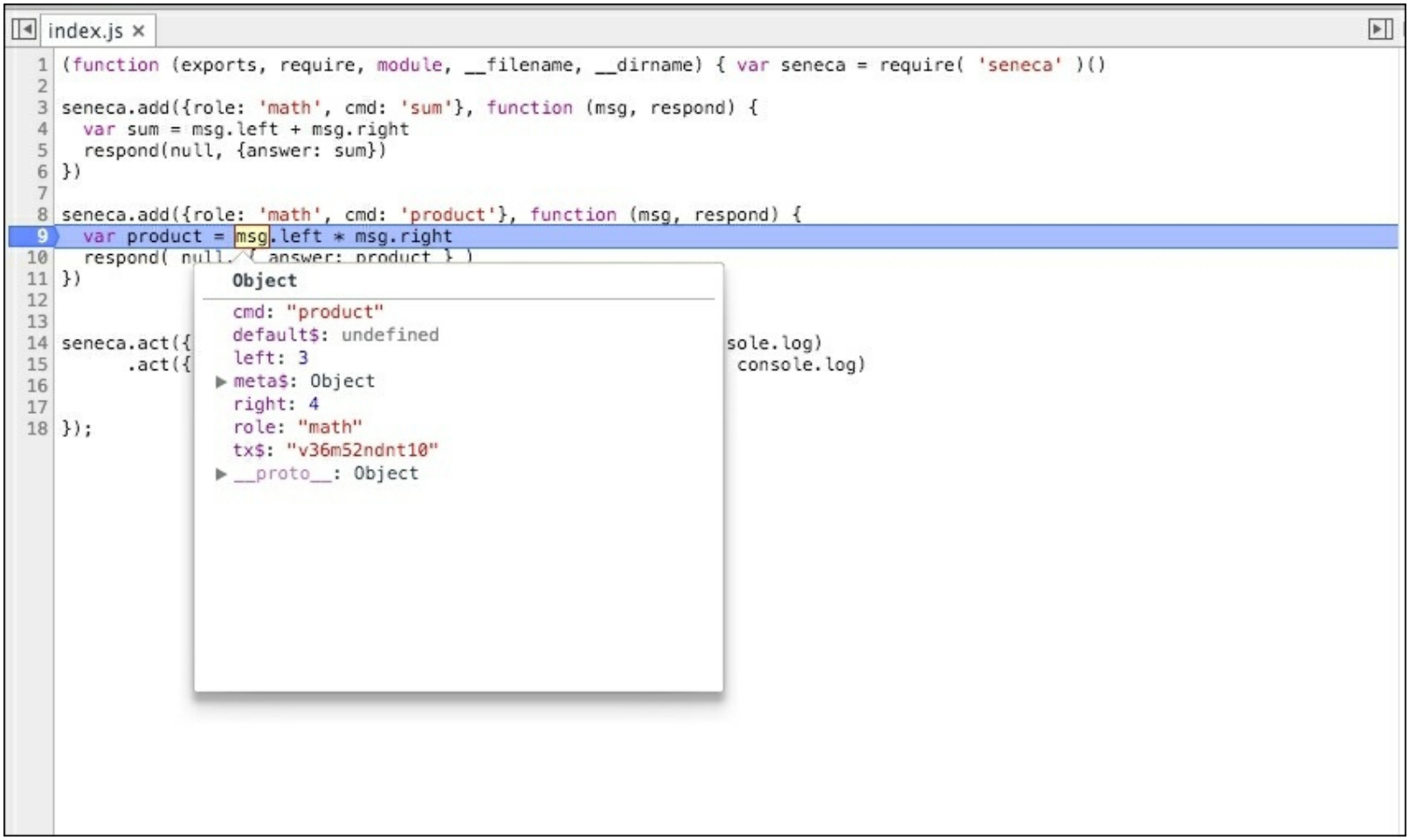
As you can see in the preceding image, we have installed a breakpoint in line **9**. Now we can use the control panel to navigate through the code and values of our variables:



The controls on the top speak for themselves if you ever debugged an application:

- The first button is called play and it allows the application to run to the next breakpoint
- Step over executes the next line in the current file
- Step into goes into the next line, getting deeper in the call stack so that we can see the call hierarchy
- Step out is the reverse of step into
- Disable breakpoints will prevent the program from stopping at the breakpoints
- Pause on exceptions, as its name indicates, will cause the program to stop on exceptions (it is very useful when trying to catch errors)

If we click on play, we can see how the script will stop in line **9** in the following image:



As a good debugger, it will let us inspect the value of our variables by hovering the cursor over the variable name.

Summary

This chapter has been pretty intense. We have gone through a lot of content that helped us in building a small microservices ecosystem that, when orchestrated together, would fairly work well. We have been a bit simplistic some times, but the idea of the book is to indicate the power of the microservices-oriented software. At this stage, I would recommend the reader to start performing some testing around Seneca.

The documentation on the website is quite helpful, and there are a lot of examples to follow.

There are a few plugins for storage and transport, as well as other type of plugins (such as user authentication), that would allow you to experiment with different features of Seneca.

We will be talking more about some of them in the following chapters.

Chapter 5. Security and Traceability

Security is one of the biggest concerns in systems nowadays. The amount of information leaked from big companies is worrying, especially because 90% of the information leaks could be mended with very small actions by the software developers. Something similar happens with the logging of events and the traceability of errors. No one really pays too much attention until someone requests the logs that you don't have in order to audit a failure. In this chapter, we will discuss how to manage security and logging so that our system is safe and traceable, with the help of the following topics:

- **Infrastructure logical security**: We will discuss how to secure our software infrastructure in order to provide the industry standard security layer in our communications.
- **Application security**: We will introduce the common techniques to secure our applications. Practices such as output encoding or input validation are the industry standard and they could save us from a catastrophe.
- **Traceability**: Being able to follow the requests around our system is a must in microservices architectures. We will leverage this task to Seneca and learn how to get the information from this fantastic framework.
- **Auditing**: Even though we put our best efforts in building a software, accidents happen. The ability to rebuild the sequence of calls and see exactly what happened is important. We will discuss how to enable our system in order to be able to recover the required information.

Infrastructure logical security

Infrastructure security is usually ignored by software engineers as it is completely different from their area of expertise. However, nowadays, and especially if your career is leaning towards DevOps, it is a subject that should not be ignored.

In this book, we are not going to go very deep into the infrastructure security more than few rules of thumb to keep your microservices safe.

It is strongly recommended for the readers to read and learn about cryptography and all the implications around the usage of SSH, which is one of the main resources for keeping communications secure nowadays.

SSH – encrypting the communications

In any organization, there is a strict list of people who can access certain services. In general, this authentication for these services is done via username and password, but it can also be done using a key to *verify the identity of the user*.

No matter what authentication method is used, the communication should always be done over a secure channel such as **SSH**.

SSH stands for **Secure Shell** and it is a software used to access shells in remote machines, but it can also be a very helpful tool to create proxies and tunnels to access remote servers.

Let's explain how it works using the following command:

```
/home/david:(develop) x ssh david@192.168.0.1
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is SHA256:S22/A2/eqxSqkS4VfR1BrcDxNX1rmfM1JkZaGhrjMbk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of known hosts.
vagrant@192.168.0.1's password:
Last login: Mon Jan 25 02:30:21 2016 from 10.0.2.2
Welcome to your virtual machine.
```

In this case, I am using **Vagrant** to facilitate the building of virtual machines. Vagrant is a very popular tool to automate development environments and their website (https://www.vagrantup.com/) consists of useful information.

In the first line, we execute the `ssh david@192.168.0.1` command. This command tries to open a terminal as the user `david` in the `192.168.0.1` host.

As it is the first time that this command is executed against the machine in the IP `192.168.0.1`, our computer will not trust the remote server.

This is done by maintaining a file called `known_hosts`, under the `/home/david/.ssh/known_hosts` folder in this case (it will depend on the user).

This file is a list of hosts with the corresponding key. As you can see, the following two lines explain that the *host cannot be trusted* and present the fingerprint of the key held by the remote server in order to verify it:

```
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is SHA256:S22/A2/eqxSqkS4VfR1BrcDxNX1rmfM1JkZaGhrjMbk.
```

At this point, the user is supposed to verify the identity of the server by checking the key. Once this is done, we can instruct SSH to connect to the server, which will result in the following log being printed:

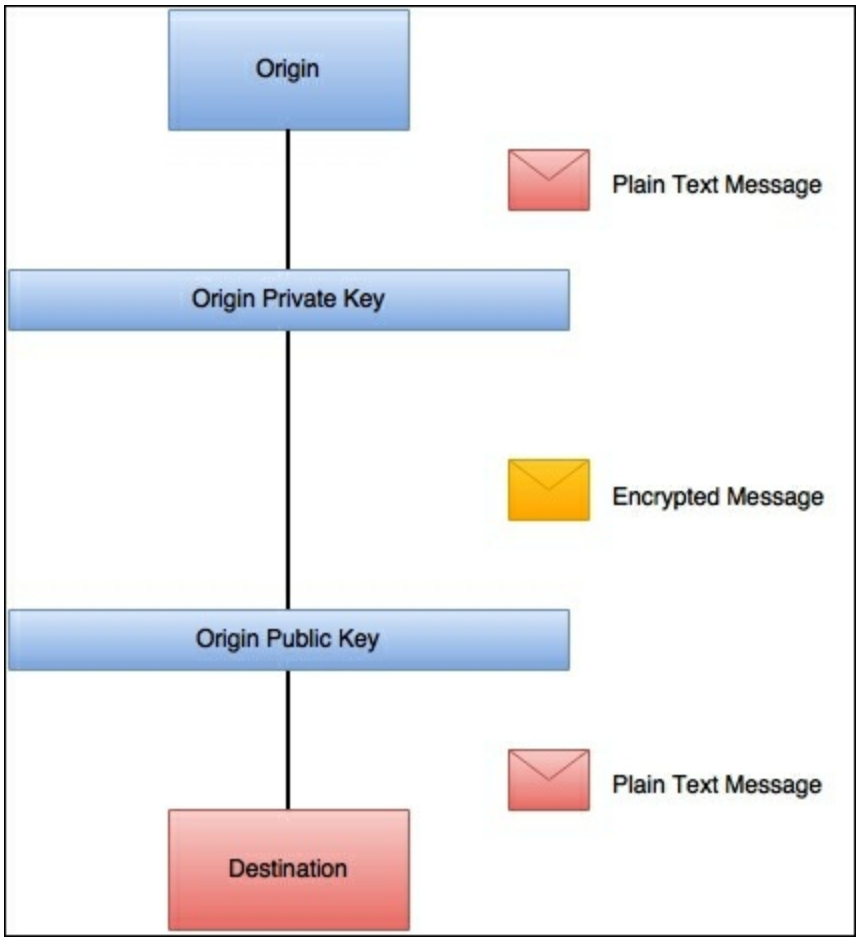
```
Warning: Permanently added '192.168.0.1' (RSA) to the list of known hosts.
```

Now, if we check our `known_hosts` file, we can see that the key has been added to the list, as follows:

```
192.168.0.1 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAxO/9E+joR8X46RL2V/wbcC15+qmQGPjfXsfpn97GV
OazzNgndR16t6WSxXmUR71fbEsjeZRYdhGp4ckkDh8AZ01MbNPuP6cKWHqy0LtOxXQR5hF/unShU8pwOPPJn8RxPB
ia3SLQ3BskfNxOrUijGqKs1JuRfeQafPuHvsOQ2kJH8PYD2UyEreHuLWiEuaiQuIguG8UiNEUkuIJEAyhD+PGVMLV
khhlTMZ+Pl0BhK7Q/9kF1e8D/ws2iBIB6I3oQx/FGW2dXuLboxODPX7iRgazf8YRv2lIWkKrh+qoD7sjTVCUgMMd4
1TbPIkNf3yrkDUQaRrfdWF0KXQ8JbNuKGhvgw==
```

This key stored in the `known_hosts` file is the public key of the remote server.

SSH uses a **cryptography algorithm** called **RSA**. This algorithm is built around the concept of **asymmetric cryptography**, which is shown in the following image:



The asymmetric cryptography relies on a set of keys: one public and one private. As the name states, the public key can be shared with everyone; whereas, the private key has to be kept secret.

The messages encrypted with the private key can only be decrypted with the public key and the other way around so that it is almost impossible (unless someone gets the other half of the key) to intercept

and decrypt a message.

At this point, our computer knows the public key of the server and we are in a position to start an encrypted session with the server. Once we get the terminal, all the commands and results of these commands will be encrypted and sent over the wire.

This key can also be used to connect to a remote server without password. The only thing we need to do is generate an SSH key in our machine and install it in the server in a file called `authorized_keys` under the `.ssh` folder, where the `known_hosts` file is.

When working with microservices, you can be remotely logged in to quite a few different machines so that this approach becomes more attractive. However, we need to be very careful about how we handle the private keys because if a user leaks that private key, our infrastructure could be compromised.

Application security

Application security is becoming more and more important. As the cloud is becoming the de-facto standard for infrastructure in large companies, we can't rely on the fact that the data is confined in a single data centre.

Usually, when someone starts a new business, the main focus is to build the product from the functional point of view. Security is not the main focus and usually gets overlooked.

This is a very dangerous practice and we are going to amend that by letting the reader know the main security threats that could compromise our application.

The main four big security points to develop applications in a secure manner are as follows:

- Injection
- Cross-site scripting
- Cross-site request forgery token protection
- Open redirects

At the end of this section, we will be able to identify the main vulnerabilities, but we won't be armored against a malicious attacker. In general, a software engineer should be up to date with the security as much as they are up to date with new technologies. No matter how good the product you build is, if it is not secure, someone will find out and take the advantage of it.

Common threats – how to be up to date

As we stated before, security is an ongoing subject in application development. No matter what type of software you are building, there will always be security implications around it.

The best way I've found during my professional career to be up to date with security around web development without being a full-time dedicated security engineer is to follow the **OWASP** project. **OWASP** stands for **Open Web Application Security Project** and they produce quite an interesting document (among others) on a yearly basis called OWASP Top 10.

Note

OWASP Top 10 was first published in 2003 and its goal is to raise awareness in the development community about the most common threats in application development.

In the previous section, we identified the four main security issues that a software developer can face and all of them are mentioned in the following sections.

Injection

Injection is, by far, the most dangerous attack that we could be exposed to. Specifically, a SQL injection is the most common form of injection that affects applications and it consists of an attacker forcing a SQL code in one of our application queries, leading to a different query that could compromise the data of our company.

There are other types of injections, but we are going to focus on SQL injection, as pretty much every application in the modern world uses a relational database.

SQL injection consists of the injection or manipulation of SQL queries in our application through the input from non-validated sources, such as a web form or any other data source with arbitrary input of text.

Let's consider the following example:

```
SELECT * FROM users WHERE username = 'username' AND password = 'password'
```

Tip

Never store passwords in plain in the database. Always hash and salt them to avoid rainbow-table attacks. This is just an example.

This query will give us the user that corresponds to a given name and password. In order to build the query from the client's input, we can consider doing something similar to the following code as a good idea:

```
var express = require('express');
var app = express();
```

```
var mysql      = require('mysql');

var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'test_db'
});

app.get('/login', function(req, res) {
  var username = req.param("username");
  var password = req.param("password");

  connection.connect();
  var query = "SELECT * FROM users WHERE username = '" + username + "' AND
password = '" + password + "'";
  connection.query(query, function(err, rows, fields) {
    if (err) throw err;
    res.send(rows);
  });
  connection.end();
});

app.listen(3000, function() {
  console.log("Application running in port 3000.");
});
```

At first sight, it looks like an easy program that accesses the database called `test_db` and issues a query to check whether there is a user that matches the username and password and renders it back to the client so that if we open the browser and try to browse to the `http://localhost:3000/login?username=david&password=mypassword` URL, the browser will render a JSON object with the result of the following query:

```
SELECT * FROM users WHERE username = 'david' AND password = 'mypassword'
```

Nothing strange yet, but what happens if the customer tries to hack us?

Take a look at the following input:

```
http://localhost:3000/login?username=' OR 1=1 --&password=mypassword
```

As you can see, the query generated by it is the following code:

```
SELECT * FROM users WHERE username = '' OR 1=1 -- AND password = 'mypassword'
```

In SQL, the `--` character sequence is used to comment the rest of the line so that the effective query would be as follows:

```
SELECT * FROM users WHERE username='' OR 1=1
```

This query returns the full list of users, and if our software is using the result of this query to resolve

whether the user should be logged in or not, we are in some serious problems. We have just granted access to our system to someone who does not even know a valid username.

This is one of the many examples on how SQL injection can affect us.

In this case, it is pretty obvious that we are concatenating untrusted data (coming from the user) into our query, but believe me, when the software gets more complicated, it is not always easy to identify.

A way to avoid SQL injection is through the usage of prepared statements.

Input validation

Applications interact with users mainly through forms. These forms usually contain free text input fields that could lead to an attack.

The easiest way to prevent corrupted data from getting into our server is through input validation, which as the name suggests, consists of validating the input from the user to avoid the situation described earlier.

There are two types of input validation, as follows:

- White listing
- Black listing

Black listing is a dangerous technique. In majority of cases, trying to define what is incorrect in the input takes a lot more effort than simply defining what we expect.

The recommended approach is (and will always be) to **white list** the data coming from the user, validating it through the use of a regular expression: we know how a phone number looks like, we also know how a username should look like, and so on.

Input validation is not always easy. If you have ever come across the validation of an e-mail, you will know what I am talking about: the regular expression to validate an e-mail is anything but simple.

The fact that there is not an easy way to validate some data should not restrict us from doing it as the omission of input validation could lead to a serious security flaw.

Input validation is not the silver bullet for SQL injections, but it also helps with other security threats such as cross-site scripting.

In the query from the previous section, we do something quite dangerous: concatenate user input into our query.

One of the solutions could be to use some sort of escaping library that will sanitize the input from the user, as follows:

```
app.get('/login', function(req, res) {
```

```
  var username = req.param("username");
  var password = req.param("password");

  connection.connect();
  var query = "SELECT * FROM users WHERE username = '" +
connection.escape(username) + "' AND password = '" + connection.escape(password)
+ "'";
  connection.query(query, function(err, rows, fields) {
    if (err) throw err;
    res.send(rows);
  });
  connection.end();
});
```

In this case, the `mysql` library used provides a suite of methods to escape strings. Let's see how it works:

```
var mysql = require('mysql');
var connection = mysql.createConnection({
  host: 'localhost',
  username: 'root',
  password: 'root'
});

console.log(connection.escape("' OR 1=1 --"))
```

The small script from earlier escapes the string provided as `username` in the previous example, the result is `\' OR 1=1 --`.

As you can see, the `escape()` method has replaced the dangerous characters, sanitizing the input from the user.

Cross-site scripting

Cross-site scripting, also known as **XSS**, is a security vulnerability that mainly affects web applications. It is one of the most common security issues and the implications can be huge for the customer as potentially, someone could steal the user identity with this attack.

The attack is an injection code put into a third-party website that could steal data from the client's browser. There are a few ways of doing it, but by far, the most common is by unescaped input from the client.

In few websites on the Internet, users can add comments containing arbitrary input. This arbitrary input can contain script tags that load a JavaScript from a remote server that can steal the session cookie (or other types of valuable information), letting the attacker replicate the user session on a remote machine.

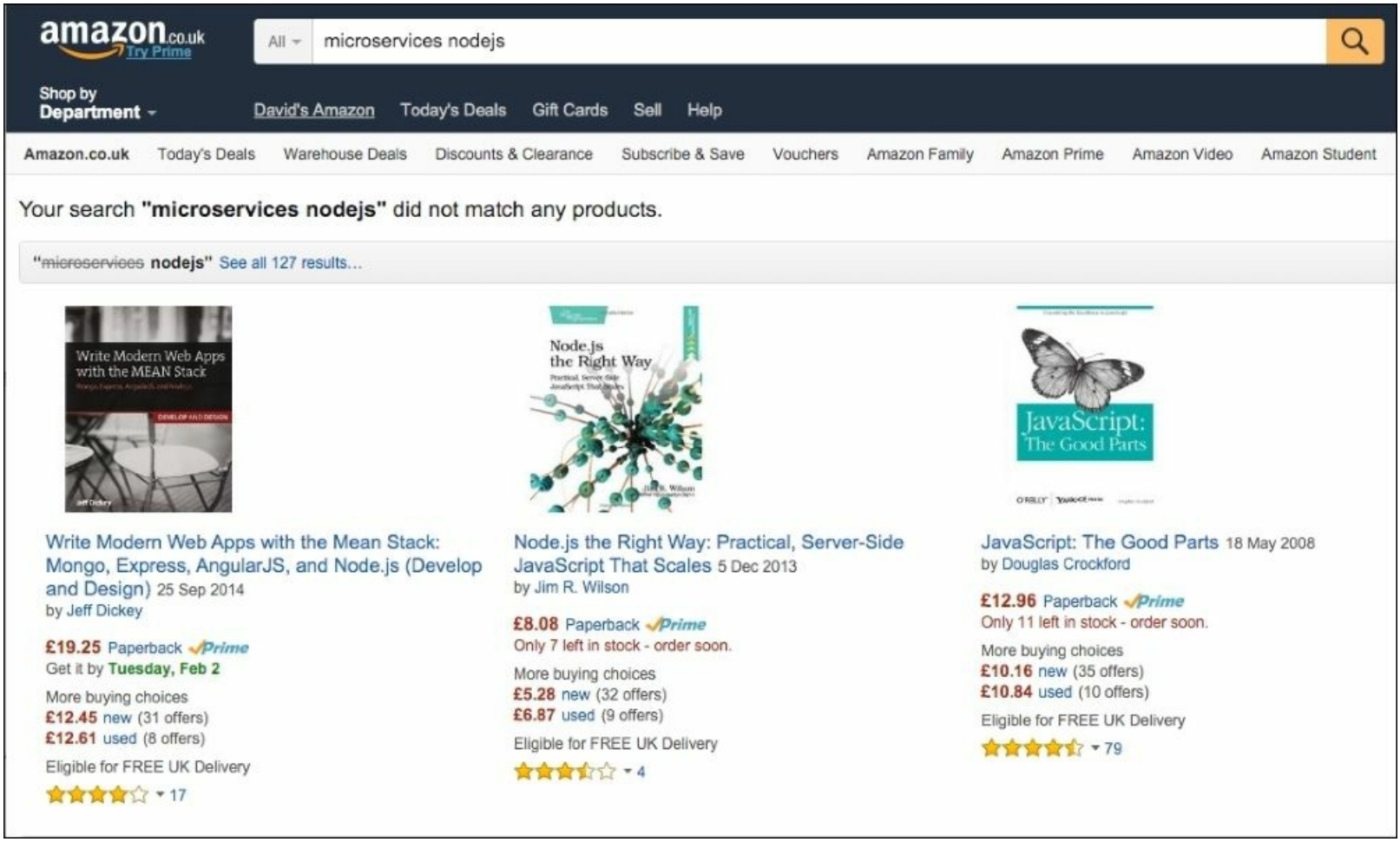
There are two main types of XSS attacks: **persistent** and **non-persistent**.

The **persistent** type of XSS consists of storing the XSS attack by crafting a particular string of text

that resolves into the attack once it is displayed to the user in the website. This code could be injected via an arbitrary input text that is stored in the database (such as a comment in a forum).

The **non-persistent** type of XSS is when the attack is inserted into a non-persistent part of the application due to bad data handling.

Let's take a look at the following screenshot:



As you can see, we have searched for a book (this book) in http://www.amazon.co.uk/. It does not produce any output (as the book is not published yet), but it specifies that **Your search "microservices nodejs" did not match any products**, which is somehow using the input from the web browser as output. Even more, when I clicked on search, Amazon redirected me to the following URL:

http://www.amazon.co.uk/s/ref=nb_sb_noss?url=search-alias%3Daps&field-keywords=microservices+nodejs

We know Amazon is secure, but if it was sensible to XSS attacks, we could have modified the value of the `field-keywords` parameter to craft a request that injected a script tag in the content, leading to the attacker being able to steal the session cookie that could result in some serious problems for the website.

Output encoding

A way to protect against this attack is output encoding. We have done it before, when we used `connection.escape()` in the *Input validation* section of this chapter. In fairness, we should be validating all the data entered from the user and encoding all the outputs that come from third parties. This includes the input entered by the user, as well as sources of information coming from outside of the system.

When narrowing the problem to web development, we have to be aware of the three different areas where output encoding is needed:

- CSS
- JavaScript
- HTML

The most problematic two are JavaScript and HTML, where an attacker could easily steal information without too much effort.

Generally, no matter which framework we use for building our app, it always has functions to encode the output.

Cross-site request forgery

Cross-site request forgery (**CSRF**) is the reverse of cross-site request scripting. In cross-site request scripting, the problem is in the client trusting the data coming from the server. With cross-site request forgery, the problem is that the server trusts the data coming from the client.

After stealing the session cookie, the attacker can not only steal information from the user, but can also modify the information of the account associated with the cookie.

This is done by posting the data to the server via HTTP requests.

HTTP classifies its requests in methods. A method is basically used to specify what is the operation to be carried by the request. The most interesting four methods are the following ones:

- `GET`: This gets the data from the server. It should not modify any persistent data.
- `POST`: This creates a resource in the server.
- `PUT`: This updates a resource in the server.
- `DELETE`: This deletes a resource from the server.

There are more methods (such as `PATCH` or `CONNECT`), but let's focus on these four. As you can see, three of these four methods modify data from the server, and a user with a valid session could potentially steal data, create payments, order goods, and so on.

A way to avoid the cross-site request forgery attack is by protecting the `POST`, `PUT` and `DELETE` endpoints with a cross-site request token.

Take a look at the following HTML form:

```
<form action="/register" method="post">
  <input name="email" type="text" />
  <input name="password" type="password" />
</form>
```

This form describes a perfectly valid situation: a user registering on our website; very simple, but still valid and flawed.

We are specifying a URL and the list of expected parameters so that an attacker can register hundreds or thousands of accounts within a span of minutes, with a small script that issues a `POST` request with the two parameters (`email` and `password`) in the body.

Now, look at the following form:

```
<form action="/register" method="post">
  <input name="email" type="text" />
  <input name="password" type="password" />
  <input name="csrftoken" type="hidden" value="as7d6fasd678f5a5sf5asf" />
</form>
```

You can see the difference: there is an extra hidden parameter called `csrftoken`.

This parameter is a random string that is generated every time a form is rendered so that we can add this extra parameter to every form.

Once the form is submitted, the `csrftoken` parameter is validated to only let go through the requests with a valid token and generate a new token to be rendered on the page again.

Open redirects

Sometimes, our application might need to redirect the user to a certain URL. For example, when hitting a private URL without a valid authentication, the user will usually be redirected to the login page:

```
http://www.mysite.com/my-private-page
```

This could result into a redirect to the following:

```
http://www.mysite.com/login?redirect=/my-private-page
```

This sounds legit. The user is sent to the login page, and once he provides a valid set of credentials, it is redirected to `/my-private-page`.

What happens if someone tries to steal the account of our user?

Look at the following request:

```
http://www.mysite.com/login?redirect=http://myslte.com
```

This is a crafted request that will redirect the user to `myslte.com` instead of `mysite.com` (note the `l` instead of `i`).

Someone could make `myslte.com` look like the login page of `mysite.com` and steal your user's password and username by distributing the preceding URL in the social media as the users will be redirected to a malicious page.

The solution for the preceding problem is quite simple: don't redirect the user to untrusted third-party websites.

Again, the best way of doing such task is white listing the target hosts for redirects. Basically, we don't let our software redirect our customers to unknown websites.

Effective code reviews

One of the most effective ways to reduce security flaws in our applications is through a systematic and informed code review process. The problem with code reviews is that they always end up being a dump area for opinions and personal preferences that usually not only won't improve the quality of the code, but will also lead to last minute changes that could expose vulnerabilities in our application.

A dedicated stage in the product development life cycle for a security code review helps to drastically reduce the amount of bugs delivered to production.

The problem that the software engineers have is that their mind is trained to build things that work well, but they don't have the mindset to find defects, especially around the things that they build. This is why you should not be testing your own code (any further than the test carried on when developing), and even less, security testing your application.

However, we usually work in teams and that enables us to review someone else's code, but we have to do it in an effective manner.

Code reviews require as much brain power as needed to write software, especially if you are reviewing a complex code. You should never spend more than two hours reviewing the same functionality, otherwise important flaws will be missed and attention to detail will decrease to a worrying level.

This is not a big problem in microservices-based architectures as the functionality should be small enough to be read in a reasonable period of time, especially if you talked to the author about what he was trying to build.

You should always follow a two phase review, as follows:

- Review the code quickly to get the big picture: how it works, what technology it uses that you are not familiar with, does it do what it is supposed to do, and so on
- Review the code following a checklist of items to look for

This list of items has to be decided upfront and depends on the nature of the software that your company is building.

Usually, the list of items to check around the code security concerns during a code review is quite big, but we can narrow it down to the following components:

- Is all the input validated/encoded when applicable?
- Is all the output encoded, including logs?
- Do we protect endpoints with cross-site request forgery tokens?
- Are all the user credentials encrypted or hashed in the database?

If we check this list, we will be able to identify the biggest issues around security in our apps.

Traceability

Traceability is extremely important in the modern information systems. It is a delicate matter in microservices that is gracefully solved in Seneca, making the requests easy to follow around our system so that we can audit the failure.

Logging

Seneca is pretty good with the logging. There are so many options that can be configured in Seneca in order to get the required information about how everything is working (if it is working).

Let's see how logging works with a small application:

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  callback(null, {message: "Hello " + args.name});
});

seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  console.log(result);
});
```

This is the simplest Seneca application that can be written. Let's run it as follows:

```
seneca  node index.js
2016-02-01T09:55:40.962Z 3rhomq69cbe0/1454579740947/84217/- INFO hello
Seneca/1.0.0/3rhomq69cbe0/1454579740947/84217/-
{ message: 'Hello David' }
```

This is the result of running the app with the default logging configuration. Aside from the `console.log()` method that we have used in the code, there is some internal information about Seneca being logged. Sometimes, you might want to only log what your application is producing so that you can debug the application without all the noise. In this case, just run the following command:

```
seneca  node index.js --seneca.log.quiet
{ message: 'Hello David' }
```

However, sometimes, there are weird behaviors in the system (or even a bug in the frameworks used) and you want to get all the information about what is happening. Seneca supports that as well, as shown in the following command:

```
seneca  node index.js --seneca.log.print
```

The preceding command will print an endless amount of information that might not be helpful.

In order to reduce the amount of logging produced by Seneca, there is a fine-grain control in what gets logged into the output. Let's take a look at the following lines:

```
2016-02-01T10:00:07.191Z dyy9ixcavqu4/1454580006885/85010/- DEBUG register
install transport {exports:[transport/utils]} seneca-8t1dup
2016-02-01T10:00:07.305Z dyy9ixcavqu4/1454580006885/85010/- DEBUG register init
seneca-y9os9j
2016-02-01T10:00:07.305Z dyy9ixcavqu4/1454580006885/85010/- DEBUG plugin seneca-
y9os9j DEFINE {}
2016-02-01T10:00:07.330Z dyy9ixcavqu4/1454580006885/85010/- DEBUG act root$
```

```
IN o5onzziv9i7a/b7dtf6v1u9sq cmd:greeter {cmd:greeter,name:David} ENTRY (mnb89)
- - -
```

They are random lines from a log output on the preceding code example, but it will give us useful information: these entries are debug-level log lines for different actions (such as plugin, register, and act) on the Seneca framework. In order to filter them, Seneca provides a control over what levels or actions do we want to see. Consider the following for example:

```
node index.js --seneca.log=level:INFO
```

This will only output the logs related to the `INFO` level:

```
seneca  node index.js --seneca.log=level:INFO
2016-02-04T10:39:04.685Z q6wnh8qmm1l3/1454582344670/91823/- INFO hello
Seneca/1.0.0/q6wnh8qmm1l3/1454582344670/91823/-
{ message: 'Hello David' }
```

You can also filter by action type, which is quite interesting. When you are working with microservices, knowing the chain of events that happened in a flow is one of the first things that you need to look into in order to audit a failure. With this control over the logging that Seneca gives us, it is as easy as executing the following command:

```
node index.js --seneca.log=type:act
```

This will produce the following output:



As you can see, all the preceding lines correspond to the `act` type, and even more, if we follow the output of the command from top to bottom, we exactly know the sequence of events to which Seneca reacted and their order.

Tracing requests

Tracing requests is also a very important activity, and sometimes, it is even a legal requirement, especially if you work in the world of finance. Again, Seneca is fantastic at tracing requests. For every call, Seneca generates a unique identifier. This identifier can be traced across all the paths to where the call is going to, as follows:

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  console.log(this.fixedargs['tx$']);
  callback(null, {message: "Hello " + args.name});
});
seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  console.log(this.fixedargs['tx$']);
});
```

Here, we are logging a dictionary that contains the transaction ID in Seneca to the terminal. So, if we execute it, we will get the following output:

```
2016-02-04T10:58:07.570Z zl0u7hj3hbeg/1454583487555/95159/- INFO hello
Seneca/1.0.0/zl0u7hj3hbeg/1454583487555/95159/-
3jlroj2n91da
3jlroj2n91da
```

You can see how all the requests in Seneca are traced: the framework assigns an ID and it gets propagated across endpoints. In this case, all our endpoints are in the local machine, but if we distribute them in different machines, the ID will still be the same.

With this unique ID, we will be able to reconstruct the journey of the customer data in our system, and ordering the requests with the associated timestamp, we can get an accurate picture of what the user was doing, how much time did every action take, what are the possible problems associated with delays, and so on. Usually, the logging combined with circuit breakers output information allows the engineers to solve issues within a very reduced time frame.

Auditing

Up till now, we have been using `console.log()` to output the data into the logs. This is a bad practice. It breaks the format of the logs and throws the content to the standard output.

Again, Seneca comes to the rescue:

```
var seneca = require("seneca")();

seneca.add({cmd: "greeter"}, function(args, callback){
  this.log.warn(this.fixedargs['tx$']);
  callback(null, {message: "Hello " + args.name});
});

seneca.act({cmd: "greeter", name: "David"}, function(err, result) {
  this.log.warn(this.fixedargs['tx$']);
});
```

Let's see what Seneca produces as output:

```
seneca  node index.js
2016-02-04T11:17:28.772Z wo10oa299tub/1454584648758/98550/- INFO hello
Seneca/1.0.0/wo10oa299tub/1454584648758/98550/-
2016-02-04T11:17:29.156Z wo10oa299tub/1454584648758/98550/- WARN - - ACT
02jlpyiux70s/9ca086d19x7n cmd:greeter 9ca086d19x7n
2016-02-04T11:17:29.157Z wo10oa299tub/1454584648758/98550/- WARN - - ACT
02jlpyiux70s/9ca086d19x7n cmd:greeter 9ca086d19x7n
```

As you can see, we are now outputting the transaction ID using the logger. We have produced a `WARN` message instead of a simple console dump. From now on, we can use Seneca log filters to hide the output of our actions in order to focus on what we are trying to find.

Seneca provides the following five levels of logging:

- **DEBUG**: This is used to debug applications when you are developing them and also trace problems in production systems.
- **INFO**: This log level is used to produce important messages about events such as a transaction has started or completed.
- **WARN**: This is the warning level. We use it when something bad happens in the system, but it is not critical, the user usually does not get affected; however, it is an indication that something is going in the wrong way.
- **ERROR**: This is used to log errors. Generally, the user gets affected by it and it also interrupts the flow.
- **FATAL**: This is the most catastrophic level. It is only used when a non-recoverable error has occurred and the system won't be able to function normally.

A way to produce logs in different levels is to use the associated functions. As we have seen earlier, we called `this.log.warn()` to log a warning. If we call the `this.log.fatal()` method, we will be

logging a fatal error, and same with the other levels.

Tip

Try to adjust the logs in your application as a part of the development process or you will regret the lack of information when something bad occurs in production.

In general, INFO, DEBUG, and WARN will be the most used log levels.

HTTP codes

HTTP codes are often ignored, but they are a really important mechanism to standardize responses from remote servers.

When a program (or user) issues a request to a server, there are a few things that could happen, as follows:

- It could be successful
- It could fail validation
- It could produce a server error

As you can see, the possibilities are endless. The problem that we now have is that HTTP was created for the communication between machines. How do we handle the fact that machines will be reading these codes?

HTTP solved this problem in a very elegant way: every single request has to be resolved with an HTTP code and these codes have ranges that indicate the nature of the code.

1xx – informational

The codes in the 100-199 range are purely informational. The most interesting code in this range is the 102 code. This code is used to specify that an operation is happening in the background and might take some time to complete.

2xx – success codes

Success codes are used to indicate a certain level of success in the HTTP request. It is the most common (and desired) codes.

The most common codes in this range are as follows:

- `200: Success`: This code indicates a full success. Nothing went wrong even remotely.
- `201: Created`: This code is used mainly for REST APIs when the client requests to create a new entity in the server.
- `203: Non-authoritative information`: This code is intended to be used when, while routing the request through a transforming proxy, the origin responds with a 200.
- `204: No Content`: This is a successful code, but there is no content coming back from the server. Sometimes, APIs returns 200, even if there is no content.
- `206: Partial Content`: This code is used for paginated responses. A header is sent, specifying a range (and an offset) that the client will accept. If the response is bigger than the range, the server will reply with 206, indicating that there is more data to follow.

3xx – redirection

The codes in the 300 to 399 range indicate that the client must take some additional actions to

complete the request.

The most common codes in this range are described as follows:

- `301: Moved permanently`: This status code is indicating that the resource that the client was trying to get has been moved permanently to another location.
- `302: Found`: This code indicates that the user is required to perform a temporary redirect for some reason, but the browsers started implementing this code as `303 See Other`. This lead to the introduction of the 303 and `307 Temporary redirect` codes to disambiguate the overlap of behavior.
- `308 Permanent Redirect`: This code, as the name indicates, is used to specify a permanent redirect for a resource. It could be confused with 301, but there is a small difference, the 308 code does not allow the HTTP method to change.

4xx – client errors

The codes in the 400 to 499 range represent errors generated by the client. They indicate that there is a problem with the request. This range is particularly important as it is the way that HTTP servers have to indicate the clients that something is wrong with their request.

The common codes in this range are as follows:

- `400 Bad Request`: This code indicates that the request from the user is syntactically incorrect. There could be parameters missing or some of the values didn't pass validation.
- `401 Unauthorized`: This code represents a lack of authentication of the client. Usually, a valid login will fix this problem.
- `403 Forbidden`: This is similar to 401, but in this case, it is indicating that the user does not have enough privileges.
- `404 Not Found`: This means that the resource is not found in the server. This is the error that you get when you navigate to a page that does not exist.

5xx – server errors

This range indicates that there has been a processing error in the server. When a 5xx code is issued, it means that there was some sort of problem in the server and it cannot be fixed from the client.

Some of the codes in this range are as follows:

- `500 Internal Server Error`: This means that an error has occurred in the software in the server. There is no more information disclosed.
- `501 Not Implemented`: This error occurs when a client hits an endpoint that has not been implemented yet.
- `503 Service unavailable`: This code is issued when the server is not available for some reason, either an excess of the load or the server is down.

Why HTTP codes matter in microservices

The popular saying *don't reinvent the wheel* is one of my favorite principles when building software.

HTTP codes are a standard, so everybody understands the consequences of the different codes.

When building microservices, you always need to keep in mind that your system will be interacting with proxies, caches, and other services that already speak HTTP so that they can react according to the response from the servers.

The best example of this is the circuit-breaker pattern. No matter how you implement it and what software you use, a circuit breaker has to understand that an HTTP request with 500 code is an error, so it can open the circuit accordingly.

In general, it is good practice to keep the codes of your application as accurate as possible as it will benefit your system in the long run.

Summary

In this chapter, you have learned how to build secure software (and not only microservices), although it is a subject big enough to write a full book on it. The problem with security is that companies usually see investing in security as burning money, but that is far from reality. I am a big fan of the 80-20 rule: 20% of time will give you 80% of features and the 20% of missing features will require 80% of the time.

In security, we really should be aiming for 100% coverage; however, the 80% shown in this chapter will cover majority of the cases. Anyway, as I mentioned before, a software engineer should be up to date with security as a flaw in the security of an application is the easiest way to kill a company.

We have also been talking about traceability and logging, one of the most ignored subjects in the modern software engineering that are becoming more and more important, especially if your software is built using a microservices approach.

Chapter 6. Testing and Documenting Node.js Microservices

Until now, all that we have done is develop microservices and discuss the frameworks around the process of building software components. Now it is time to test all of them. Testing is the activity of validating the software that has been built. Validating is a very broad term. In this chapter, we are going to learn how to test microservices, not only from the functional point of view, but we will also learn how to test the performance of our applications, as well as other aspects such as integration with different modules. We will also build a proxy using Node.js to help us to inspect the inputs and outputs of our services so that we can validate that what we have designed is actually happening and, once again, reassure the versatility of a language, such as JavaScript, to quickly prototype features.

It is also nowadays a trend to release features with an A/B test, where we only enable the features for certain type of users, and then we collect metrics to see how the changes to our system are performing. In this chapter, we will build a microservice that is going to give us the capability of rolling out features in a controlled way.

On the other hand, we are going to document our application, which unfortunately, is a forgotten activity in traditional software development: I haven't found a single company where the documentation captures 100% the information needed by new developers.

We will cover the following topics in this chapter:

- **Functional testing**: In this section, we will learn how to test microservices and what a good testing strategy is. We will also get to study a tool called Postman to manually test our APIs, as well as build a proxy with Node.js to spy our connections.
- **Documenting microservices**: We will learn how to use Swagger to document our microservices using the open API standard. We will also generate the code from the YAML definition using an open source tool.

Functional testing

Testing is usually a time-consuming activity that does not get all the required attention while building a software.

Think about how a company evolves:

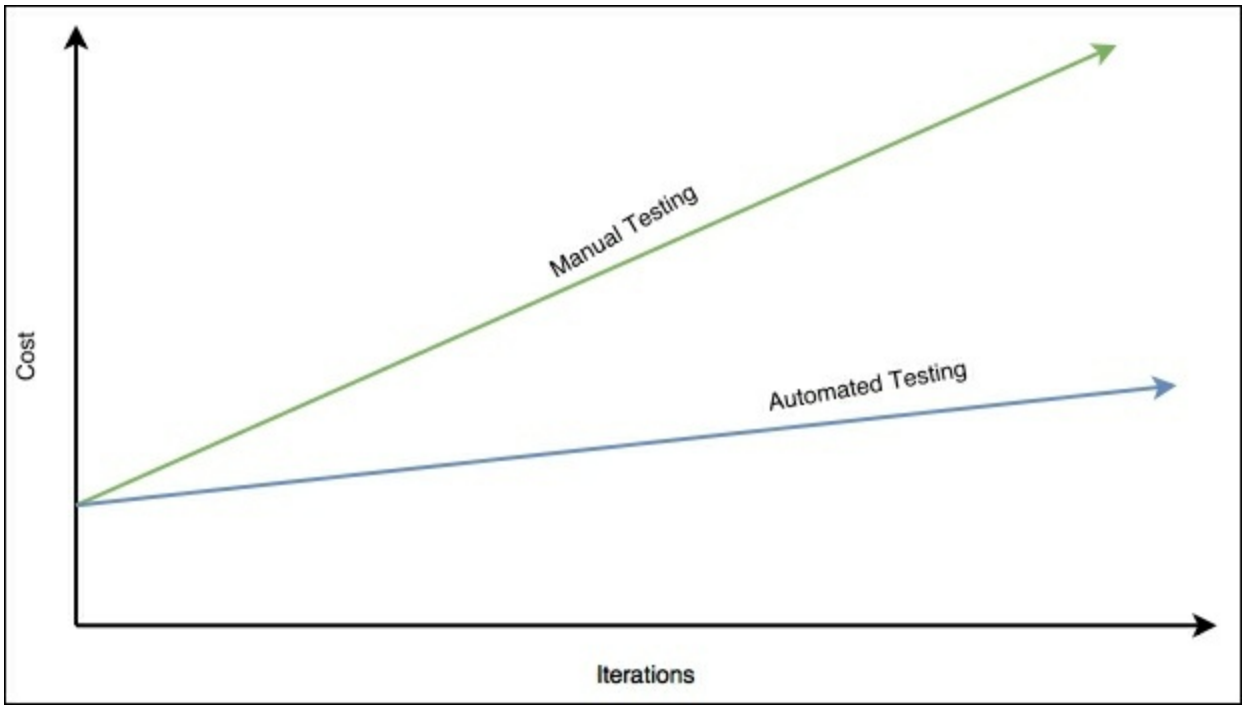
1. Someone comes up with an idea.
2. A few engineers/product people build the system.
3. The company goes to market.

There is no time to test more than the minimal required manual testing. Especially, when someone reads on the Internet that testing done right could take up to 40% of your development time, and once again, the common sense fails.

Automation is good and unit, integration, and end-to-end tests are a form of automation. By letting a computer test our software, we are drastically cutting down the human effort required to validate our software.

Think about how the software is developed. Even though our company likes to claim that *we are agile*, the truth is that every single software project has some level of iterative development, and testing is a part of every cycle, but generally, it is overlooked in favour of delivering new features.

By automating the majority (or a big chunk) of the testing, we are saving money, as shown in the following diagram:



Costs and Iterations

Testing is actually a cost saver if is done right, and the key is doing it right, which is not always easy. How much testing is too much testing? Should we cover every single corner of our application? Do we really need deep performance testing?

These questions usually lead to a different stream of opinions, and the interesting thing is that there is not a single source of truth. It depends on the nature of your system.

In this chapter, we are going to learn a set of extensive testing techniques, which does not mean that we should be including all of them in our test plan, but at least we will be aware of the testing methodologies.

In the past seven years, Ruby on Rails has created a massive trend towards a new paradigm, called **Test-driven development** (**TDD**), up to a point that, nowadays, majority of the new development platforms are built with TDD in mind.

Personally, I am not a fierce adopter of TDD, but I like to take the good parts. Planning the test before the development helps to create modules with the right level of cohesion and define a clear and easy-to-test interface. In this chapter, we won't cover the TDD in depth, but we will mention it a few times and explain how to apply the exposed techniques to a TDD test plan.

The pyramid of automated testing

How to lay down your testing plan is a tricky question. No matter what you do, you will always end up with the sensation that *this is completely wrong*.

Before diving into the deep, let's define the different type of tests that we are going to be dealing with from the functional point of view, and what should they be designed for.

Unit tests

A **unit test** is a test that covers individual parts of the application without taking into account the integration with different modules. It is also called **white box testing** as the aim is to cover and verify as many branches as possible.

Generally, the way to measure the quality of our tests is the test coverage and it is measured in percentage. If our code spans over ten branches and our tests cover seven branches, our code coverage is 70%. This is a good indication of how reliable our test coverage is. However, it could be misleading as the tests could be flawed, or even though all the branches are tested, a different input would cause a different output that wasn't captured by a test.

In unit tests, as we don't interact with other modules, we will be making a heavy use of mocks and stubs in order to simulate responses from third-party systems and control the flow to hit the desired branch.

Integration tests

Integration tests, as the name suggests, are the tests designed to verify the integration of our module in the application environment. They are not designed to test the branches of our code, but business units, where we will be saving the data into databases, calling third-party web services or other microservices of our architecture.

These tests are the perfect tool for checking whether our service is behaving as expected, and sometimes, could be hard to maintain (more often than not).

During my years of experience, I haven't found a company where the integration testing is done right and there are a number of reasons for this, as stated in the following list:

- Some companies think that integration testing is expensive (and it is true) as it requires extra resources (such as databases and extra machines)
- Some other companies try to cover all the business cases just with unit testing, which depending on the business cases, could work, but it is far from ideal as unit tests make assumptions (mocks) that could give us a false confidence in our test suite
- Sometimes, integration tests are used to verify the code branches as if they were unit tests, which is time consuming as you need to work out the environment to make the integration test to hit the required branch

No matter how smart you want to be, integration testing is something that you want to do right, as it is the first real barrier in our software to prevent integration bugs from being released into production.

End-to-end tests

Here, we will demonstrate that our application actually works. In an integration test, we are invoking the services at code level. This means that we need to build the context of the service and then issue the call.

The difference with end-to-end testing is that, in end-to-end testing, we actually fully deploy our application and issue the required calls to execute the target code. However, many times, the engineers can decide to bundle both type of tests (integration and end-to-end tests) together, as the modern frameworks allow us to quickly run E2E tests as if they were integration tests.

As the integration tests, the target of the end-to-end tests is not to test all the paths of the application but test the use cases.

In end-to-end tests, we can find a few different modalities (paradigms) of testing, as follows:

- We can test our API issuing JSON requests (or other type of requests)
- We can test our UI using Selenium to emulate clicks on the DOM
- We can use a new paradigm called **behavior-driven development** (**BDD**) testing, where the use cases are mapped into actions in our application (clicks on the UI, requests in the API, and so on) and execute the use cases for which the application was built

End-to-end tests are usually very fragile and they get broken fairly easy. Depending on our application, we might get relaxed about these tests as the cost-value ratio is pretty low, but still, I would recommend having some of them covering at least the most basic and essential flows.

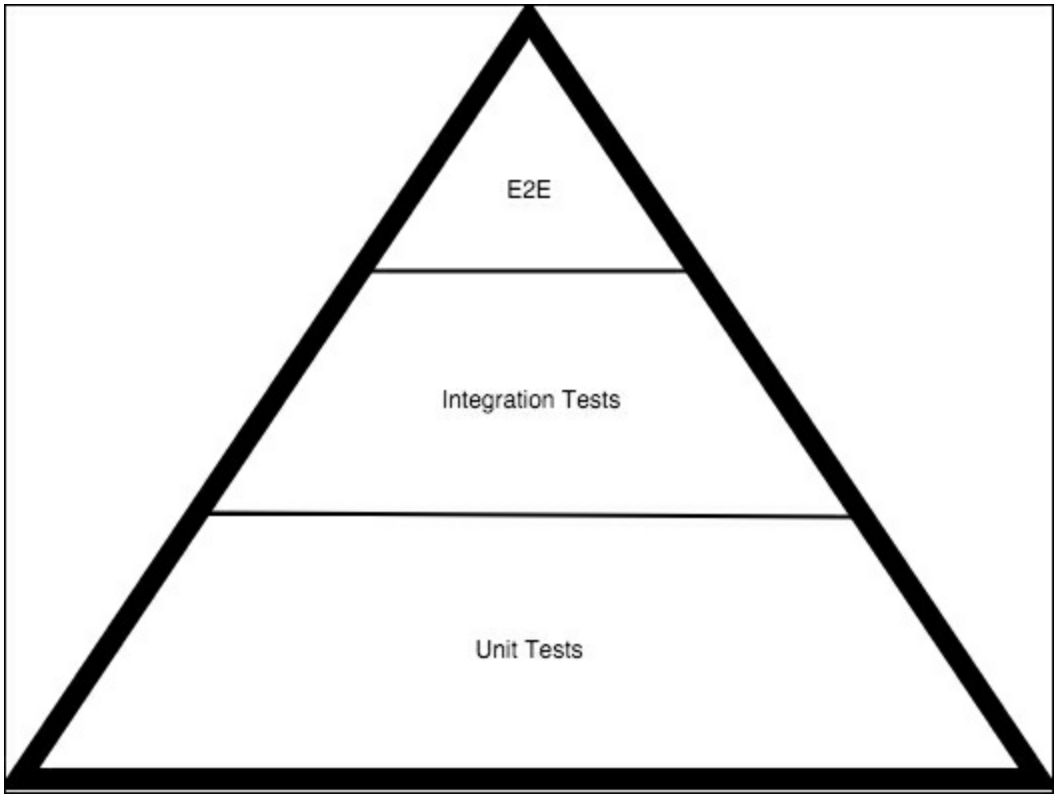
How much testing is too much?

Questions such as the following are not easy to answer, especially in fast paced businesses, like startups:

- Do we have too many integration tests?
- Should we aim for 100% unit test coverage?
- Why bother with Selenium tests if they break every second day for no reason?

There is always a compromise. Test coverage versus time consumed, and also, there is no simple and single answer to these questions.

The only useful guideline that I've found along the years is what the testing world calls the **pyramid of testing**, which is shown in the following figure. If you think for a moment, in the projects where you worked before, how many tests did you have in total? What percentage of these were integration tests and unit tests? What about end-to-end tests?:



The pyramid of testing

The preceding pyramid shows the answers for these questions. In a healthy test plan, we should have a lot of unit tests: some integration tests and very few E2E tests.

The reason for this is very simple, majority of the problems can be caught within unit testing. Hitting the different branches of our code will verify the functionality of pretty much every functional case in our application, so it makes sense to have plenty of them in our test plan. Based on my experience, in a balanced test plan, around 70% of our tests should be unit tests. However, in a microservices-oriented architecture, especially with a dynamic language such as Node.js, this figure can easily go down and still be effective with our testing. The reasoning behind it is that Node.js allows you to write integration tests very quickly so that we can replace some unit tests by integration tests.

Tip

Testing is a very well-documented, complex process. Trying to outsmart the existing methodologies could result in a hard-to-maintain and difficult-to-trust test suite.

Integration tests are responsible for catching integration problems, as shown in the following:

- Can our code call the SMS gateway?
- Would the connection to the database be OK?
- Are the HTTP headers being sent from our service?

Again, around 20% of our tests, based on my experience, should be integration tests; focus on the

positive flows and some of the negative that depend on third-party modules.

When it comes down to E2E tests, they should be very limited and only test the main flows of the applications without going into too much detail. These details should be already captured by the unit and integration tests that are easy to fix in an event of failure. However, there is a catch here: when testing microservices in Node.js, 90% of the time, integration and E2E tests can be the same thing. Due to the dynamic nature of Node.js, we can test the rest API from the integration point of view (the full server running), but in reality, we will also be testing how our code behaves when integrated with other modules. We will see an example later in this chapter.

Testing microservices in Node.js

Node.js is an impressive language. The amount of libraries around any single aspect of the development is amazing. No matter how bizarre the task that you want to achieve in Node.js is, there will be always an npm module.

Regarding the testing, Node.js has a very powerful set of libraries, but two of them are especially popular: **Mocha** and **Chai**.

They are pretty much the industry standard for app testing and are very well maintained and upgraded.

Another interesting library is called **Sinon.JS**, and it is used for mocking, spying and stubbing methods. We will come back to these concepts in the following sections, but this library is basically used to simulate integrations with third parties without interacting with them.

Chai

This library is a BDD/TDD assertions library that can be used in conjunction with any other library to create high quality tests.

An assertion is a code statement that will either be fulfilled or throw an error, stopping the test and marking it as a failure:

```
5 should be equal to A
```

The preceding statement will be correct when the variable `A` contains the value `5`. This is a very powerful tool to write easy-to-understand tests, and especially with Chai, we have access to assertions making use of the following three different interfaces:

- `should`
- `expect`
- `assert`

At the end of the day, every single condition can be checked using a single interface, but the fact that the library provides us with such a rich interface facilitates the verbosity of the tests in order to write clean, easy, and maintainable tests.

Let's install the library:

```
npm install chai
```

This will produce the following output:

```
├── assertion-error@1.0.1
├── type-detect@1.0.0
└── deep-eql@0.1.3 (type-detect@0.1.1)
```

This means that Chai depends on `assertion-error`, `type-detect`, and `deep-eql`. As you can see, this is a good indication that we will be able to check, with simple instructions, complex statements such as deep equality in objects or type matching.

Testing libraries such as Chai are not a direct dependency of our application, but a development dependency. We need them to develop applications, but they should not be shipped to production. This is a good reason to restructure our `package.json` and add Chai in the `devDependencies` dependency tag, as follows:

```
{
  "name": "chai-test",
  "version": "1.0.0",
  "description": "A test script",
  "main": "chai.js",
  "dependencies": {
  },
  "devDependencies": {
    "chai": "*"
  },
  "author": "David Gonzalez",
  "license": "ISC"
}
```

This will prevent our software to ship into production libraries such as Chai, which has nothing to do with the operation of our application.

Once we have installed Chai, we can start playing around with the interfaces.

BDD-style interfaces

Chai comes with two flavors of BDD interfaces. It is a matter of preference which one to use, but my personal recommendation is to use the one that makes you feel more comfortable in any situation.

Let's start with the `should` Interface. This one is a BDD-style interface, using something similar to the natural language, we can create assertions that will decide whether our test succeeds or fails:

```
myVar.should.be.a('string')
```

In order to be able to build sentences like the one before, we need to import the `should` module in our program:

```
var chai = require('chai');

chai.should();

var foo = "Hello world";
console.log(foo);

foo.should.equal('Hello world');
```

Although it looks like a bit of dark magic, it is really convenient when testing our code as we use something similar to the natural language to ensure that our code is meeting some criteria: *foo should be equal to 'Hello world'* has a direct translation to our test.

The second BDD-style interface provided by Chai is `expect`. Although it is very similar to `should`, it changes a bit of syntax in order to set expectations that the results have to meet.

Let's see the following example:

```
var expect = require('chai').expect;

var foo = "Hello world";

expect(foo).to.equal("Hello world");
```

As you can see, the style is very similar: a fluent interface that allows us to check whether the conditions for the test to succeed are met, but what happens if the conditions are not met?

Let's execute a simple Node.js program that fails in one of the conditions:

```
var expect = require('chai').expect;
var animals = ['cat', 'dog', 'parrot'];
expect(animals).to.have.length(4);
```

Now, let's execute the previous script, assuming that you have already installed Chai:

```
code/node_modules/chai/lib/chai/assertion.js:107
      throw new AssertionError(msg, {
            ^
AssertionError: expected [ 'cat', 'dog', 'parrot' ] to have a length of 4 but
got 3
    at Object.<anonymous> (/Users/dgonzalez/Dropbox/Microservices with
Node/Writing Bundle/Chapter 6/code/chai.js:24:25)
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
    at Module.load (module.js:355:32)
    at Function.Module._load (module.js:310:12)
    at Function.Module.runMain (module.js:501:10)
    at startup (node.js:129:16)
    at node.js:814:3
```

An exception is thrown and the test fails. If all the conditions were validated, no exception would have been raised and the test would have succeeded.

As you can see, there are a number of natural language words that we can use for our tests using both `expect` and `should` interfaces. The full list can be found in the Chai documentation (http://chaijs.com/api/bdd/#-include-value-), but let's explain some of the most interesting ones in the following list:

- `not`: This word is used to negate the assertions following in the chain. For example, `expect("some string").to.not.equal("Other String")` will pass.
- `deep`: This word is one of the most interesting of all the collection. It is used to deep-compare objects, which is the quickest way to carry on a full equal comparison. For example, `expect(foo).to.deep.equal({name: "David"})` will succeed if `foo` is a JavaScript object with one property called `name` with the `"David"` string value.
- `any`/`all`: This is used to check whether the dictionary or object contains any of the keys in the given list so that `expect(foo).to.have.any.keys("name", "surname")` will succeed if `foo` contains any of the given keys, and `expect(foo).to.have.all.keys("name", "surname")` will only succeed if it has all of the keys.
- `ok`: This is an interesting one. As you probably know, JavaScript has a few pitfalls, and one of them is the true/false evaluation of expressions. With `ok`, we can abstract all the mess and do something similar to the following list of expressions:
  - `expect('everything').to.be.ok`: `'everything'` is a string and it will be evaluated to `ok`
  - `expect(undefined).to.not.be.ok`: Undefined is not ok in the JavaScript world, so this assertion will succeed
- `above`: This is a very useful word to check whether an array or collection contains a number of elements above a certain threshold, as follows:
  `expect([1,2,3]).to.have.length.above(2)`

As you can see, the Chai API for fluent assertions is quite rich and enables us to write very descriptive tests that are easy to maintain.

Now, you may be asking yourself, why have two flavors of the same interface that pretty much work the same? Well, they functionally do the same, however, take a look at the detail:

- `expect` provides a starting point in your chainable language
- `should` extends the `Object.prototype` signature to add the chainable language to every single object in JavaScript

From Node.js' point of view, both of them are fine, although the fact that `should` is instrumenting the prototype of `Object` could be a reason to be a bit paranoid about using it as it is intrusive.

Assertions interface

The **assertions** interface matches the most common old-fashioned tests assertion library. In this flavor, we need to be specific about what we want to test, and there is no such thing as fluent chaining of expressions:

```
var assert = require('chai').assert;
var myStringVar = 'Here is my string';
// No message:
assert.typeOf(myStringVar, 'string');
// With message:
assert.typeOf(myStringVar, 'string', 'myStringVar is not string type.');
// Asserting on length:
```

```
assert.lengthOf(myStringVar, 17);
```

There is really nothing more to go in depth if you have already used any of the existing test libraries in any language.

Mocha

Mocha is, in my opinion, one of the most convenient testing frameworks that I have ever used in my professional life. It follows the principles of **behavior-driven development testing** (**BDDT**), where the test describes a use case of the application and uses the assertions from another library to verify the outcome of the executed code.

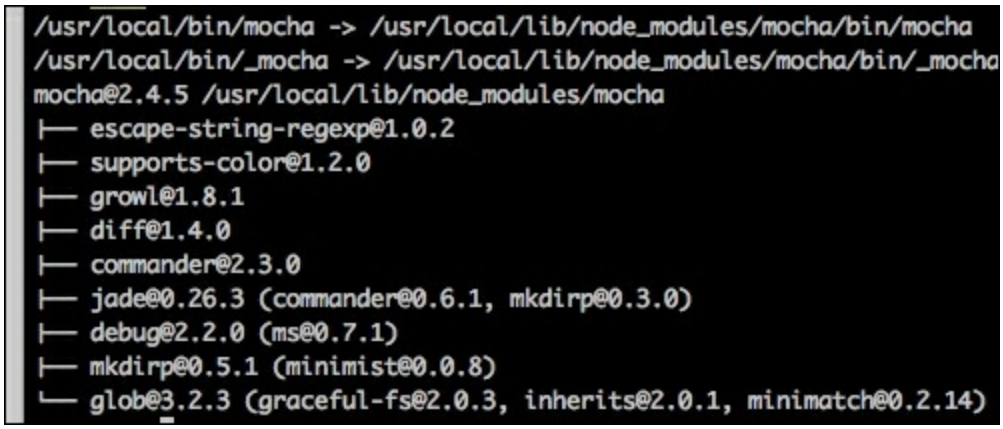
Although it sounds a bit complicated, it is really convenient to ensure that our code is covered from the functional and technical point of view, as we will be mirroring the requirements used to build the application into automated tests that verifies them.

Let's start with a simple example. Mocha is a bit different from any other library, as it defines its own **domain-specific language** (**DSL**) that needs to be executed with Mocha instead of Node.js. It is an extension of the language.

First we need to install Mocha in the system:

```
npm install mocha -g
```

This will produce an output similar to the following image:



From now on, we have a new command in our system: `mocha`.

The next step is to write a test using Mocha:

```
function rollDice() {
  return Math.floor(Math.random() * 6) + 1;
}

require('chai').should();
```

```
var expect = require('chai').expect;

describe('When a customer rolls a dice', function(){

  it('should return an integer number', function() {
    expect(rollDice()).to.be.an('number');
  });

  it('should get a number below 7', function(){
    rollDice().should.be.below(7);
  });

  it('should get a number bigger than 0', function(){
    rollDice().should.be.above(0);
  });

  it('should not be null', function() {
    expect(rollDice()).to.not.be.null;
  });

  it('should not be undefined', function() {
    expect(rollDice()).to.not.be.undefined;
  });
});
```

The preceding example is simple. A function that rolls a dice and returns an integer number from `1` to `6`. Now we need to think a bit about the use cases and the requirements:

- The number has to be an integer
- This integer has to be below 7
- It has to be above 0, dice don't have negative numbers
- The function cannot return `null`
- The function cannot return `undefined`

This covers pretty much every corner case about rolling a dice in Node.js. What we are doing is describing situations that we certainly want to test, in order to safely make changes to the software without breaking the existing functionality.

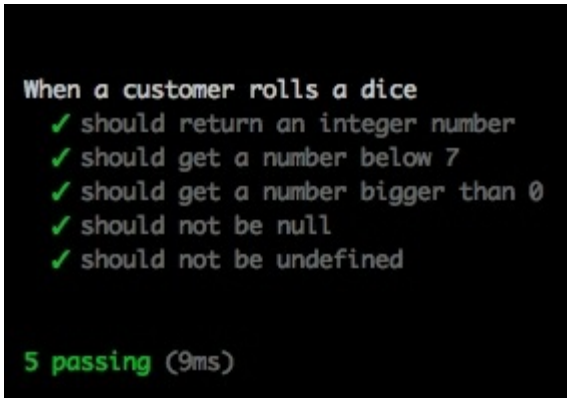
These five use cases are an exact map to the tests written earlier:

- **We describe the situation**: *When a customer rolls a dice*
- **Conditions get verified**: *It should return an integer number*

Let's run the previous test and check the results:

```
mocha tests.js
```

This should return something similar to the following screenshot:



As you can see, Mocha returns a comprehensive report on what is going on in the tests. In this case, all of them pass, so we don't need to be worried about problems.

Let's force some of the tests to fail:

```
function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}

require('chai').should();
var expect = require('chai').expect;

describe('When a customer rolls a dice', function(){

  it('should return an integer number', function() {
    expect(rollDice()).to.be.an('number');
  });

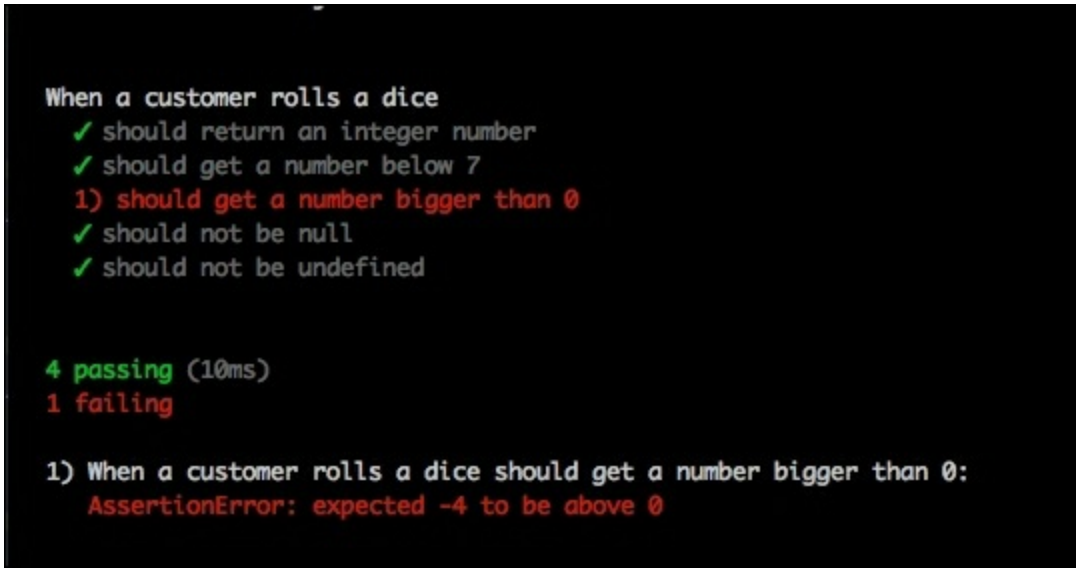
  it('should get a number below 7', function(){
    rollDice().should.be.below(7);
  });

  it('should get a number bigger than 0', function(){
    rollDice().should.be.above(0);
  });

  it('should not be null', function() {
    expect(rollDice()).to.not.be.null;
  });

  it('should not be undefined', function() {
    expect(rollDice()).to.not.be.undefined;
  });
});
```

Accidentally, someone has bumped a code fragment into the `rollDice()` function, which makes the function return a number that does not meet some of the requirements. Let's run Mocha again, as shown in the following image:



Now, we can see the report returning one error: the method is returning `-4`, where it should always return a number bigger than `0`.

Also, one of the benefits of this type of testing in Node.js using Mocha and Chai is the time. Tests run very fast so that it is easy to receive feedback if we have broken something. The preceding suite ran in `10ms`.

Sinon.JS – a mocking framework

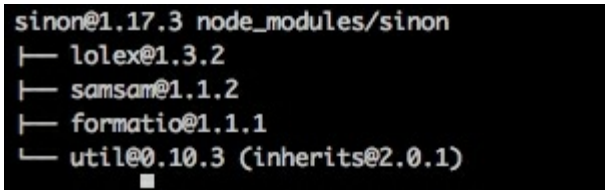
The previous two chapters have been focused on asserting conditions on return values of functions, but what happens when our function does not return any value? The only correct measurement is to check whether the method was called or not. Also, what if one of our modules is calling a third-party web service, but we don't want our tests to call the remote server?

For answering these questions, we have two conceptual tools called mocks and spies, and Node.js has the perfect library to implement them: Sinon.JS.

First install it, as follows:

```
npm install sinon
```

The preceding command should produce the following output:



Now let's explain how it works through an example:

```
function calculateHypotenuse(x, y, callback) {
  callback(null, Math.sqrt(x*x + y*x));
}

calculateHypotenuse(3, 3, function(err, result){
  console.log(result);
});
```

This simple script calculates the hypotenuse of a triangle, given the length of the other two sides of the triangle. One of the tests that we want to carry on is the fact that the callback is executed with the right list of arguments supplied. What we need to accomplish such task is what Sinon.JS calls a spy:

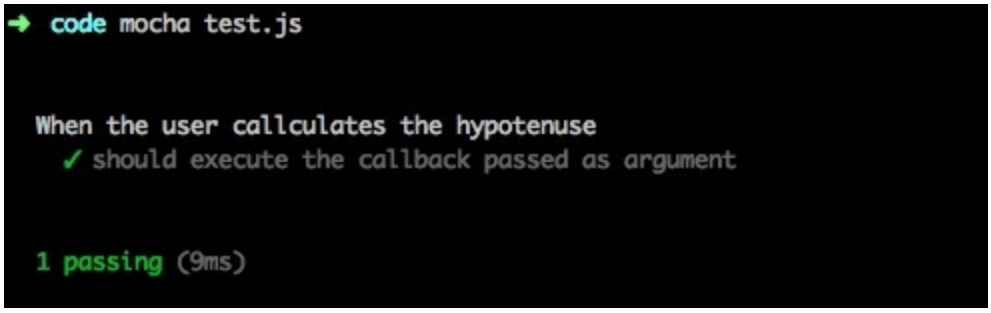
```
var sinon = require('sinon');

require('chai').should();

function calculateHypotenuse(x, y, callback) {
  callback(null, Math.sqrt(x*x + y*x));
}

describe("When the user calculates the hypotenuse", function(){
  it("should execute the callback passed as argument", function() {
    var callback = sinon.spy();
    calculateHypotenuse(3, 3, callback);
    callback.called.should.be.true;
  });
});
```

Once again, we are using Mocha to run the script and Chai to verify the results in the test through the `should` interface, as shown in the following image:



The important line in the preceding script is:

```
var callback = sinon.spy();
```

Here, we are creating the spy and injecting it into the function as a callback. This function created by Sinon.JS is actually not only a function, but a full object with a few interesting points of information. Sinon.JS does that, taking advantage of the dynamic nature of JavaScript. You can actually see what is

in this object by dumping it into the console with `console.log()`.

Another very powerful tool in Sinon.JS are the stubs. **Stubs** are very similar to mocks (identical at practical effects in JavaScript) and allow us to fake functions to simulate the required return:

```
var sinon = require('sinon');
var expect = require('chai').expect;

function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}
describe("When rollDice gets called", function() {
  it("Math#random should be called with no arguments", function() {
    sinon.stub(Math, "random");
    rollDice();
    console.log(Math.random.calledWith());
  });
})
```

In this case, we have stubbed the `Math#random` method, which causes the method to be some sort of overloaded empty function (it does not issue the get call) that records stats on what or how it was called.

There is one catch in the preceding code: we never restored the `random()` method back and this is quite dangerous. It has a massive side effect, as other tests will see the `Math#random` method as a stub, not as the original one, and it can lead to us coding our tests according to invalid information.

In order to prevent this, we need to make use of the `before()` and `after()` methods from Mocha:

```
var sinon = require('sinon');
var expect = require('chai').expect;

var sinon = require('sinon');
var expect = require('chai').expect;

function rollDice() {
  return -1 * Math.floor(Math.random() * 6) + 1;
}
describe("When rollDice gets called", function() {

  it("Math#random should be called with no arguments", function() {
    sinon.stub(Math, "random");
    rollDice();
    console.log(Math.random.calledWith());
  });
after(function(){
    Math.random.restore();
  });
});
```

If you pay attention to the highlighted code, we are telling Sinon.JS to restore the original method that

was stubbed inside one of the `it` blocks, so that if another `describe` block makes use of `http.get`, we won't see the stub, but the original method.

Tip

The `before()` and `after()` methods are very helpful to set up and wind down the context for the tests. However, you need to be careful with the scope where they are executed as it could lead to test interactions.

Mocha has a few flavors of before and after:

- `before(callback)`: This is executed before the current scope (at the beginning of the `describe` block in the preceding code)
- `after(callback)`: This is executed after the current scope (at the end of the `describe` block in the preceding code)
- `beforeEach(callback)`: This is executed at the beginning of every element in the current scope (before each `it` in the preceding example)
- `afterEach(callback)`: This is executed at the end of every element in the current scope (after every `it` in the preceding example)

Another interesting feature in Sinon.JS is the time manipulation. Some of the tests need to execute periodic tasks or respond after a certain time of an event's occurrence. With Sinon.JS, we can dictate time as one of the parameters of our tests:

```
var sinon = require('sinon');
var expect = require('chai').expect

function areWeThereYet(callback) {

  setTimeout(function() {
    callback.apply(this);
  }, 10);

}

var clock;

before(function(){
  clock = sinon.useFakeTimers();
});

it("callback gets called after 10ms", function () {
  var callback = sinon.spy();
  var throttled = areWeThereYet(callback);

  areWeThereYet(callback);

  clock.tick(9);
  expect(callback.notCalled).to.be.true;

  clock.tick(1);
```

```
  expect(callback.notCalled).to.be.false;
});

after(function(){
  clock.restore();
});
```

As you can see, we can now control the time in our tests.

Testing a real microservice

Now, it is time to test a real microservice in order to get a general picture of the full test suite.

Our microservice is going to use Express, and it will filter an input text to remove what the search engines call **stop words**: *words with less than three characters and words that are banned*.

Let's see the code:

```
var _ = require('lodash');
var express = require('express');

var bannedWords = ["kitten", "puppy", "parrot"];

function removeStopWords (text, callback) {
  var words = text.split(' ');
  var validWords = [];
  _(words).forEach(function(word, index) {
    var addWord = true;

    if (word.length < 3) {
      addWord = false;
    }

    if(addWord && bannedWords.indexOf(word) > -1) {
      addWord = false;
    }

    if (addWord) {
      validWords.push(word);
    }

    // Last iteration:
    if (index == (words.length - 1)) {
      callback(null, validWords.join(" "));
    }
  });
}
var app = express();

app.get('/filter', function(req, res) {
  removeStopWords(req.query.text, function(err, response){
    res.send(response);
  });
```

```
});

app.listen(3000, function() {
  console.log("app started in port 3000");
});
```

As you can see, the service is pretty small, so it is the perfect example for explaining how to write unit, integration, and E2E tests. In this case, as we stated before, E2E and integration tests are going to be the exact same as testing the service through the REST API will be equivalent to testing the system from the end-to-end point of view, but also how our component is integrated within the system. Given that, if we were to add a UI, we would have to split integration tests from E2E in order to ensure the quality.

TDD – Test-driven development

Our service is done and working. However, now we want to unit test it, but we find some problems:

- The function that we want to unit test is not visible outside the main `.js` file
- The server code is tightly coupled to the functional code and has bad cohesion

Here TDD comes to the rescue; we should always ask ourselves "how am I going to test this function when writing software?" It does not mean that we should modify our software with the specific purpose of testing, but if you are having problems while testing a part of your program, more than likely, you should look into cohesion and coupling, as it is a good indication of problems. Let's take a look at the following file:

```
var _ = require('lodash');
var express = require('express');

module.exports = function(options) {
  bannedWords = [];
  if (typeof options !== 'undefined') {
    console.log(options);
    bannedWords = options.bannedWords || [];
  }

  return function bannedWords(text, callback) {
    var words = text.split(' ');
    var validWords = [];
    _(words).forEach(function(word, index) {
      var addWord = true;

      if (word.length < 3) {
        addWord = false;
      }

      if(addWord && bannedWords.indexOf(word) > -1) {
        addWord = false;
      }

      if (addWord) {
```

```
        validWords.push(word);
      }

      // Last iteration:
      if (index == (words.length - 1)) {
        callback(null, validWords.join(" "));
      }
    });
  }
}
```

This file is a module that, in my opinion, is highly reusable and has good cohesion:

- We can import it everywhere (even in a browser)
- The banned words can be injected when creating the module (very useful for testing)
- It is not tangled with the application code

Laying down the code this way, our application module will look similar to the following:

```
var _ = require('lodash');
var express = require('express');

var removeStopWords = require('./remove-stop-words')({bannedWords: ["kitten",
"puppy", "parrot"]});

var app = express();

app.get('filter', function(req, res) {
  res.send(removeStopWords(req.query.text));
});

app.listen(3000, function() {
  console.log("app started in port 3000");
});
```

As you can see, we have clearly separated the business unit (the function that captures the business logic) from the operational unit (the setup of the server).

As I mentioned before, I am not a big fan of writing the tests prior to the code, but they should be written (in my opinion) alongside the code, but always having in mind the question mentioned before.

There seem to be a push in companies to adopt a TDD methodology, but it could lead to a significant inefficiency, especially if the business requirements are unclear (as they are 90% of the time) and we face changes along the development process.

Unit testing

Now that our code is in a better shape, we are going to unit test our function. We will use Mocha and Chai to accomplish such task:

```
var removeStopWords = require('./remove-stop-words')({bannedWords: ["kitten",
```

```
"parrot"]});

var chai = require('chai');
var assert = chai.assert;
chai.should();
var expect = chai.expect;

describe('When executing "removeStopWords"', function() {

  it('should remove words with less than 3 chars of length', function() {
    removeStopWords('my small list of words', function(err, response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should remove extra white spaces', function() {
    removeStopWords('my small       list of words', function(err, response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should remove banned words', function() {
    removeStopWords('My kitten is sleeping', function(err, response) {
      expect(response).to.equal("sleeping");
    });
  });

  it('should not fail with null as input', function() {
    removeStopWords(null, function(err, response) {
      expect(response).to.equal("small list words");
    });
  });

  it('should fail if the input is not a string', function() {
    try {
      removeStopWords(5, function(err, response) {});
      assert.fail();
    }
    catch(err) {
    }
  });
});
```

As you can see, we have covered pretty much every single case and branch inside our application, but how is our code coverage looking?

Until now, we have mentioned it, but never actually measured it. We are going to use one tool, called **Istanbul**, to measure the test coverage:

```
npm install -g istanbul
```

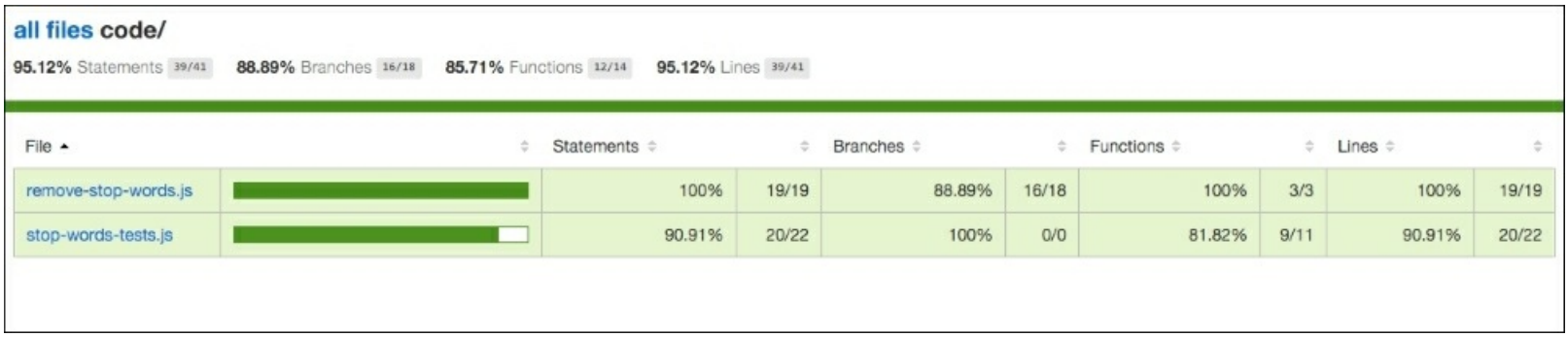
This should install Istanbul. Now we need to run the coverage report:

```
istanbul cover _mocha my-tests.js
```

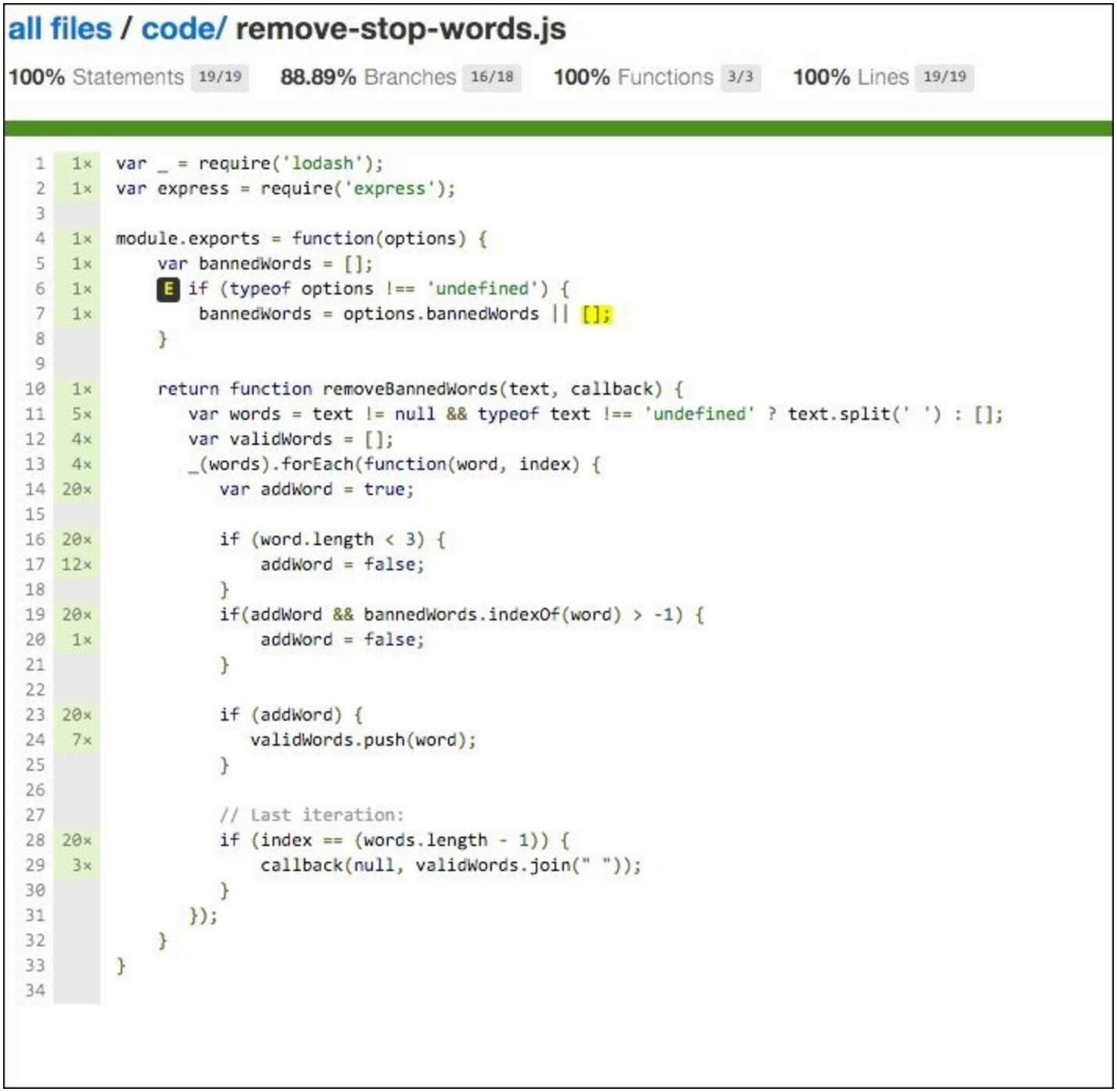
This will produce an output similar to the one shown in the following image:



This will also generate a coverage report in HTML, pointing out which lines, functions, branches, and statements are not being covered, as shown in the following screenshot:



As you can see, we are looking pretty well. Our code (not the tests) is actually well covered, especially if we look into the detailed report for our code file, as shown in the following image:



We can see that only one branch (the `or` operator in line **7**) is not covered and the `if` operator in line **6** never diverted to the `else` operator.

We also got information about the number of times a line is executed: it is showing in the vertical bar beside the line number. This information is also very useful to spot the hot areas of our application where an optimization will benefit the most.

Regarding the right level of coverage, in this example, it is fairly easy to go up to 90%+, but unfortunately, it is not that easy in production systems:

- Code is a lot more complex
- Time is always a constraint

- Testing might not be seen as productive time

However, you should exercise caution when working with a dynamic language. In Java or C#, calling a function that does not exist results in a compilation time error; whereas in JavaScript, it will result in a runtime error. The only real barrier is the testing (manual or automated), so it is a good practice to ensure that at least every line is executed once. In general code coverage, over 75% should be good enough for the majority of cases.

End-to-end testing

In order to test our application end to end, we are going to need a server running it. Usually, end-to-end tests are executed against a controlled environment, such as a QA box or a pre-production machine, to verify that our about-to-be-deployed software is behaving as expected.

In this case, our application is an API, so we are going to create the end-to-end tests, which at the same time, are going to be used as integration tests.

However, in a full application, we might want to have a clear separation between the integration and end-to-end tests and use something like Selenium to test our application from the UI point of view.

Selenium is a framework that allows our code to send instructions to the browser, as follows:

- Click the button with the `button1` ID
- Hover over the `div` element with the CSS class `highlighted`

In this way, we can ensure that our app flows work as expected, end to end, and our next release is not going to break the key flows of our app.

Let's focus on the end-to-end tests for our microservice. We have been using Chai and Mocha with their corresponding assertion interfaces to unit test our software, and Sinon.JS to mock services functions and other elements to avoid the calls being propagated to third-party web services or get a controlled response from one method.

Now, in our end-to-end test plan, we actually want to issue the calls to our service and get the response to validate the results.

The first thing we need to do is run our microservice somewhere. We are going to use our local machine just for convenience, but we can execute these tests in a continuous development environment against a QA machine.

So, let's start the server:

```
node stop-words.js
```

I call my script `stop-words.js` for convenience. Once the server is running, we are ready to start testing. In some situations, we might want our test to start and stop the server so that everything is self-contained. Let's see a small example about how to do it:

```
var express = require('express');

var myServer = express();

var chai = require('chai');

myServer.get('/endpoint', function(req, res){
  res.send('endpoint reached');
});

var serverHandler;

before(function(){
  serverHandler = myServer.listen(3000);
});

describe("When executing 'GET' into /endpoint", function(){
  it("should return 'endpoint reached'", function(){
    // Your test logic here. http://localhost:3000 is your server.
  });
});

after(function(){
  serverHandler.close();
});
```

As you can see, Express provides a handler to operate the server programmatically, so it is as simple as making use of the `before()` and `after()` functions to do the trick.

In our example, we are going to assume that the server is running. In order to issue the requests, we are going to use a library called `request` to issue the calls to the server.

The way to install it, as usual, is to execute `npm install request`. Once it is finished, we can make use of this amazing library:

```
var chai = require('chai');
var chaiHttp = require('chai-http');
var expect = chai.expect;
chai.use(chaiHttp);

describe("when we issue a 'GET' to /filter with text='aaaa bbbb cccc'",
function(){
  it("should return HTTP 200", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.status).to.equal(200);
        done();
      });
  });
});
```

```
describe("when we issue a 'GET' to /filter with text='aa bb ccccc'", function(){
  it("should return 'ccccc'", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.text).to.equal('ccccc');
        done();
      });
  });
});

describe("when we issue a 'GET' to /filter with text='aa bb cc'", function(){
  it("should return ''", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb cc'}).end(function(req, res){
        expect(res.text).to.equal('');
        done();
      });
  });
});
```

With the simple test from earlier, we managed to test our server in a way that ensures that every single mobile part of the application has been executed.

There is a particularity here that we didn't have before:

```
  it("should return 'ccccc'", function(done) {
    chai.request('http://localhost:3000')
      .get('/filter')
      .query({text: 'aa bb ccccc'}).end(function(req, res){
        expect(res.text).to.equal('ccccc');
        done();
      });
  });
```

If you take a look at the highlighted code, you can see a new callback called `done`. This callback has one mission: prevent the test from finishing until it is called, so that the HTTP request has time to be executed and return the appropriated value. Remember, Node.js is asynchronous, there is no such thing as a thread being blocked until one operation finishes.

Other than that, we are using a new DSL introduced by `chai-http` to build get requests.

This language allows us to build a large range of combinations, consider the following, for example:

```
chai.request('http://mydomain.com')
  .post('/myform')
  .field('_method', 'put')
  .field('username', 'dgonzalez')
  .field('password', '123456').end(...)
```

In the preceding request, we are submitting a form that looks like a login, so that in the `end()`

function, we can assert the return from the server.

There are an endless number of combinations to test our APIs with `chai-http`.

Manual testing – the necessary evil

No matter how much effort we put in to our automated testing, there will always be a number of manual tests executed.

Sometimes, we need to do it just when we are developing our API, as we want to see the messages going from our client to the server, but some other times, we just want to hit our endpoints with a pre-forged request to cause the software to execute as we expect.

In the first case, we are going to take the advantage of Node.js and its dynamic nature to build a proxy that will sniff all the requests and log them to a terminal so that we can debug what is going on. This technique can be used to leverage the communication between two microservices and see what is going on without interrupting the flow.

In the second case, we are going to use software called Postman to issue requests against our server in a controlled way.

Building a proxy to debug our microservices

My first contact with Node.js was exactly due to this problem: two servers sending messages to each other, causing misbehavior without an apparent cause.

It is a very common problem that has many already-working solutions (man-in-the-middle proxies basically), but we are going to demonstrate how powerful Node.js is:

```
var http = require('http');
var httpProxy = require('http-proxy');
var proxy = httpProxy.createProxyServer({});

http.createServer(function(req, res) {
  console.log(req.rawHeaders);
  proxy.web(req, res, { target: 'http://localhost:3000' });
}).listen(4000);
```

If you remember from the previous section, our `stop-words.js` program was running on the port `3000`. What we have done with this code is create a proxy using `http-proxy`, that tunnels all the requests made on the port `4000` into the port `3000` after logging the headers into the console.

If we run the program after installing all the dependencies with the `npm install` command in the root of the project, we can see how effectively the proxy is logging the requests and tunneling them into the target host:

```
curl http://localhost:4000/filter?text=aaa
```

This will produce the following output:

```
➜ code  node proxy.js
[ 'Host',
  'localhost:4000',
  'User-Agent',
  'curl/7.43.0',
  'Accept',
  '*/*' ]
```

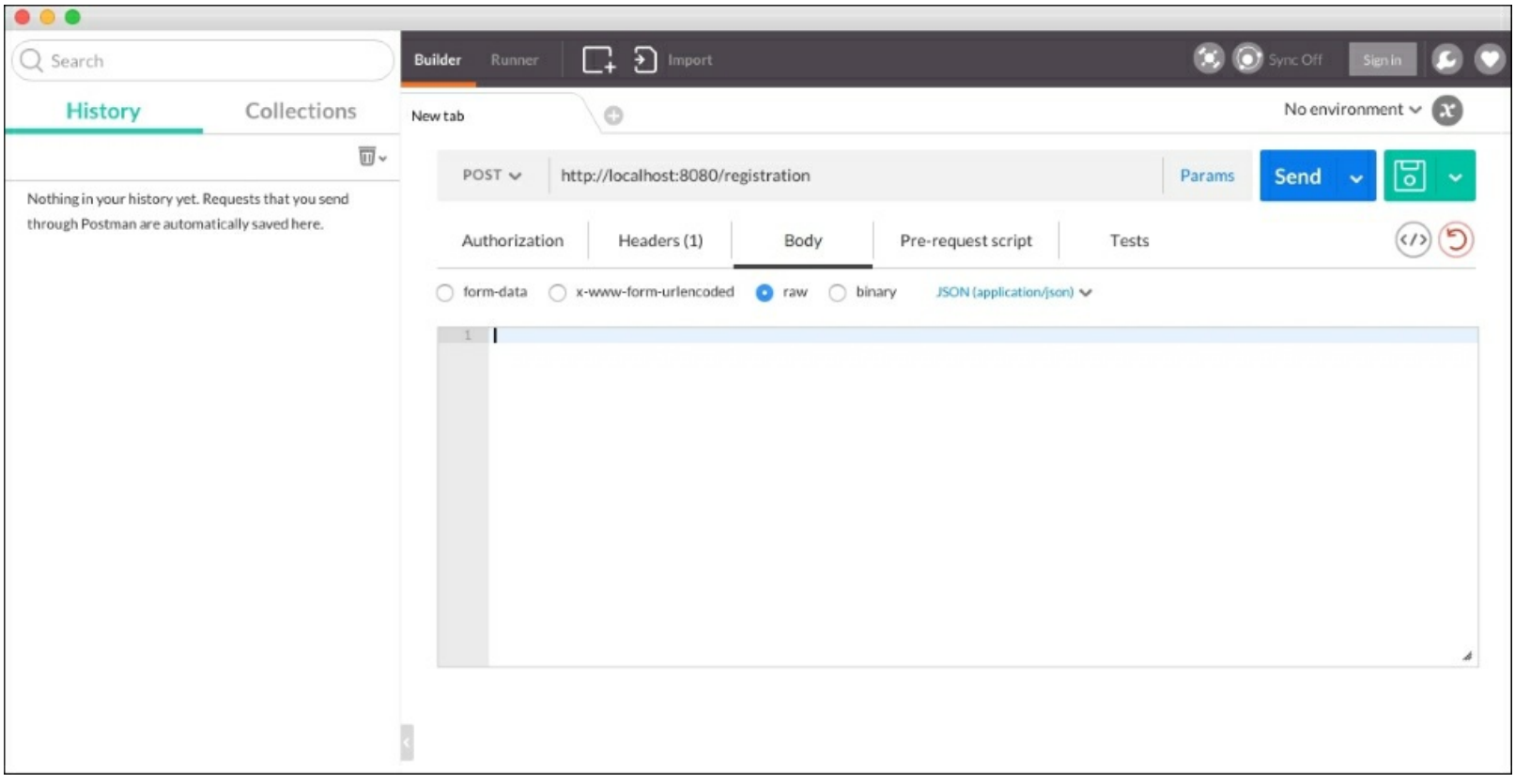
This example is very simplistic, but this small proxy could virtually be deployed anywhere in between our microservices and give us very valuable information about what is going on in the network.

Postman

Out of all the software that we can find on the Internet for testing APIs, Postman is my favorite. It started as a extension for Google Chrome, but nowadays, has taken the form of a standalone app built on the Chrome runtime.

It can be found in the Chrome web store, and it is free (so you don't need to pay for it), although it has a version for teams with more advanced features that is paid.

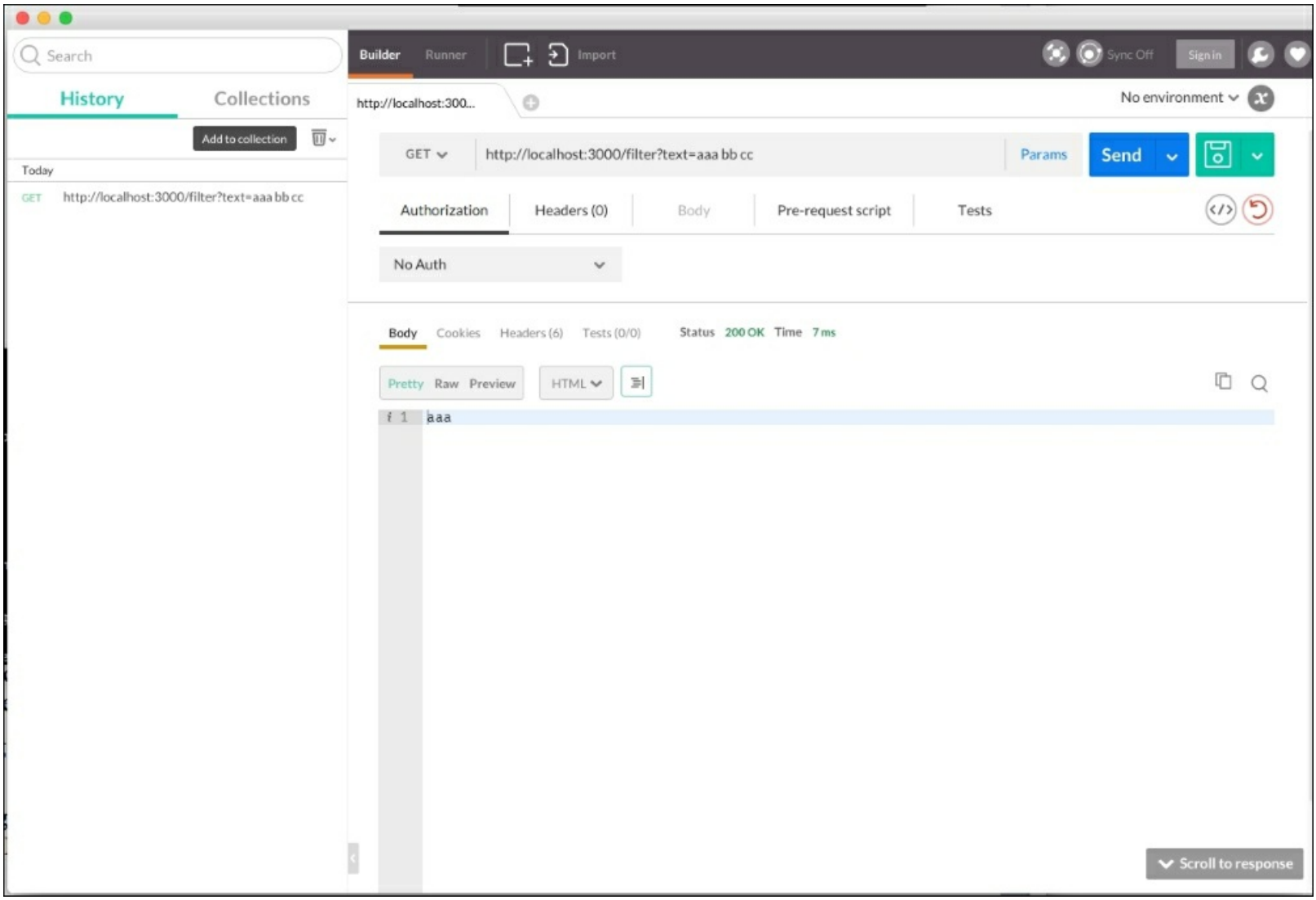
The interface is very concise and simple, as shown in the following screenshot:



On the left-hand side, we can see the **History** of requests, as well as the **Collections** of requests, which will be very handy for when we are working on a long-term project and we have some complicated requests to be built.

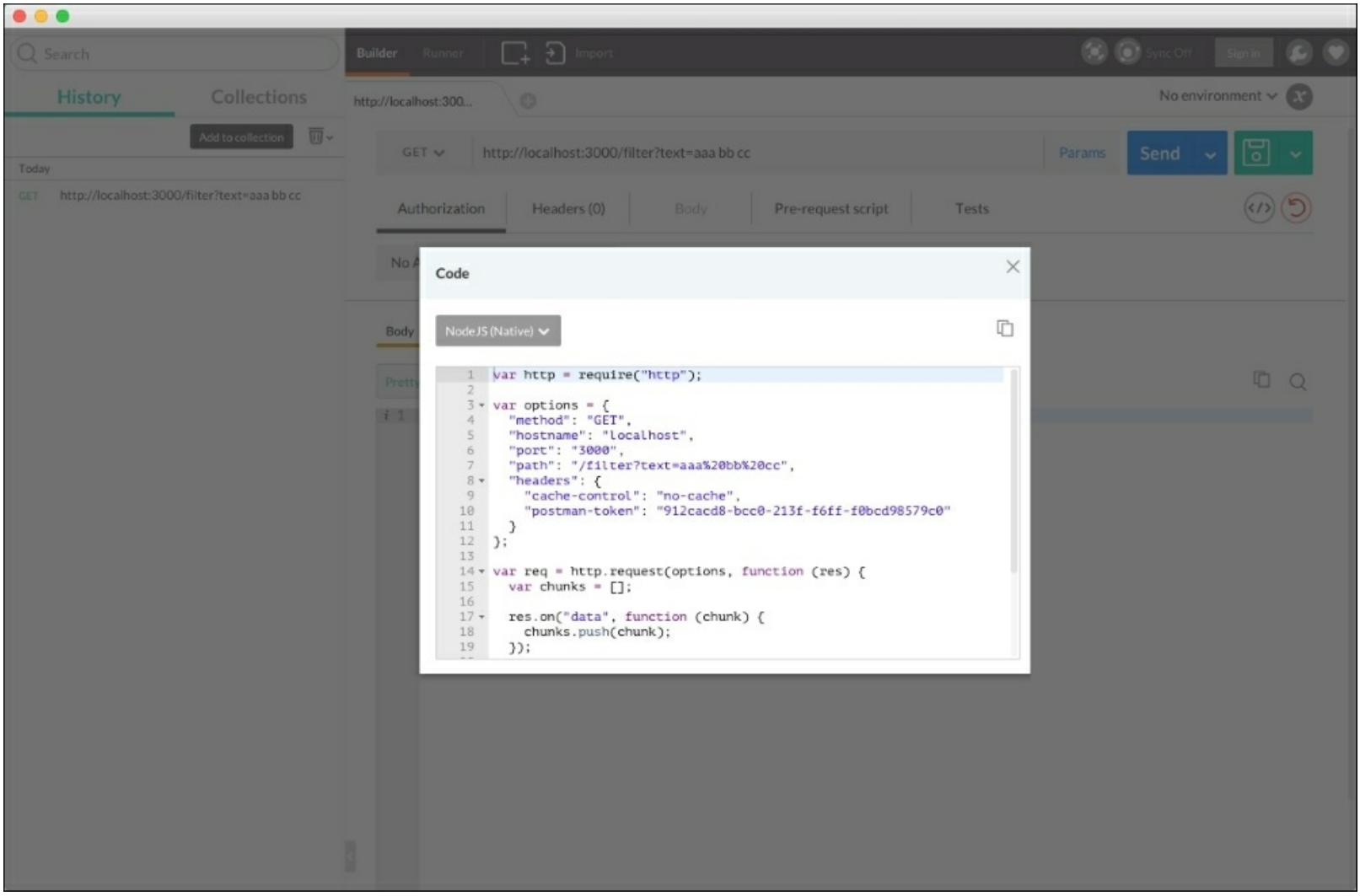
We are going to use again our `stop-words.js` microservice to show how powerful Postman can be.

Therefore, the first thing is to make sure that our microservice is running. Once it is, let's issue a request from Postman, as shown in the following screenshot:



As simple as that, we have issued the request for our service (using the **GET** verb) and it has replied with the text filtered: very simple and effective.

Now imagine that we want to execute that call over Node.js. Postman comes with a very interesting feature, which is generating the code for the requests that we issue from the interface. If you click on the icon under the save button on the right-hand side of the window, the appearing screen will do the magic:



Let's take a look at the generated code:

```
var http = require("http");

var options = {
  "method": "GET",
  "hostname": "localhost",
  "port": "3000",
  "path": "/filter?text=aaa%20bb%20cc",
  "headers": {
    "cache-control": "no-cache",
    "postman-token": "912cacd8-bcc0-213f-f6ff-f0bcd98579c0"
  }
};

var req = http.request(options, function (res) {
  var chunks = [];

  res.on("data", function (chunk) {
    chunks.push(chunk);
  });
```

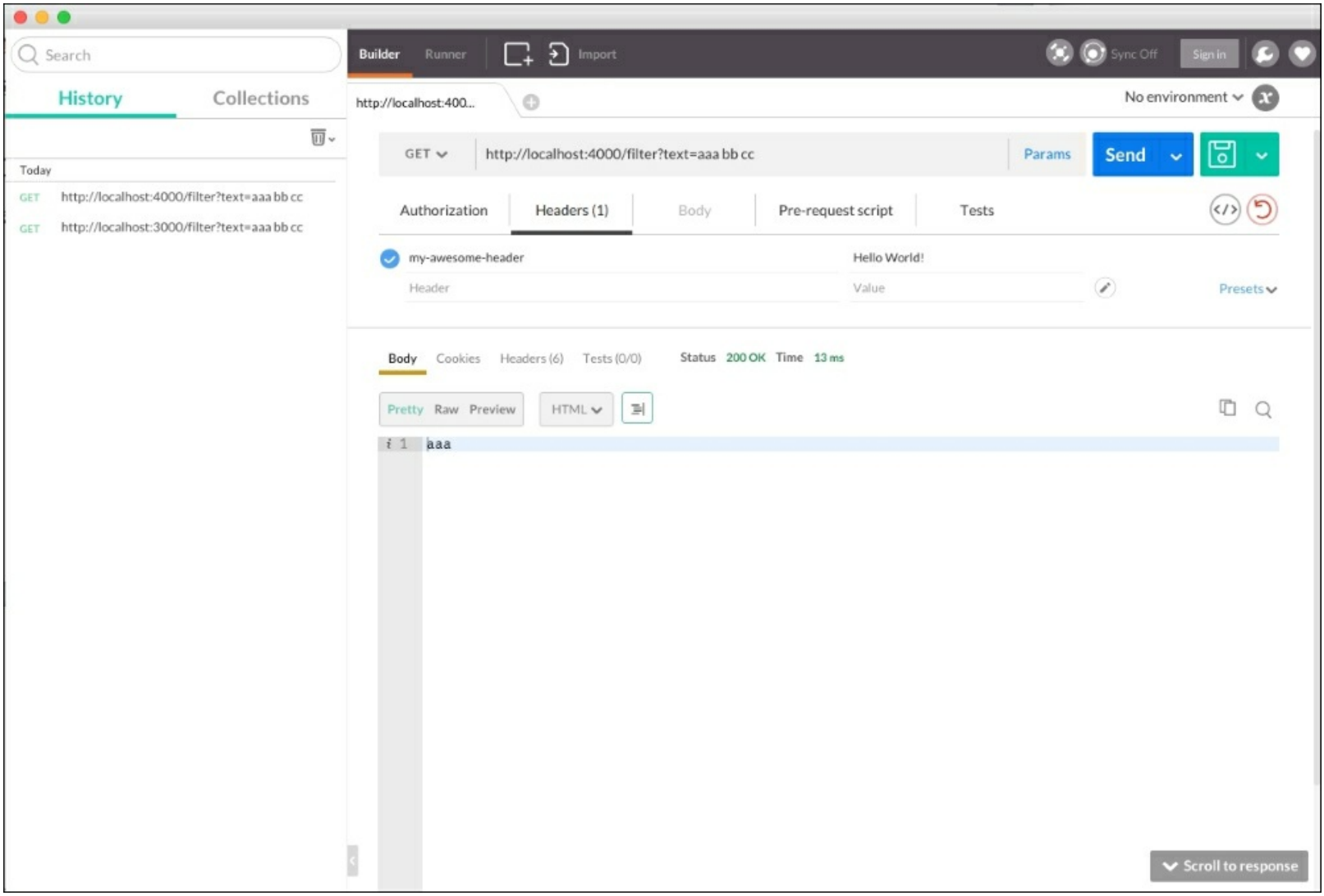
```
  res.on("end", function () {
    var body = Buffer.concat(chunks);
    console.log(body.toString());
  });
});

req.end();
```

It is quite an easy code to understand, especially if you are familiar with the HTTP library.

With Postman, we can also send cookies, headers, and forms to the servers in order to mimic the authentication that an application will fulfill by sending the authentication token or cookie across.

Let's redirect our request to the proxy that we created in the preceding section, as shown in the following screenshot:



If you have the proxy and the `stop-words.js` microservice running, you should see something similar to the following output in the proxy:

```
➜  code  node proxy.js
[ 'Host',
  'localhost:4000',
  'User-Agent',
  'curl/7.43.0',
  'Accept',
  '*/*' ]
[ 'Host',
  'localhost:4000',
  'Connection',
  'keep-alive',
  'Cache-Control',
  'no-cache',
  'my-awesome-header',
  'Hello World!',
  'User-Agent',
  'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.116 Safari/537.36',
  'Postman-Token',
  '9381e356-2d24-8ffe-82c6-6b16c837af18',
  'Accept',
  '*/*',
  'Accept-Encoding',
  'gzip, deflate, sdch',
  'Accept-Language',
  'es-ES,es;q=0.8,en;q=0.6' ]
```

The header that we sent over with Postman, **my-awesome-header**, will show up in the list of raw headers.

Documenting microservices

In this section, we are going to learn how to use Swagger to document APIs. Swagger is an API manager that follows the **Open API standard**, so that it is a *common language* for all the API creators. We will discuss how to write definitions and why it is so important to agree on how to describe resources.

Documenting APIs with Swagger

Documentation is always a problem. No matter how hard you try, it will always eventually go out of date. Luckily, in the past few years, there has been a push into producing a high quality documentation for REST APIs.

API managers have played a key role in it, and Swagger is particularly an interesting platform to look at. More than a module for documentation, Swagger manages your API in a such way that gives you a holistic view of your work.

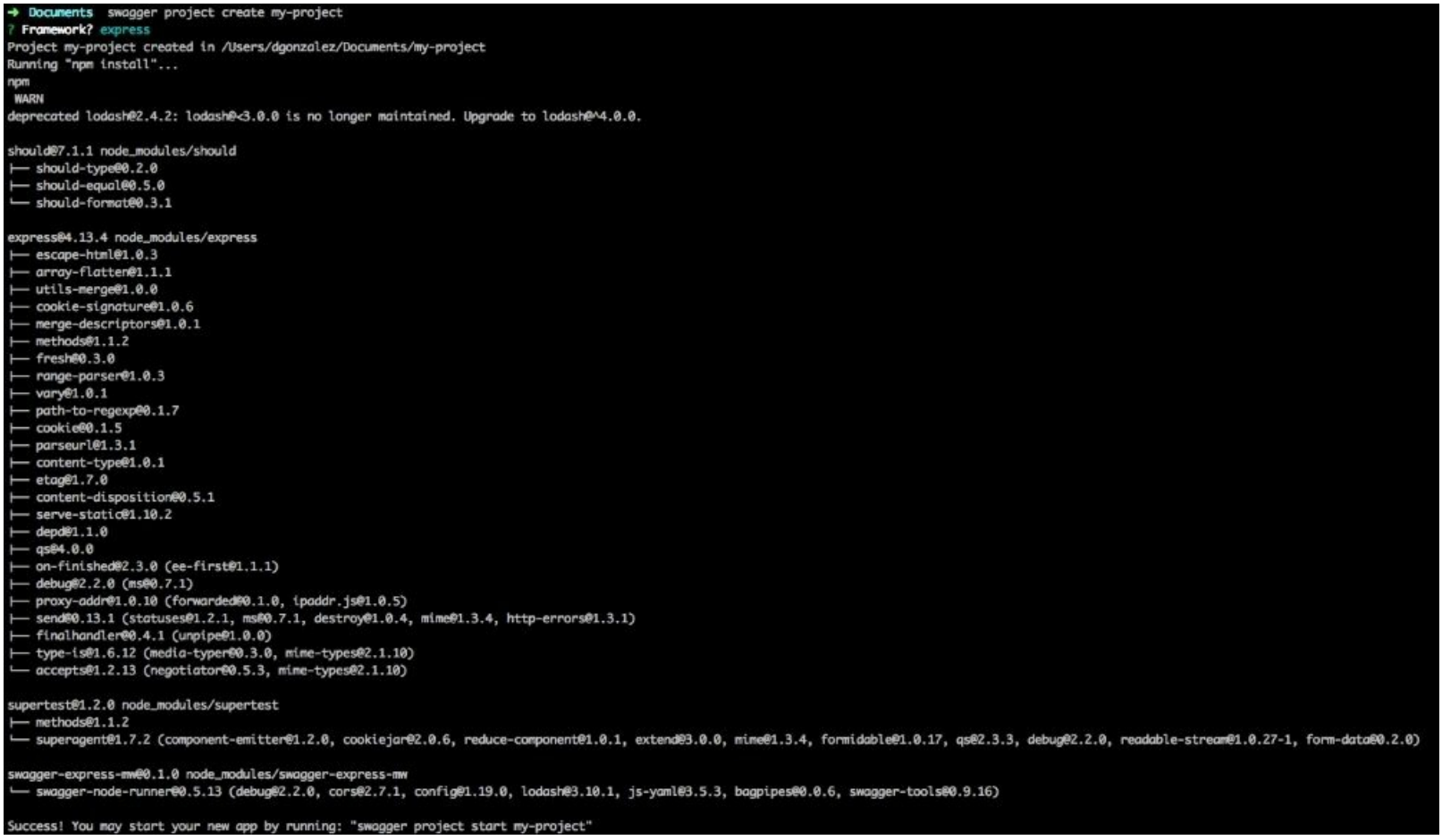
Let's start installing it:

```
npm install -g swagger
```

This will install Swagger system-wide, so it will be another command in our system. Now, we need to create a project using it:

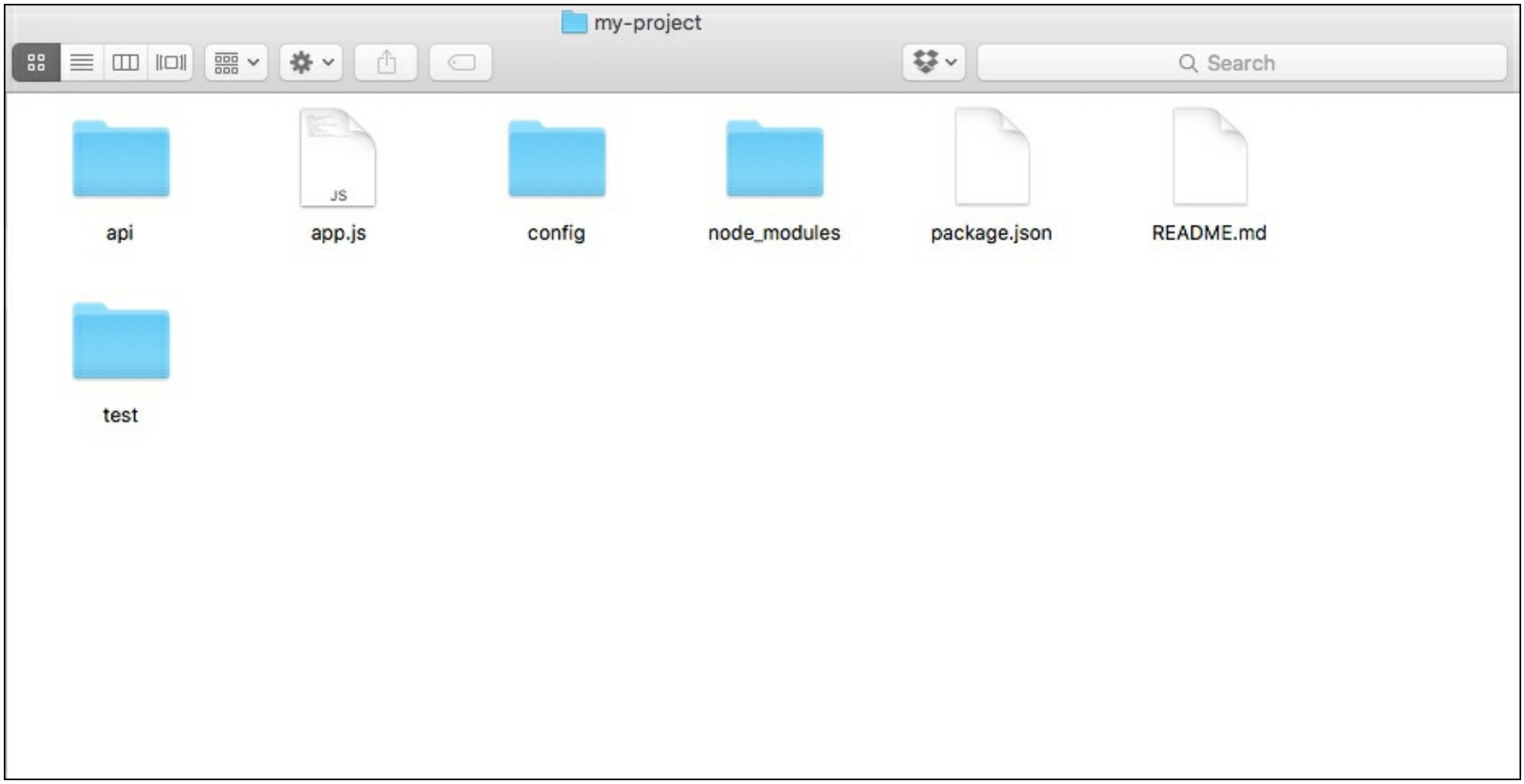
```
swagger project create my-project
```

This command will allow you to choose different web frameworks. We are going to choose Express, as it is the one that we have already been using. The output of the preceding command is shown in the following screenshot:



This screenshot is showing how to start a project with Swagger

Now we can find a new folder, called `my-project`, that looks like the following image:



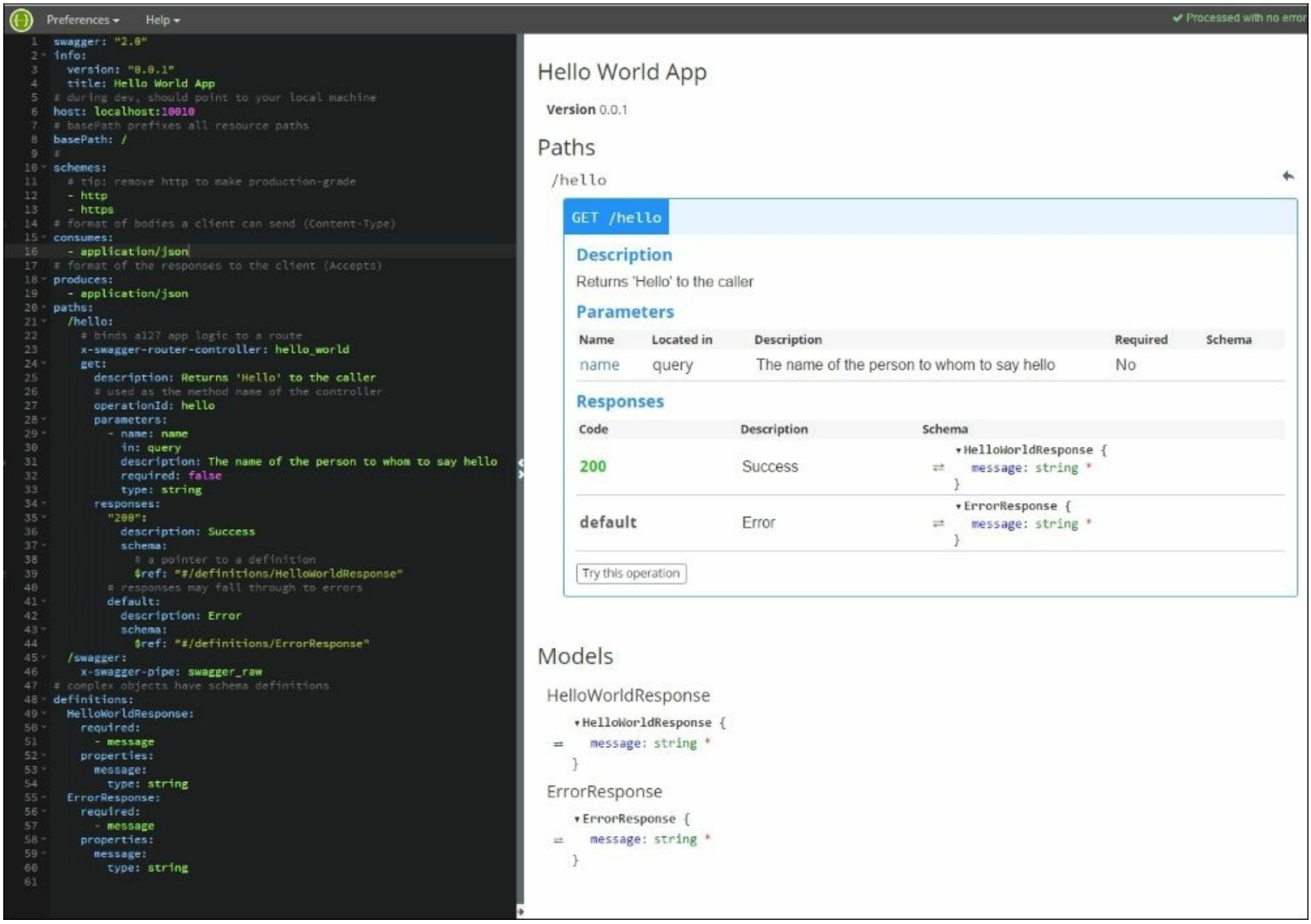
The structure is self-explanatory and it is the common layout of a Node.js application:

- `api`: Here, our API code will lay down
- `config`: All the configuration sits in here
- `node_modules`: This is a folder with all the dependencies required to run our application
- `test`: This is where Swagger has generated some dummy tests and where we could add our own tests

Swagger comes with an impressive feature: an embedded editor that allows you to model the endpoints of your API. In order to run it, from within the generated folder, execute the following command:

```
Swagger project edit
```

It will open Swagger Editor in the default browser, with a window similar to the following image:



Swagger makes use of **Yet Another Markup Language** (**YAML**). It is a language that is very similar to **JSON**, but with a different syntax.

In this document, we can customize a number of things, such as paths (routes in our application). Let's take a look at the path generated by Swagger:

```
/hello:
  # binds a127 app logic to a route
  x-swagger-router-controller: hello_world
  get:
    description: Returns 'Hello' to the caller
    # used as the method name of the controller
    operationId: hello
    parameters:
      - name: name
        in: query
        description: The name of the person to whom to say hello
        required: false
        type: string
    responses:
```

```
      "200":
        description: Success
        schema:
          # a pointer to a definition
          $ref: "#/definitions/HelloWorldResponse"
      # responses may fall through to errors
      default:
        description: Error
        schema:
          $ref: "#/definitions/ErrorResponse"
```

The definition is self-documented. Basically, we will configure the parameters used by our endpoint, but in a declarative way. This endpoint is mapping the incoming actions into the `hello_world` controller, and specifically into the `hello` method, which is defined by the `id` operation. Let's see what Swagger has generated for us in this controller:

```
'use strict';

var util = require('util');

module.exports = {
  hello: hello
};

function hello(req, res) {
  var name = req.swagger.params.name.value || 'stranger';
  var hello = util.format('Hello, %s!', name);
  res.json(hello);
}
```

This code can be found in the `api/controllers` folder of the project. As you can see, it is a pretty standard Express controller packed as a module (well-cohesioned). The only strange line is the first one in the `hello` function, where we pick up the parameters from Swagger. We will come back to this later, once we run the project.

The second part of the endpoint is the responses. As we can see, we are referencing two definitions: `HelloWorldResponse` for `http code 200` and `ErrorResponse` for the rest of the codes. These objects are defined in the following code:

```
definitions:
  HelloWorldResponse:
    required:
      - message
    properties:
      message:
        type: string
  ErrorResponse:
    required:
      - message
    properties:
      message:
        type: string
```

This is something really interesting, although we are using a dynamic language, the contract is being defined by Swagger so that we have a language-agnostic definition that can be consumed by a number of different technologies, respecting the principle of technology heterogeneity that we were talking about earlier in Chapter 1, *Microservices Architecture*, and Chapter 2, *Microservices in Node.js – Seneca and PM2 Alternatives*.

After explaining how the definition works, it is time to start the server:

```
swagger project start
```

This should produce an output that is very similar to the following code:

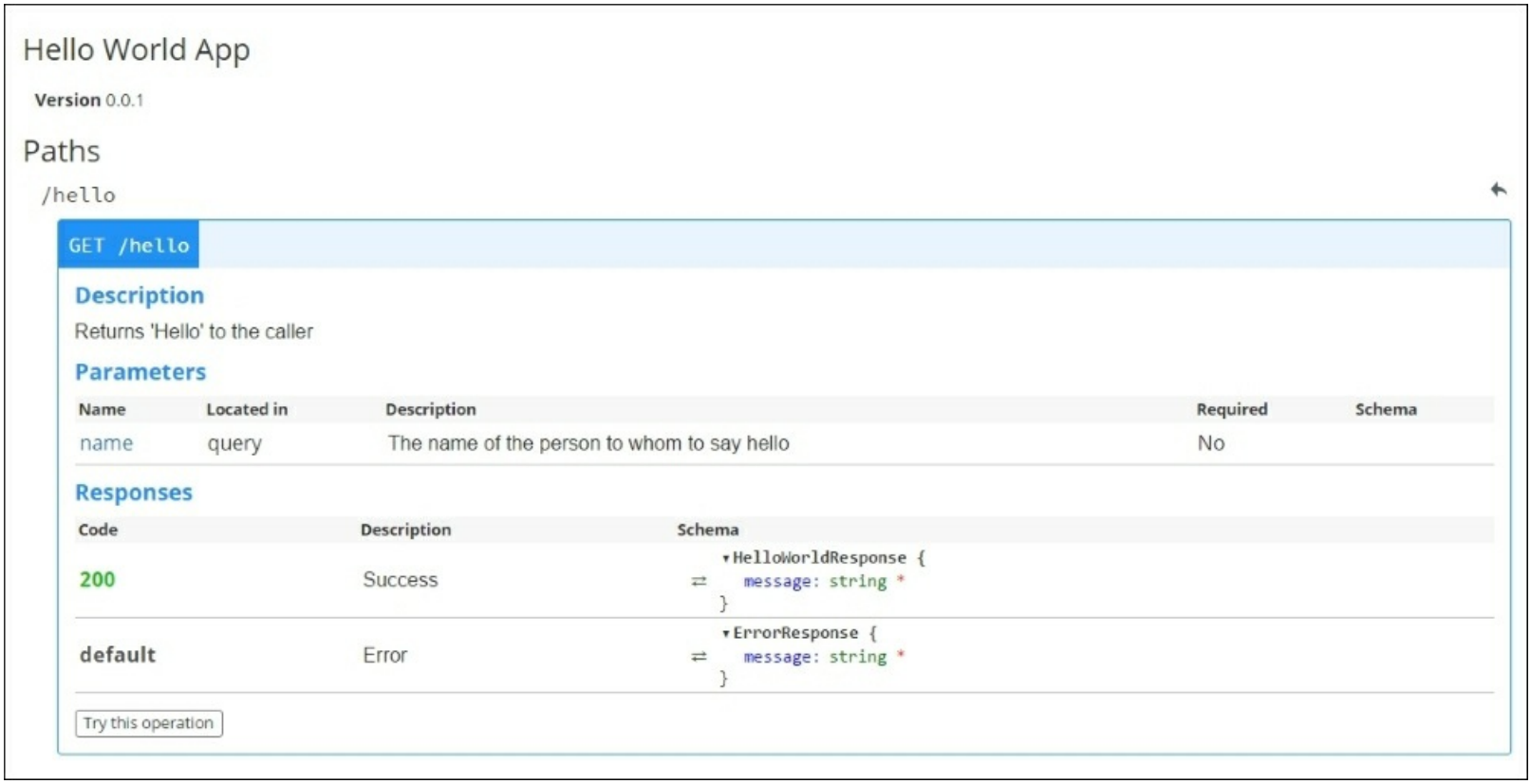
```
Starting: C:\my-project\app.js...
  project started here: http://localhost:10010/
  project will restart on changes.
  to restart at any time, enter `rs`
try this:
curl http://127.0.0.1:10010/hello?name=Scott
```

Now, if we follow the instructions of the output and execute the curl command, we get the following output:

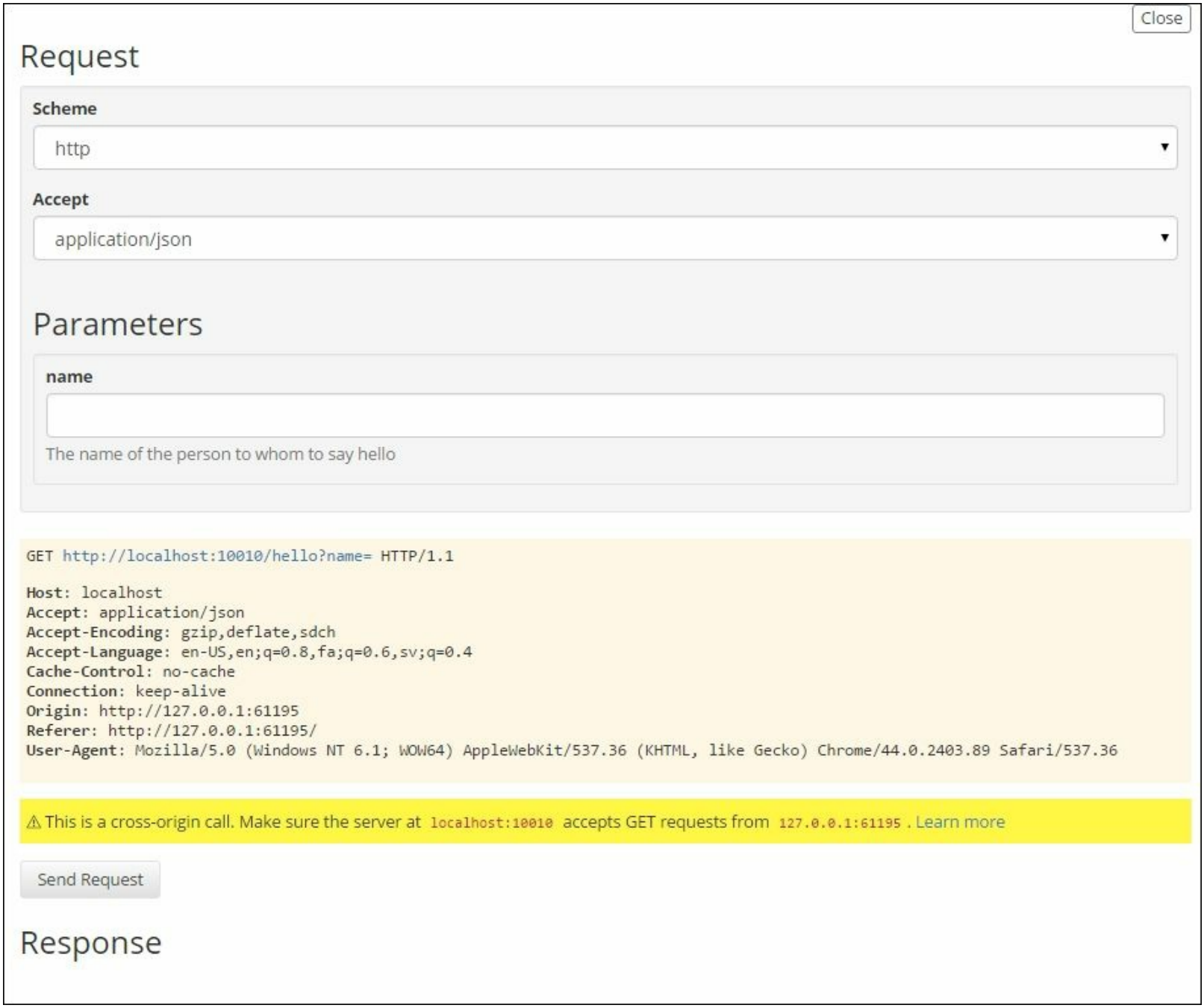
```
curl http://127.0.0.1:10010/hello?name=David
"Hello David!"
```

Swagger is binding the `name` query parameter to the Swagger parameter specified in the YAML definition. This may sound bad, as we are coupling our software to Swagger, but it gives you an immense benefit: Swagger allows you to test the endpoint through the editor. Let's see how it works.

On the right-hand side of the editor, you can see a button with the **Try this operation** label, as shown in the following screenshot:



Once you click it, it will present you a form that allows you to test the endpoint, as shown in the following screenshot:



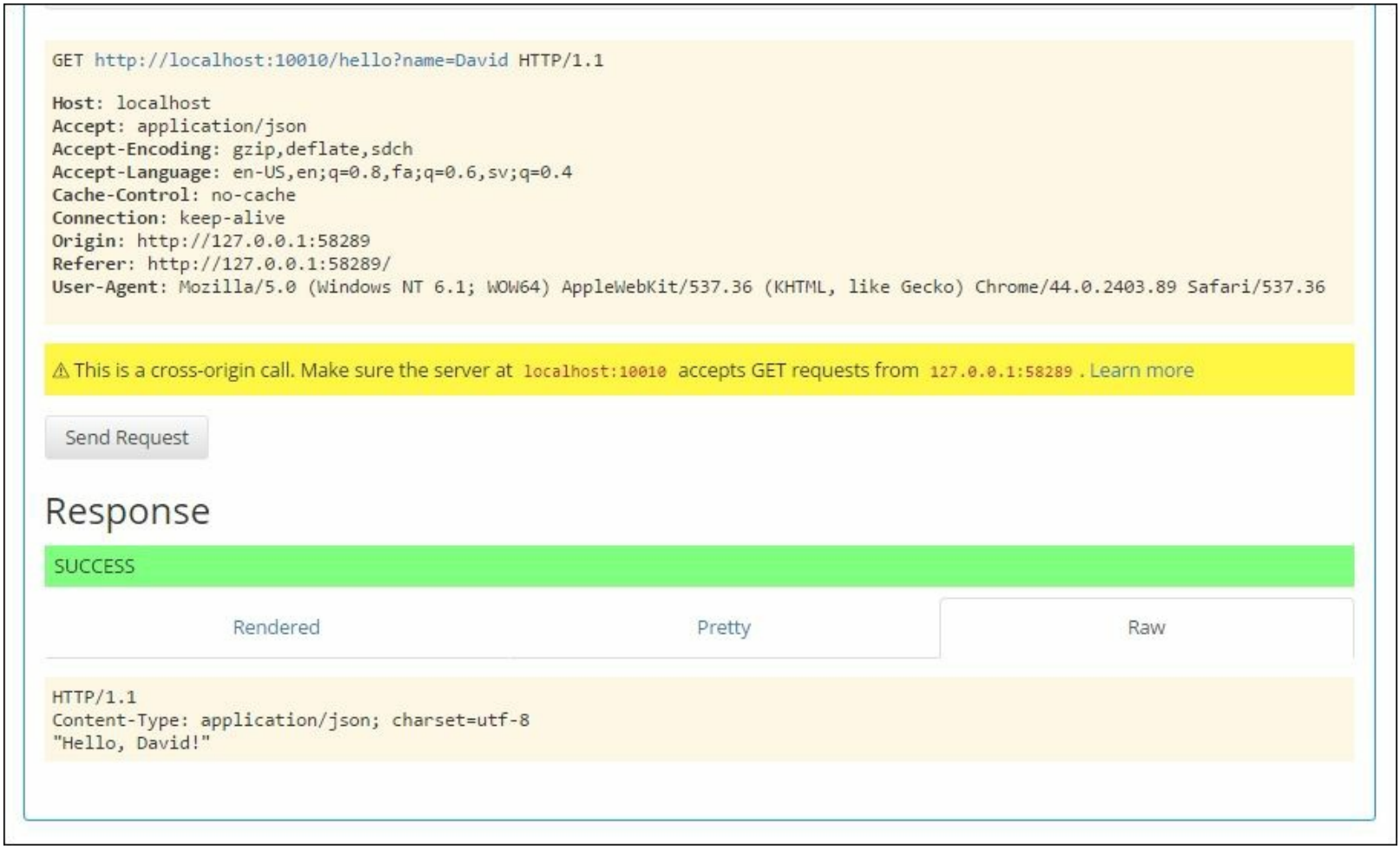
There is a warning message on this form about cross-origin requests. We don't need to worry about it when developing in our local machine; however, we could have problems when testing other hosts using the Swagger Editor.

Note

For more information, visit the following URL:

https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

Enter a value for the **name** parameter, and after that, click on **Send Request**, as shown in the following image:



This is a response example using Swagger Editor to test the endpoint

Be aware that, for this test to work, our app server has to be up and running.

Generating a project from the Swagger definition

Until now, we have been playing with Swagger and the generated project, but we are now going to generate the project from the `swagger.yaml` file. We will use the already generated project as a starting point, but we will add a new endpoint:

```
swagger: "2.0"
info:
  version: "0.0.1"
  title: Stop Words Filtering App
host: localhost:8000
basePath: /
schemes:
  - http
  - https
consumes:
  - application/json
produces:
  - application/json
paths:
  /stop-words:
    x-swagger-router-controller: stop_words
    get:
      description: Removes the stop words from an arbitrary input text.
      operationId: stopwords
      parameters:
        - name: text
          in: query
          description: The text to be sanitized
          required: false
          type: string
      responses:
        "200":
          description: Success
          schema:
            $ref: "#/definitions/StopWordsResponse"
  /swagger:
    x-swagger-pipe: swagger_raw
definitions:
  StopWordsResponse:
    required:
      - message
    properties:
      message:
        type: string
```

This endpoint might sound very familiar to you, as we unit tested it earlier in this chapter. As you probably know by now, the Swagger Editor is quite cool: it provides feedback as you type on, about what is going on in the YAML file, as well as saves the changes.

The next step is to download the Swagger code generator from https://github.com/swagger-

api/swagger-codegen. It is a Java project, so we are going to need the Java SDK and Maven to build it, as follows:

```
mvn package
```

Codegen is a tool that allow us to read the API definition from the Swagger YAML and build the basic structure for a project in a language of our choice, in this case, Node.js.

The preceding command in the root of the project should build all the submodules. Now, it is as easy as executing the following command in the root of the `swagger-codegen` folder:

```
java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar generate -i
my-project.yaml -l nodejs -o my-project
```

The Swagger code generator supports a number of languages. Here, the trick is that when using it for microservices, we can define the interface and then use the most appropriate technology to build our service.

If you go to the `my-project` folder, you should find the full structure of the project in there, ready to start coding.

Summary

In this chapter, you learned how to test and document microservices. It is usually the forgotten activity in software development, due to the pressures to deliver new functionalities, but in my opinion, it is a risky decision. We have to find the balance between too much and very little testing. In general, we will always try to find the right proportion for unit, integration and end-to-end tests.

You also learned about manual testing and the tools to efficiently test our software manually (there is always a component of manual testing).

Another interesting point is the documentation and API management. In this case, we got to know Swagger, which is probably the most popular API manager that led to the creation of the Open API standard.

If you want to go deeper in to the API world (there is a lot to learn in order to build a practical and efficient API), you should probably browse http://apigee.com. Apigee are a company expert on building APIs and providing tools for developers and enterprises that could help you to build a better API.

Chapter 7. Monitoring Microservices

Monitoring servers is always a controversial subject. It usually falls under system administration, and software engineers don't even go near it, but we are losing one of the huge benefits of monitoring: *the ability to react quickly to failures*. By monitoring our system very closely, we can be aware of problems almost immediately so that the actions to correct the problem may even save us from impacting the customers. Along with monitoring, there is the concept of performance. By knowing how our system behaves during load periods, we will be able to anticipate the necessity of scaling the system. In this chapter, we will discuss how to monitor servers, and specifically microservices, in order to maintain the stability of our system.

In this chapter, we will cover the following topics:

- Monitoring services
- Monitoring using PM2 and Keymetrics
- Monitoring metrics
- Simian Army – the active monitoring from Spotify
- Throughput and performance degradation

Monitoring services

When monitoring a microservice, we are interested in a few different types of metrics. The first big group of metrics is the hardware resources, which are described as follows:

- **Memory metrics**: This indicates how much memory is left in our system or consumed by our application
- **CPU utilization**: As the name suggests, this indicates how much CPU are we using at a given time
- **Disk utilization**: This indicates the I/O pressure in the physical hard drives

The second big group is the application metrics, as follows:

- Number of errors per time unit
- Number of calls per time unit
- Response time

Even though both groups are connected and a problem in the hardware will impact the application performance (and the other way around), knowing all of them is a must.

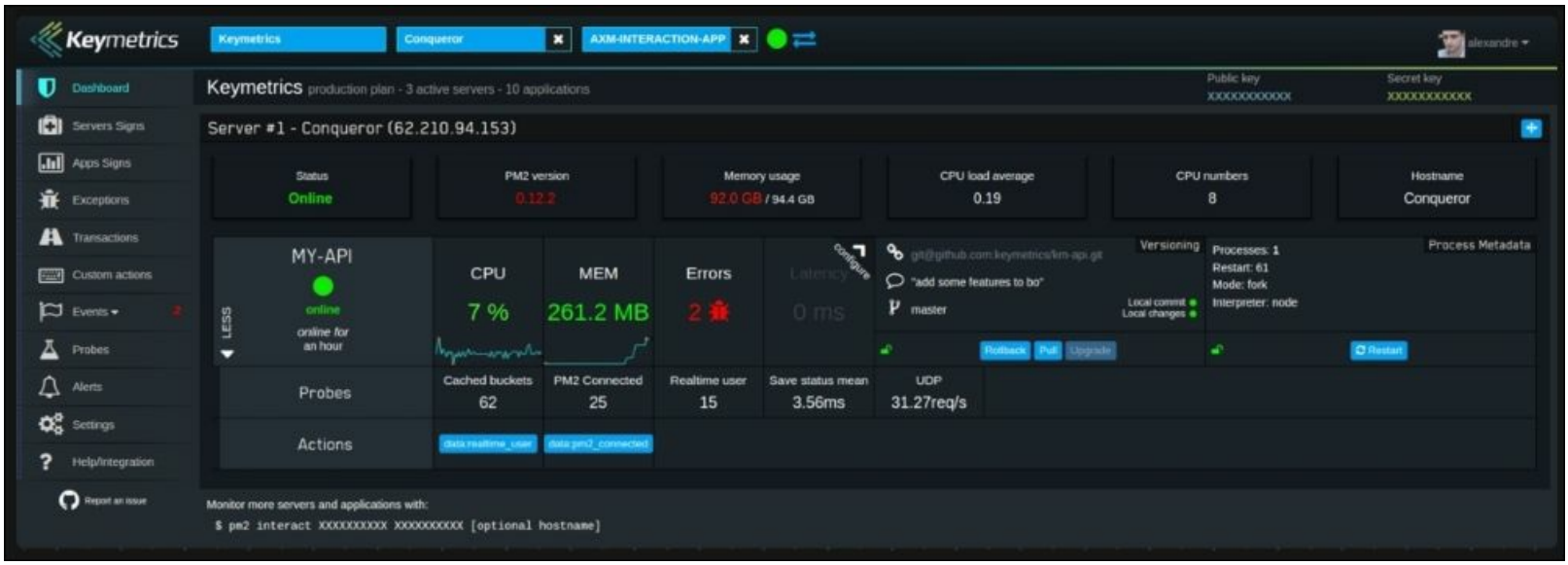
Hardware metrics are easy to query if our server is a Linux machine. On Linux, all the magic of hardware resources happens in the `/proc` folder. This folder is maintained by the kernel of the system and contains files about how the system behaves regarding a number of aspects in the system.

Software metrics are harder to collect and we are going to use **Keymetrics** from the creators of PM2 to monitor our Node.js applications.

Monitoring using PM2 and Keymetrics

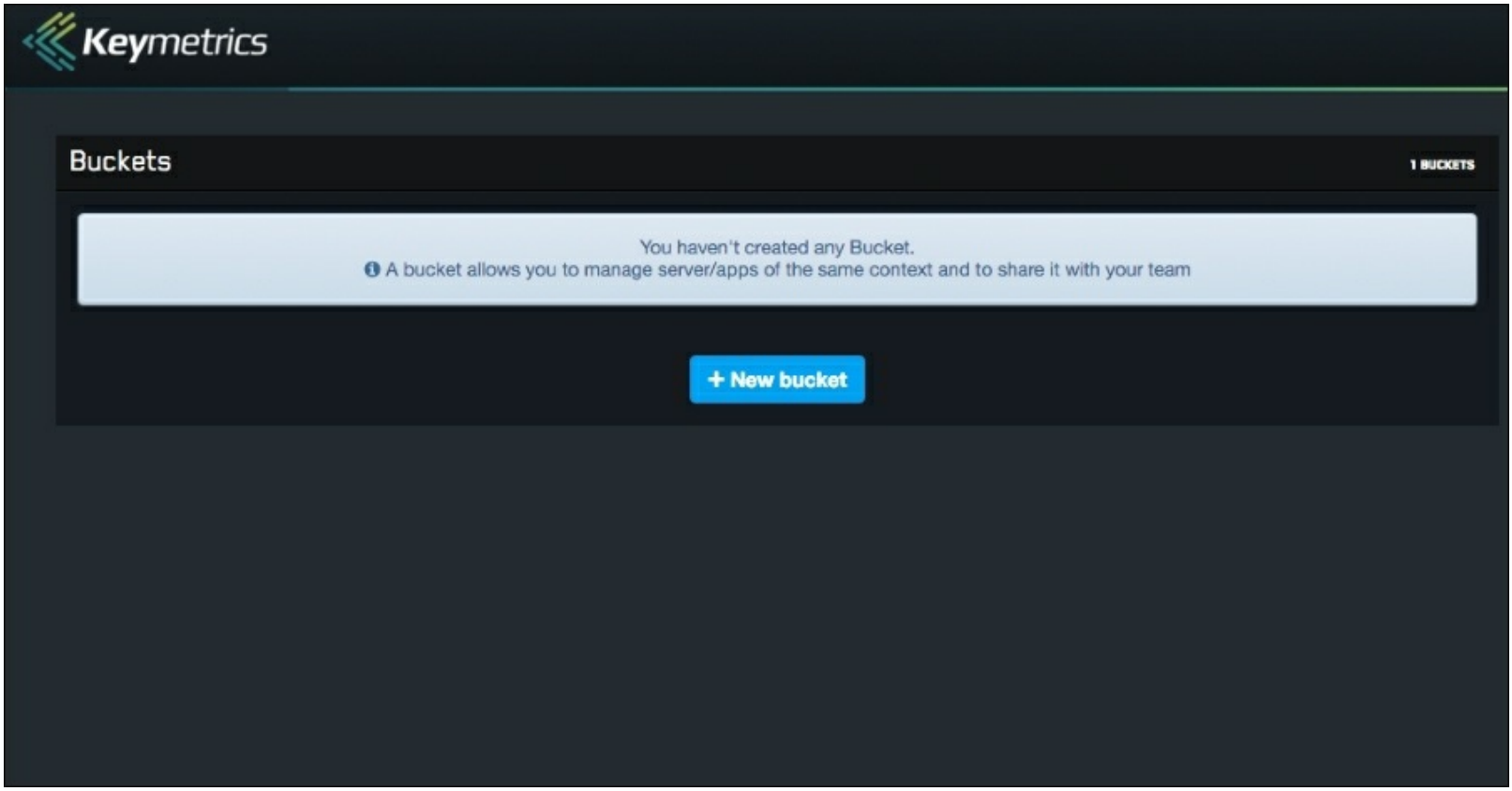
PM2, as we've seen before, is a very powerful instrument to run Node applications, but it is also very good at monitoring standalone applications in production servers. However, depending on your business case, it is not always easy to get access to the production.

The creators of PM2 have solved this problem by creating Keymetrics. Keymetrics is a **Software as a service** (**SaaS**) component that allows PM2 to send monitoring data across the network to its website, as shown in the following image (as found at https://keymetrics.io/):



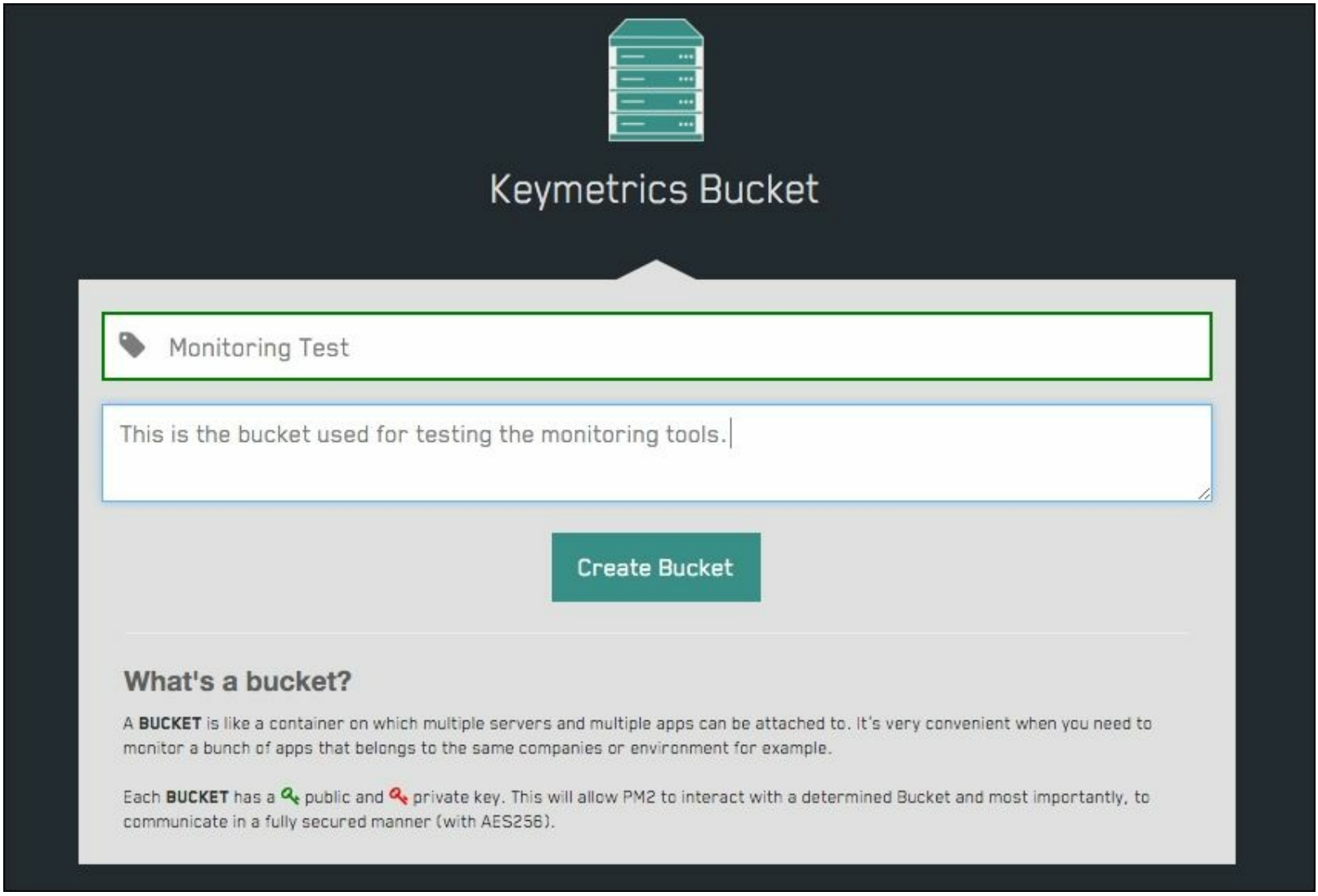
Even though Keymetrics is not free, it provides a free tier to demonstrate how it works. We are going to use it in this chapter.

The very first thing that we need to do is register a user. Once we get access to our account, we should see something similar to the following screen:

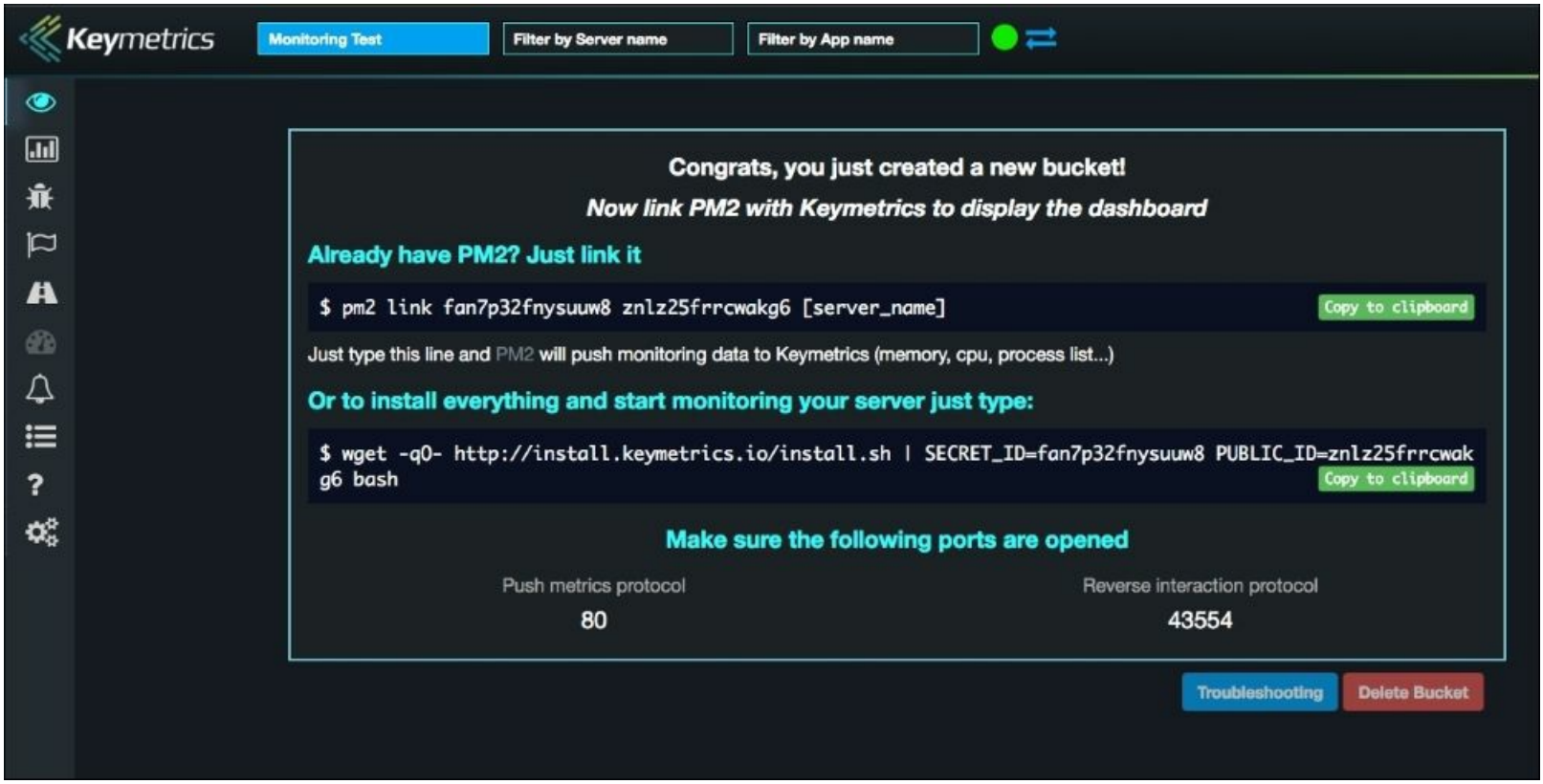


This screen is asking us to create a bucket. Keymetrics uses the bucket concept to define a context. For example, if our organization has different areas (payments, customer service, and so on) with different servers on each area, we could monitor all the servers in one bucket. There are no restrictions on how many servers you can have in one bucket. It is even possible to have all the organization in the same bucket so that everything is easy to access.

Let's create a bucket called `Monitoring Test`, as shown in the following image:



Easy, once we tap on **Create Bucket**, Keymetrics will show us a screen with the information needed to start monitoring our app, as shown in the following image:



As you can see, the screen displays information about the private key used by Keymetrics. Usually, it is a very bad idea to share this key with anyone.

As shown on the screen, the next step is to configure PM2 to push data into Keymetrics. There is also useful information about the networking needed to make Keymetrics work:

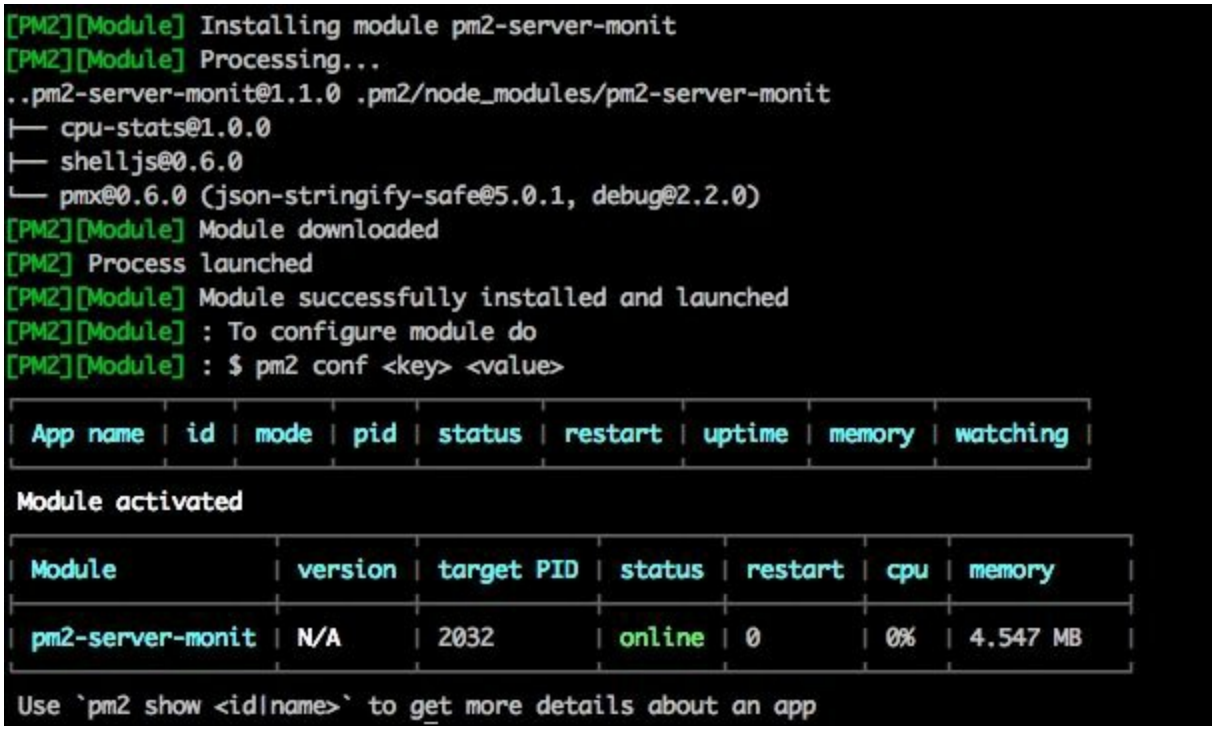
- PM2 will be pushing data to the port **80** on Keymetrics
- Keymetrics will be pushing data back to us on the port **43554**

Usually, in large organizations, there are restrictions about the networking, but if you are testing this from home, everything should work straightaway.

In order to make PM2 able to send metrics to Keymetrics, we need to install one PM2 module called `pm2-server-monit`. This is a fairly easy task:

```
pm2 install pm2-server-monit
```

This will result in an output similar to the following:



Let's run the advised command:

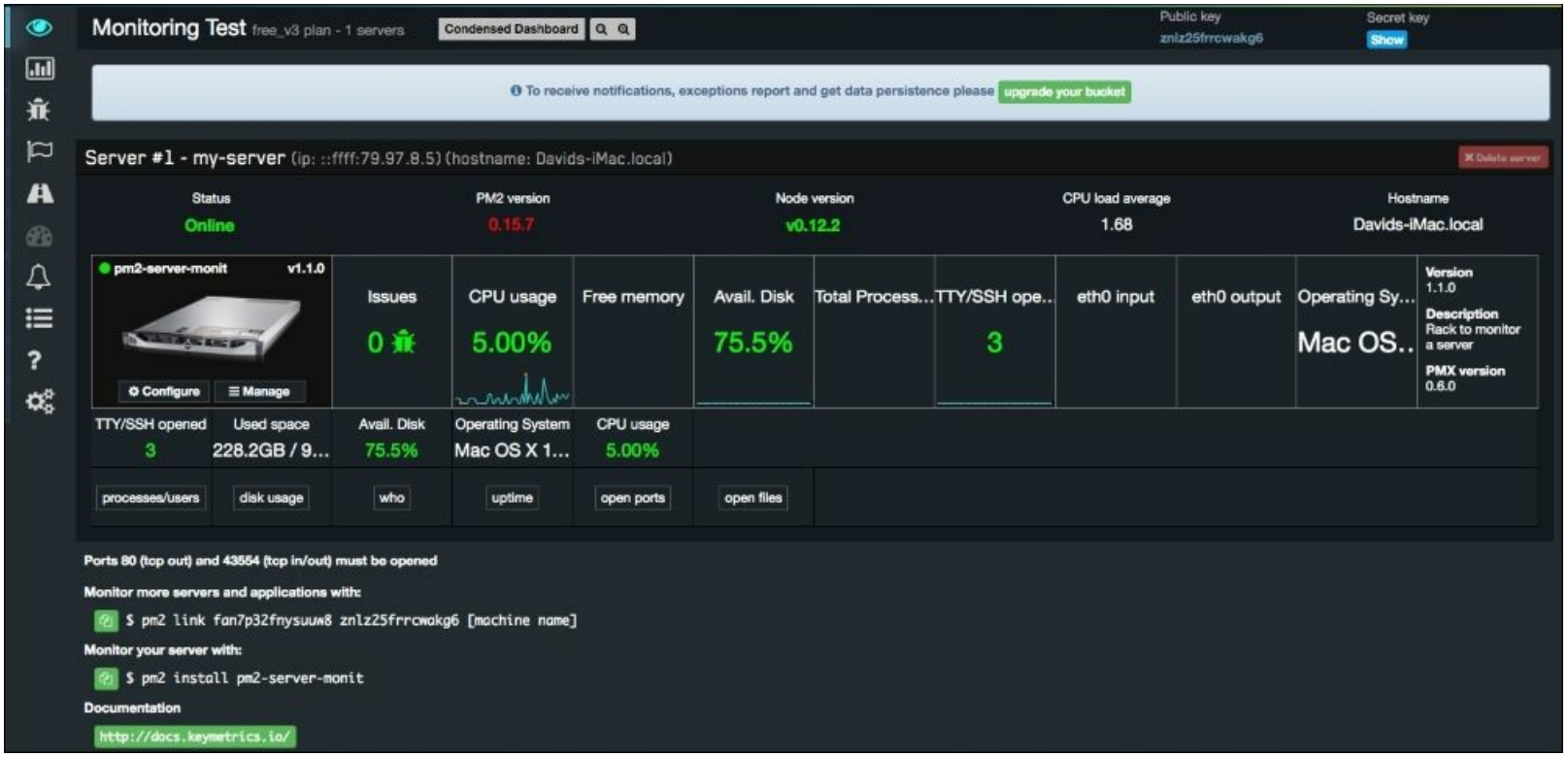
```
pm2 link fan7p32fnysuuw8 znlz25frrcwakg6 my-server
```

In this case, I have replaced `[server name]` with `my-server`. There are no restrictions on the server name; however, when rolling out Keymetrics into a real system, it is recommended to choose a descriptive name in order to easily identify the server in the dashboard.

The preceding command will produce an output similar to the following image:



This is an indication that everything went well and our application is ready to be monitored from Keymetrics that can be checked on https://app.keymetrics.io/, as follows:



Now, our server is showing up in the interface. As we previously stated, this bucket could monitor different servers. A simple virtual machine is created, and as you can see at the bottom of the screen, Keymetrics provides us with the command to be executed in order to add another server. In this case, as we are using the free access to Keymetrics, so we can only monitor one server.

Let's see what Keymetrics can offer us. At first sight, we can see interesting metrics such as CPU usage, memory available, disk available, and so on.

All these are hardware metrics that indicate how our system is behaving. Under pressure, they are the perfect indicator to point out the need for more hardware resources.

Usually, the hardware resources are the main indicator of failure in an application. Now, we are going to see how to use Keymetrics to diagnose the problem.

Diagnosing problems

A memory leak is usually a difficult problem to solve due to the nature of the flaw. Take a look at the following code.

Let's run the program using a simple `seneca.act()` action:

```
var seneca = require('seneca')();

var names = [];

seneca.add({cmd: 'memory-leak'}, function(args, done){
```

```
  names.push(args.name);
  greetings = "Hello " + args.name;
  done(null ,{result: greetings});
});

seneca.act({cmd: 'memory-leak', name: 'David'}, function(err, response) {
  console.log(response);
});
```

This program has a very obvious memory leak, and by obvious, I mean that it is written to be obvious. The `names` array will keep growing indefinitely. In the preceding example, it is not a big deal due to the fact that our application is a script that will start and finish without keeping the state in memory.

Tip

Remember that JavaScript allocates variables in the global scope if the `var` keyword is not used.

The problem comes when someone else reutilizes our code in a different part of the application.

Let's assume that our system grows to a point that we need a microservice to greet new customers (or deliver the initial payload of personal information such as name, preferences, configuration, and so on). The following code could be a good example on how to build it:

```
var seneca = require('seneca')();

var names = [];

seneca.add({cmd: 'memory-leak'}, function(args, done){
  names.push(args.name);
  greetings = "Hello " + args.name;
  done(null ,{result: greetings});
});

seneca.listen(null, {port: 8080});
```

In this example, Seneca will be listening over HTTP for requests from Seneca clients or other types of systems such as **curl**. When we run the application:

```
node index.js
2016-02-14T13:30:26.748Z szwj2mazorea/1455456626740/40489/- INFO hello
Seneca/1.1.0/szwj2mazorea/1455456626740/40489/-
2016-02-14T13:30:27.003Z szwj2mazorea/1455456626740/40489/- INFO listen
{port:8080}
```

Then from another terminal, we use curl to act as a client of our microservice, everything will work smoothly and our memory leak will go unnoticed:

```
curl -d '{"cmd": "memory-leak", "name":"David"}' http://127.0.0.1:8080/act
{"result":"Hello David"}%
```

However, we are going to use Keymetrics to find the problem. The first thing we need to do is run our

program using PM2. In order to do it so, we run the following command:

```
pm2 start index.js
```

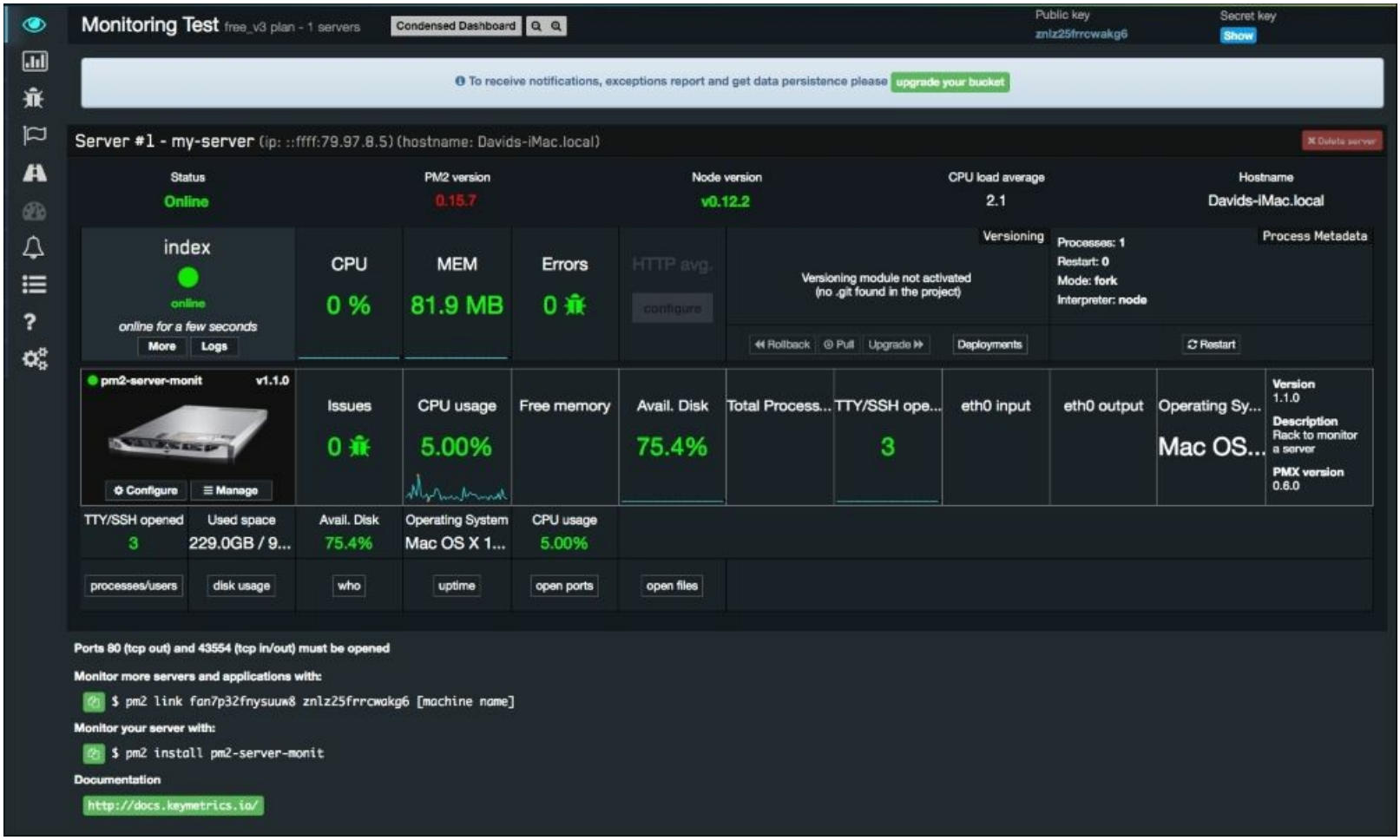
This command will produce the following output:



Let's explain the output in the following:

- The first line gives us information about the integration with Keymetrics. Data such as public key, server name, and the URL to access Keymetrics.
- In the first table, we can see the name of the application running, as well as few statistics on memory, uptime, CPU, and so on.
- In the second table, we can see the information relevant to the `pm2-server-monit` PM2 module.

Now, let's go to Keymetrics and see what has happened:



The application is now registered in Keymetrics as it can be seen in the control panel

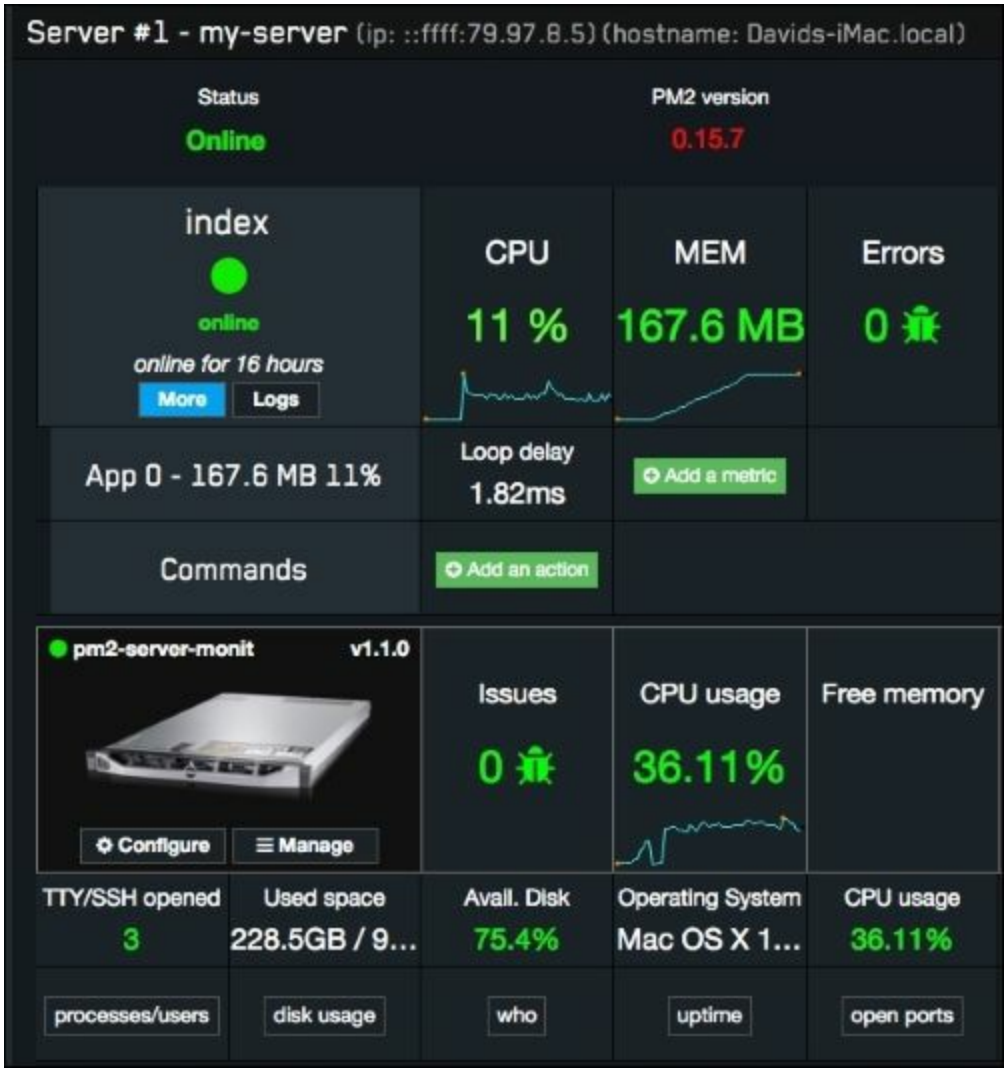
As you can see, now our application is showing up in Keymetrics.

Straightaway, we can see the very useful things about our app. One of these is the memory used. This is the metric that will indicate a memory leak, as it will keep growing.

Now, we are going to force the memory leak to cause a problem in our application. In this case, the only thing that we need to do is start our server (the small application that we wrote before) and issue a significant number of requests:

```
for i in {0..100000}
do
  curl -d '{"cmd": "memory-leak", "name":"David"}' http://127.0.0.1:8080/act
done
```

As simple as the small bash script, this is all it takes to open Pandora's Box in our application:



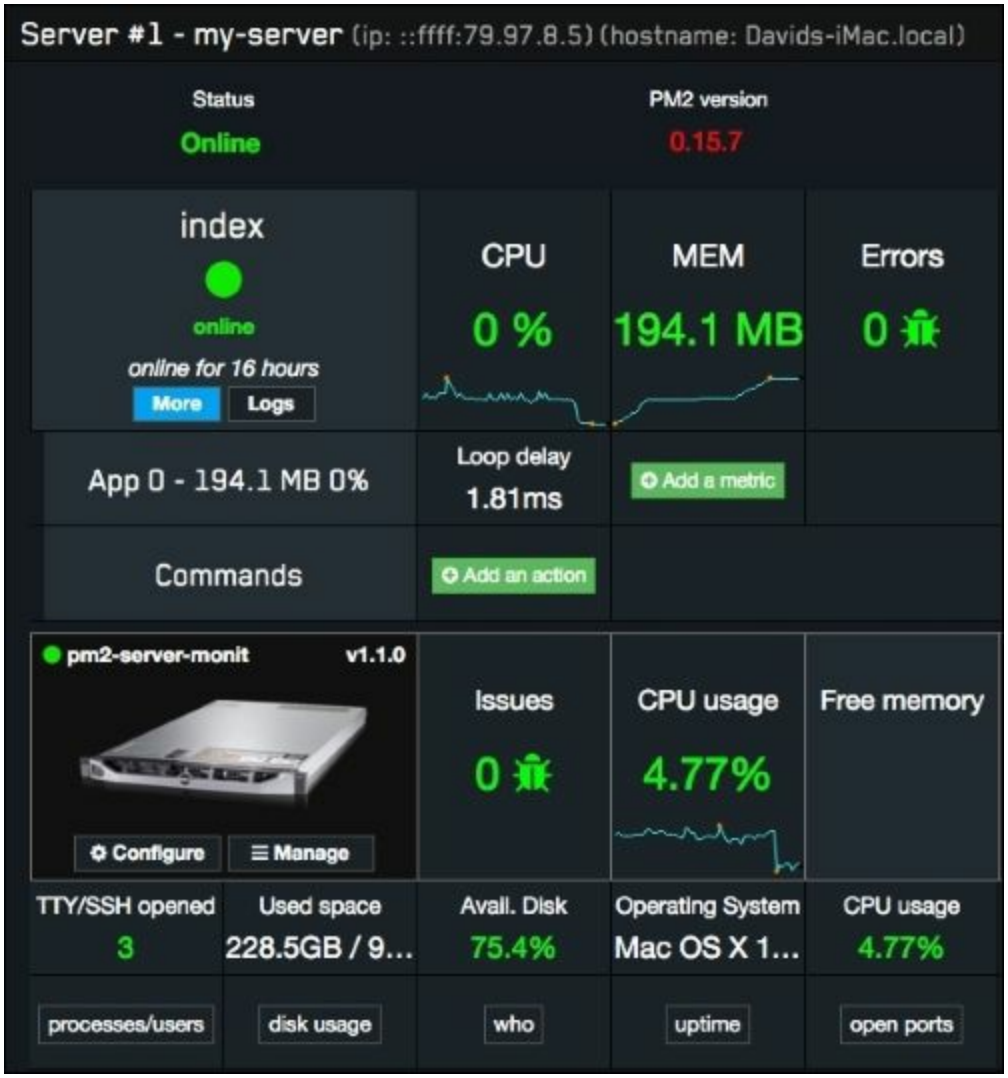
The application is now showing a high load (36% of CPU and an increased use of memory up to 167 MB)

The preceding image shows the impact of running the loop of requests in our system. Let's explain it in the following:

- The CPU in our application has gone to **11%** with an average loop delay of **1.82** milliseconds. Regarding our system, the overall CPU utilization has gone up to **36.11%** due to the fact that both the application and bash script use a significant amount of resources.
- The memory consumption has soared from **81.9 MB** to **167.6 MB**. As you can see, the line on the graph of memory allocation is not going straight up, and that is due to garbage collections. A garbage collection is an activity within the Node.js framework where unreferenced objects are freed from the memory, allowing our system to reuse the hardware resources.
- Regarding the errors, our application has been stable with **0** errors (we'll come back to this section later).

Now, once our bash script is finished (I stopped it manually, as it can take a significant amount of resources and time to finish), we can again see what happened to our system in the following

screenshot:



We can see that the CPU has gone back to normal, but what about the memory? The memory consumed by our application hasn't been freed due to the fact that our program has a memory leak, and as long as our variable is referencing the memory consumed (remember the `names` array is accumulating more and more names), it won't be freed.

In this case, we have a very simple example that clearly shows where the memory leak is, but in complex applications, it is not always obvious. This error, in particular, could never show up as a problem due to the fact that we probably deploy new versions of our app often enough to not realize it.

Monitoring application exceptions

Application errors are events that occur when our application can't handle an unexpected situation. Things such as dividing a number by zero or trying to access an undefined property of our application usually leads to these type of problems.

When working on a multithreaded framework (language) such as Tomcat, the fact that one of our

threads dies due to an exception usually only affects to one customer (the one holding the thread). However, in Node.js, a bubbled exception could be a significant problem as our application dies.

PM2 and Seneca do a very good job at keeping it alive as PM2 will restart our app if something makes it stop, and Seneca won't let the application die if an exception occurs in one of the actions.

Keymetrics has developed a module called **pmx** that we can use to programmatically get alerts on errors:

```
var seneca = require('seneca')();

var pmx = require('pmx');

var names = [];

seneca.add({cmd: 'exception'}, function(args, done){
  pmx.notify(new Error("Unexpected Exception!"));

  done(null ,{result: 100/args.number});
});

seneca.listen({port: 8085});
```

It is easy and self-descriptive: an action that sends an exception to Keymetrics if the number sent as a parameter is zero. If we run it, we will get the following output:

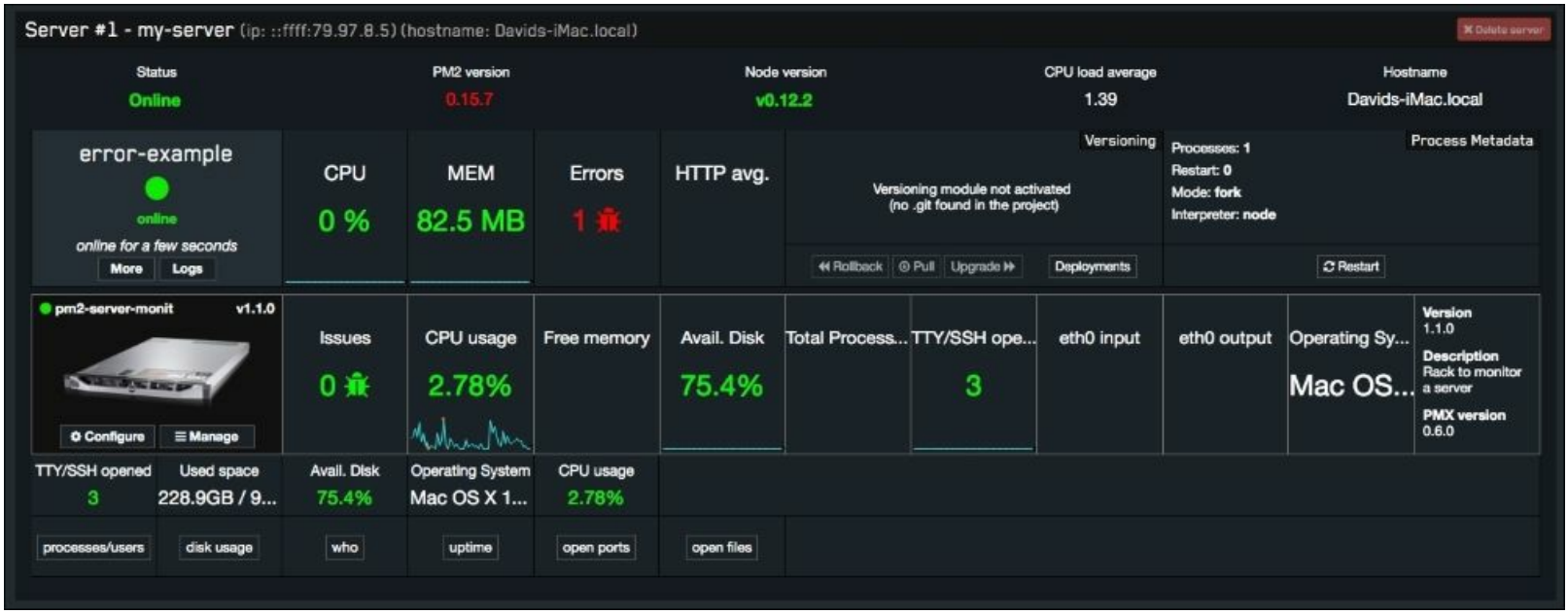
```
➜ code  pm2 start error-example.js
[PM2] Starting error-example.js in fork_mode (1 instance)
[PM2] Done.
• Agent online - public key: znlz25frrcwakg6 - machine name: my-server - Web access: https://app.keymetrics.io/
┌─────────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────────────┬──────────┐
│ App name    │ id │ mode │ pid   │ status │ restart │ uptime │ memory      │ watching │
├─────────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────────────┼──────────┤
│ error-example │ 7  │ fork │ 64724 │ online │ 0       │ 0s     │ 6.738 MB    │ disabled │
└─────────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────────────┴──────────┘
Module activated
┌──────────────┬─────────┬────────────┬────────┬─────────┬─────┬─────────────┐
│ Module       │ version │ target PID │ status │ restart │ cpu │ memory      │
├──────────────┼─────────┼────────────┼────────┼─────────┼─────┼─────────────┤
│ pm2-server-monit │ 1.1.0   │ 7139       │ online │ 0       │ 0%  │ 79.922 MB   │
└──────────────┴─────────┴────────────┴────────┴─────────┴─────┴─────────────┘
Use `pm2 show <id|name>` to get more details about an app
```

Now we need to hit the server in order to cause the error. As we did earlier, we will do this using curl:

```
curl -d '{"cmd": "exception", "number": "0"}' http://localhost:8085/act
{"result":null}%
```

Now, when we go to Keymetrics, we can see that there is an error logged, as shown in the following

image:



Another interesting point of Keymetrics is the configuration of alerts. As PM2 sends data about pretty much every metric in our system, we have the ability to configure Keymetrics on the thresholds that we consider healthy for our application.

This is very handy as we could get the notifications integrated in our corporate chat (something similar to **Slack**) and be alerted real time when something is not going correctly in our application.

Custom metrics

Keymetrics also allows us to use **probes**. A probe is a custom metric that is sent programmatically by the application to Keymetrics.

There are different types of values that the native library from Keymetrics allows us to push directly. We are going to see the most useful ones.

Simple metric

A simple metric is, as its name indicates, a very basic metric where the developer can set the value to the data sent to Keymetrics:

```
var seneca = require('seneca')();
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
});
var names = [];
```

```
var probe = pmx.probe();

var meter = probe.metric({
  name      : 'Last call'
});

seneca.add({cmd: 'last-call'}, function(args, done){
  console.log(meter);
  meter.set(new Date().toISOString());
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

In this case, the metric will send the time when the action was called for the last time to the Keymetrics:



The configuration for this metric is non-existent:

```
var probe = pmx.probe();

var meter = probe.metric({
  name      : 'Last call'
});
```

There is no complexity in this metric.

Counter

This metric is very useful to count how many times an event occurred:

```
var seneca = require('seneca')();
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
});
var names = [];
```

```
var probe = pmx.probe();

var counter = probe.counter({
  name : 'Number of calls'
});

seneca.add({cmd: 'counter'}, function(args, done){
  counter.inc();
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

In the preceding code, we can see how the counter is incremented for every single call to the action counter.

This metric will also allow us to decrement the value calling the `dec()` method on the counter:

```
counter.dec();
```

Average calculated values

This metric allows us to record when an event occurs, and it will automatically calculate the number of events per time unit. It is quite useful to calculate averages and is a good tool to measure the load in the system. Let's see a simple example, as follows:

```
var seneca = require('seneca')();
var pmx = require("pmx").init({
  http: true,
  errors: true,
  custom_probes: true,
  network: true,
  ports: true
});
var names = [];
var probe = pmx.probe();

var meter = probe.meter({
  name      : 'Calls per minute',
  samples   : 60,
  timeframe : 3600
});

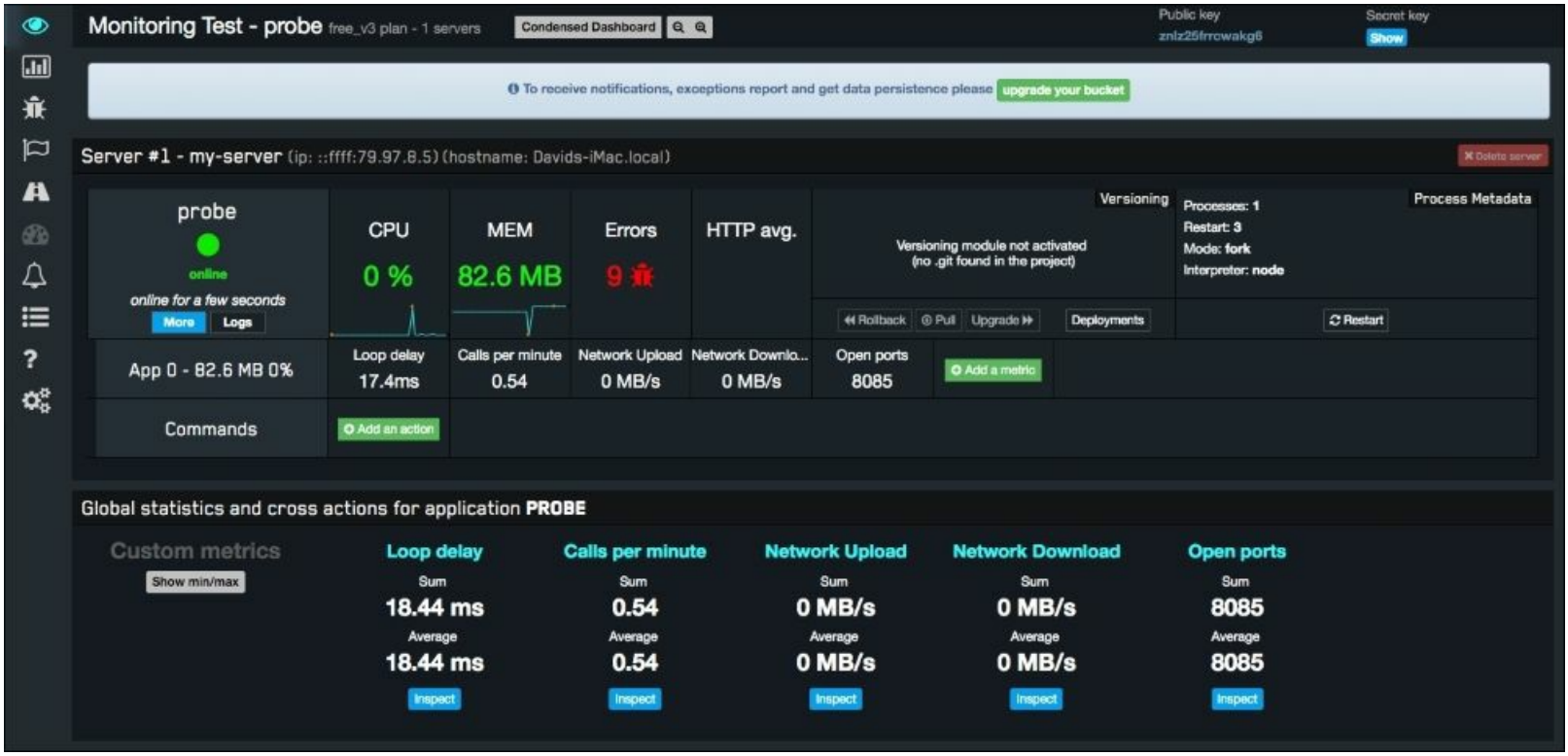
seneca.add({cmd: 'calls-minute'}, function(args, done){
  meter.mark();
  done(null, {result: "done!"});
});

seneca.listen({port: 8085});
```

The preceding code creates a probe and sends a new metric called `Calls per minute` to Keymetrics.

Now, if we run the application and the following command a few times, the metric is shown in the following Keymetrics interface:

```
curl -d '{"cmd": "calls-minute"}' http://localhost:8085/act
```



As you can see, there is a new metric called `Calls per minute` in the UI. The key to configure this metric is in the following initialization:

```
var meter = probe.meter({
  name      : 'Calls per minute',
  samples   : 60,
  timeframe : 3600
});
```

As you can see, the name of the metric is in the configuration dictionary as well as in two parameters: `samples` and `timeframe`.

The `samples` parameter correspond to the interval where we want to rate the metric; in this case, it is the number of calls per minute so that rate is `60` seconds.

The `timeframe` parameter, on the other hand, is for how long we want Keymetrics to hold the data for, or to express in simpler words, it is the timeframe over which the metric will be analyzed.

Simian Army – the active monitoring from Spotify

Spotify is one of the companies of reference when building microservices-oriented applications. They are extremely creative and talented when it boils down to coming up with new ideas.

One of my favourite ideas among them is what they call the **Simian Army**. I like to call it **active monitoring**.

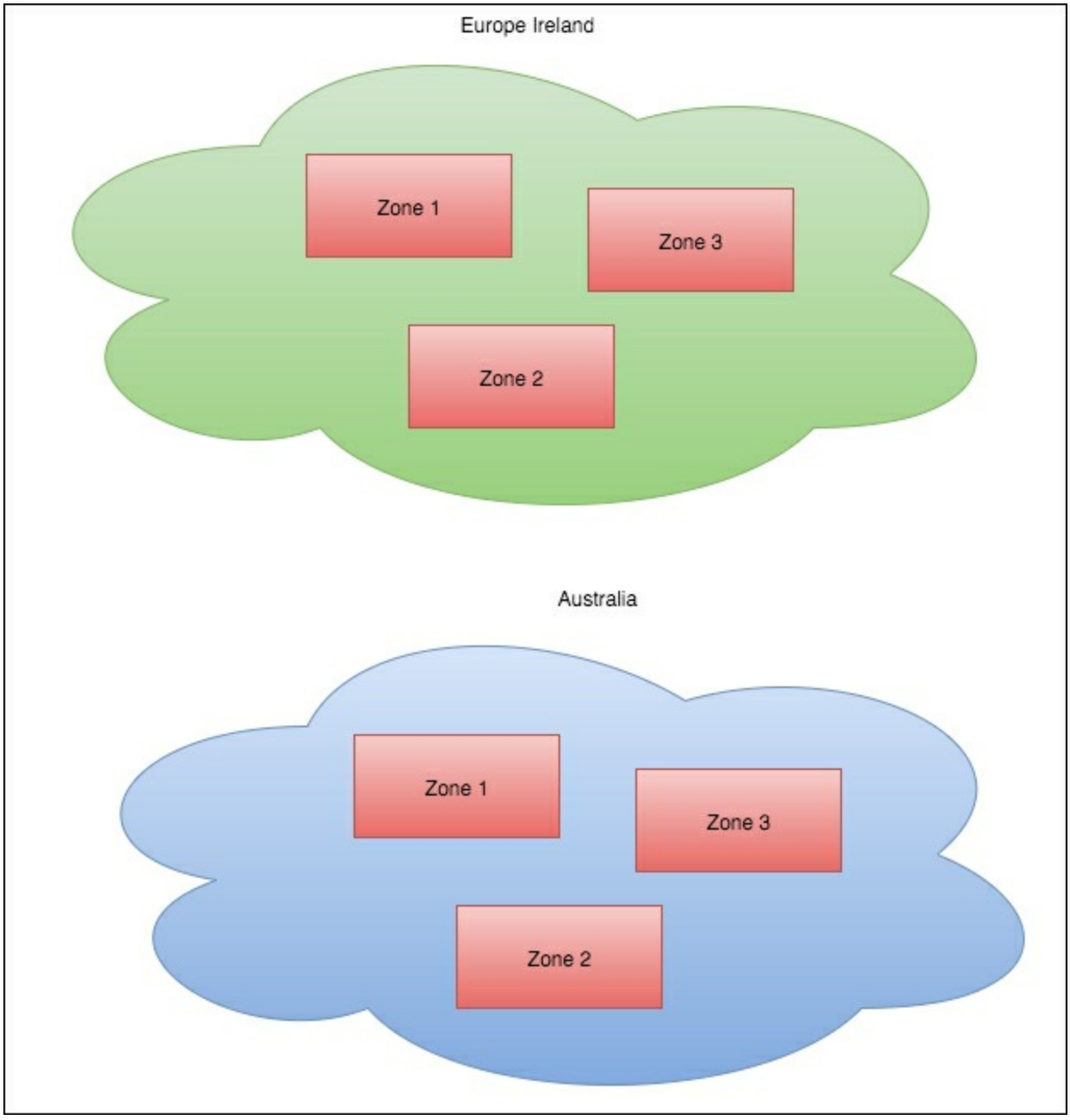
In this book, I have talked a lot times about how humans fail at performing different tasks. No matter how much effort you put in to creating your software, there are going to be bugs that will compromise the stability of the system.

This is a big problem, but it becomes a huge deal when, with the modern cloud providers, your infrastructure is automated with a script.

Think about it, what happens if in a pool of thousand servers, three of them have the *time zone out of sync* with the rest of the servers? Well, depending on the nature of your system, it could be fine or it could be a big deal. Can you imagine your bank giving you a statement with disordered transactions?

Spotify has solved (or mitigated) the preceding problem by creating a number of software agents (a program that moves within the different machines of our system), naming them after different species of monkeys with different purposes to ensure the robustness of their infrastructure. Let's explain it a bit further.

As you are probably aware, if you have worked before with Amazon Web Services, the machines and computational elements are divided in to regions (EU, Australia, USA, and so on) and inside every region, there are availability zones, as shown in the following diagram:



This enables us, the engineers, to create software and infrastructure without hitting what we call a single point of failure.

Note

A **single point of failure** is a condition in a system where the failure of a single element will cause the system to misbehave.

This configuration raised a number of questions to the engineers in Spotify, as follows:

- What happens if we blindly trust our design without testing whether we actually have any point of failures or not?
- What happens if a full availability zone or region goes down?
- How is our application going to behave if there is an abnormal latency for some reason?

To answer all these questions, Netflix has created various agents. An agent is a software that runs on a system (in this case, our microservices system) and carries on different operations such as checking the hardware, measuring the network latency, and so on. The idea of agent is not new, but until now, its application was nearly a futuristic subject. Let's take a look at the following agents created by Netflix:

- **Chaos Monkey**: This agent disconnects healthy machines from the network in a given availability zone. This ensures that there are *no single points of failures within an availability zone*. So that if our application is balanced across four nodes, when the Chaos Monkey kicks in, it will disconnect one of these four machines.
- **Chaos Gorilla**: This is similar to Chaos Monkey, Chaos Gorilla will disconnect a full availability zone in order to verify that Netflix services rebalance to the other available zones. In other words, Chaos Gorilla is the big brother of Chaos Monkey; instead of operating at the server level, it operates at the partition level.
- **Latency Monkey**: This agent is responsible for introducing artificial latencies in the connections. Latency is usually something that is hardly considered when developing a system, but it is a very delicate subject when building a microservices architecture: latency in one node could compromise the quality of the full system. When a service is running out of resources, usually the first indication is the latency in the responses; therefore, Latency Monkey *is a good way to find out how our system will behave under pressure*.
- **Doctor Monkey**: A health check is an endpoint in our application that returns an HTTP 200 if everything is correct and 500 error code if there is a problem within the application. Doctor Monkey is an agent that will randomly execute the health check of nodes in our application and report the faulty ones in order to replace them.
- **10-18 Monkey**: Organizations such as Netflix are global, so they need to be language-aware (certainly, you don't want to get a website in German when your mother tongue is Spanish). The 10-18 Monkey reports on instances that are misconfigured.

There are a few other agents, but I just want to explain active monitoring to you. Of course, this type of monitoring is out of reach of small organizations, but it is good to know about their existence so that we can get inspired to set up our monitoring procedures.

Note

The code is available under Apache License in the following repository:

https://github.com/Netflix/SimianArmy.

In general, this active monitoring follows the philosophy of *fail early*, of which, I am a big adept. No matter how big the flaw in your system is or how critical it is, you want to find it sooner than later, and ideally, without impacting any customer.

Throughput and performance degradation

Throughput is to our application what the monthly production is to a factory. It is a unit of measurement that gives us an indication about how our application is performing and answers the *how many* question of a system.

Very close to throughput, there is another unit that we can measure: **latency**.

Latency is the performance unit that answers the question of *how long*.

Let's consider the following example:

Our application is a microservices-based architecture that is responsible for calculating credit ratings of applicants to withdraw a mortgage. As we have a large volume of customers (a nice problem to have), we have decided to process the applications in batches. Let's draw a small algorithm around it:

```
var seneca = require('seneca')();
var senecaPendingApplications = require('seneca').client({type: 'tcp',
  port: 8002,
  host: "192.168.1.2"});
  var senecaCreditRatingCalculator = require('seneca').client({type: 'tcp',
  port: 8002,
  host: "192.168.1.3"});

seneca.add({cmd: 'mortgages', action: 'calculate'}, function(args, callback) {
  senecaPendingApplications.act({
    cmd: 'applications',
    section: 'mortgages',
    custmers: args.customers}, function(err, responseApplications) {
      senecaCreditRatingCalculator.act({cmd: 'rating', customers:
args.customers}, function(err, response) {

        processApplications(response.ratings,
        responseApplications.applications,
        args.customers
      );
    });
  });
});
```

This is a small and simple Seneca application (this is only theoretical, don't try to run it as there is a lot of code missing!) that acts as a client for two other microservices, as follows:

- The first one gets the list of pending applications for mortgages
- The second one gets the list of credit rating for the customers that we have requested

This could be a real situation for processing mortgage applications. In all fairness, I worked on a very similar system in the past, and even though it was a lot more complex, the workflow was very

similar.

Let's talk about throughput and latency. Imagine that we have a fairly big batch of mortgages to process and the system is misbehaving: the network is not being as fast as it should and is experiencing some dropouts.

Some of these applications will be lost and will need to be retried. In ideal conditions, our system is producing a throughput of 500 applications per hour and takes an average of 7.2 seconds on latency to process every single application. However, today, as we stated before, the system is not at its best; we are processing only 270 applications per hour and takes on average 13.3 seconds to process a single mortgage application.

As you can see, with latency and throughput, we can measure how our business transactions are behaving with respect to the previous experiences; we are operating at 54% of our normal capacity.

This could be a serious issue. In all fairness, a drop off like this should ring all the alarms in our systems as something really serious is going on in our infrastructure; however, if we have been smart enough while building our system, the performance will be degraded, but our system won't stop working. This can be easily achieved by the usage of circuit breakers and queueing technologies such as **RabbitMQ**.

Queueing is one of the best examples to show how to apply human behavior to an IT system. Seriously, the fact that we can easily decouple our software components having a simple message as a joint point, which our services either produce or consume, gives us a big advantage when writing complex software.

Other advantage of queuing over HTTP is that an HTTP message is lost if there is a network drop out.

We need to build our application around the fact that it is either full success or error. With queueing technologies such as RabbitMQ, our messaging delivery is asynchronous so that we don't need to worry about intermittent failures: *as soon as we can deliver the message to the appropriate queue, it will get persisted until the client is able to consume it (or the message timeout occurs)*.

This enables us to account for intermittent errors in the infrastructure and build even more robust applications based on the communication around queues.

Again, Seneca makes our life very easy: the plugin system on which the Seneca framework is built makes writing a transport plugin a fairly simple task.

Note

RabbitMQ transport plugin can be found in the following GitHub repository:

https://github.com/senecajs/seneca-rabbitmq-transport

There are quite few transport plugins and we can also create our own ones (or modify the existing ones!) to satisfy our needs.

If you take a quick look at the RabbitMQ plugin (just as an example), the only thing that we need to do to write a transport plugin is overriding the following two Seneca actions:

- `seneca.add({role: 'transport', hook: 'listen', type: 'rabbitmq'}, ...)`
- `seneca.add({role: 'transport', hook: 'client', type: 'rabbitmq'}, ...)`

Using queueing technologies, our system will be *more resilient against intermittent failures* and we would be able to degrade the performance instead of heading into a catastrophic failure due to missing messages.

Summary

In this chapter, we deep dived into PM2 monitoring through Keymetrics. We learned how to put tight monitoring in place so that we are quickly informed about the failures in our application.

In the software development life cycle, the **QA** phase is, in my opinion, one of the most important one: no matter how good your code looks, if it does not work, it is useless. However, if I have to choose another phase where engineers should put more emphasis, it would be the deployment, and more specifically, the monitoring that is carried out after every deployment. If you receive error reports immediately, chances are that the reaction can be quick enough to avoid bigger problems such as corrupted data or customers complaining.

We also saw an example of active monitoring carried out by Netflix on their systems, which even though might be out of the reach of your company, it can spark good ideas and practices in order to monitor your software.

Keymetrics is just an example that fits the bill for Node.js as it is extremely well integrated with PM2, but there are also other good monitoring systems such as **AppDynamics**, or if you want to go for an in-house software, you could use Nagios. The key is being clear about what you need to monitor in the application, and then, find the best provider.

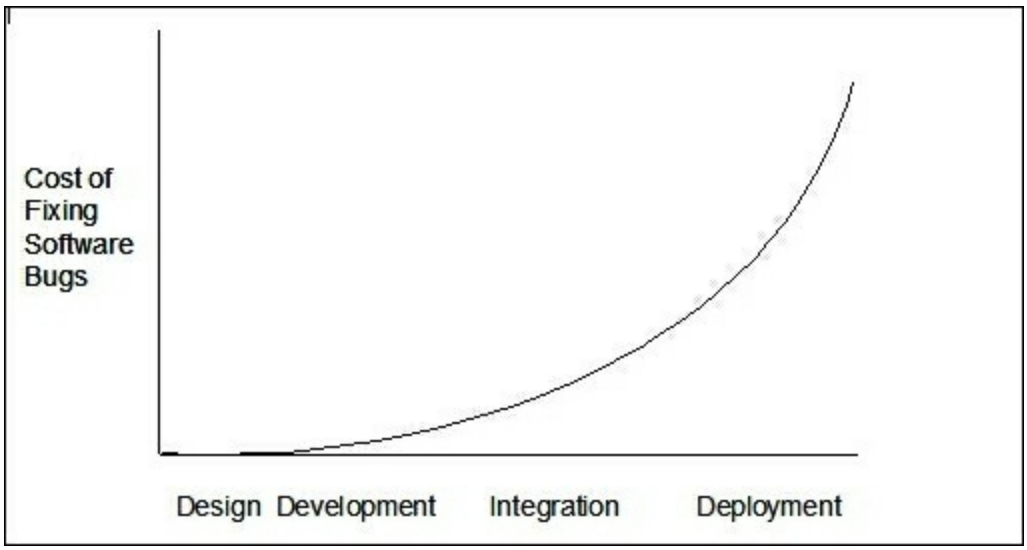
Another two good options for monitoring Node.js apps are StrongLoop and New Relic. They are both on the same line with Keymetrics, but they work better for large-scale systems, especially StrongLoop, which is oriented to applications written in Node.js and oriented to microservices.

Chapter 8. Deploying Microservices

In this chapter, we are going to deploy microservices. We will use different technologies in order to provide the reader with the knowledge required to choose the right tool for every job. First, we will use PM2 and its deployment capabilities to run applications in remote servers. Then, we will play around Docker, which is one of the most advanced deployment platforms, and the entire ecosystem around containers. In this chapter, we will show how to automate all the deployments as highly as possible.

Concepts in software deployment

Deployments are usually the ugly friend of the **Software Development Life Cycle** (**SDLC**) party. There is a missing contact point between development and systems administration that DevOps is going to solve in the next few years. The cost of fixing bugs at different stages of SDLC is shown in the following figure:



This diagram shows the cost of fixing a bug, depending on the stage of the SDLC

Fail early is one of my favorite concepts in the lean methodology. In the change management world, the cost of fixing a bug in the different stages of the software life cycle is called the **cost of change curve**.

Roughly, fixing a bug in production is estimated to cost 150 times the resources as compared to the costs to fix it when taking requirements.

No matter what the figure is, which depends a lot on the methodology and technology that we use, the lesson learned is that we can save a lot of time by catching bugs early.

From the continuous integration up to continuous delivery, the process should be automated as much as possible, where *as much as possible* means 100%. Remember, humans are imperfect and more prone to errors while carrying out manual repetitive tasks.

Continuous integration

Continuous integration (**CI**) is the practice of integrating the work from different branches on daily basis (or more than once a day) and validating that the changes do not break existing features by running integration and unit tests.

CI should be automated using the same infrastructure configuration as we will be using later in pre-production and production, so if there is any flaw, it can be caught early.

Continuous delivery

Continuous delivery (**CD**) is a software engineering approach that aims to build small, testable, and easily deployable pieces of functionality that can be delivered seamlessly at any time.

This is what we are aiming for with the microservices. Again, we should be pushing to automate the delivery process as, if we are doing it manually, we are only looking for problems.

When talking from the microservices' perspective, automation on deployments is the key. We need to tackle the overhead of having a few dozen of services instead of a few machines, or we can find ourselves maintaining a cloud of services instead of adding value to our company.

Docker is our best ally here. With Docker, we are reducing the hassle of deploying a new software to pretty much moving a file (a container) around in different environments, as we will see later in this chapter.

Deployments with PM2

PM2 is an extremely powerful tool. No matter what stage of development we are in, PM2 always has something to offer.

In this phase of software development, the deployment is where PM2 really shines. Through a JSON configuration file, PM2 will manage a cluster of applications so that we can easily deploy, redeploy, and manage applications on remote servers.

PM2 – ecosystems

PM2 calls a group of applications ecosystem. Every ecosystem is described by a JSON file, and the easiest way to generate it is executing the following command:

```
pm2 ecosystem
```

This should output something similar to the following code:

```
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
File /path/to/your/app/ecosystem.json generated
```

The content of the `ecosystem.json` file varies, depending on the version of PM2, but what this file contains is the skeleton of a PM2 cluster:

```
{
  apps : [

    {
      name      : "My Application",
      script    : "app.js"
    },

    {
      name      : "Test Web Server",
      script    : "proxy-server.js"
    }
  ],

*/
  deploy : {
    production : {
      user : "admin",
      host : "10.0.0.1",
      ref  : "remotes/origin/master",
      repo : "git@github.com:the-repository.git",
      path : "/apps/repository",
      "post-deploy" : "pm2 startOrRestart ecosystem.json --env production"
    },
    dev : {
      user : "devadmin",
      host : "10.0.0.1",
      ref  : "remotes/origin/master",
      repo : "git@github.com:the-repository.git",
      path : "/home/david/development/test-app/",
      "post-deploy" : "pm2 startOrRestart ecosystem.json --env dev",
    }
  }
}
```

This file contains two applications configured for two environments. We are going to modify this

skeleton to adapt it to our needs, modeling our entire ecosystem written in Chapter 4, *Writing Your First Microservice in Node.js*.

However, for now, let's explain a bit for the configuration:

- We have an array of applications (`apps`) defining two apps: API and WEB
- As you can see, we have a few configuration parameters for each app:
  - `name`: This is the name of the application
  - `script`: This is the startup script of the app
  - `env`: These are the environment variables to be injected into the system by PM2
  - `env_<environment>`: This is same as `env`, but it is tailored per environment
- There are two environments defined in the default ecosystem under the `deploy` key, as follows:
  - `production`
  - `dev`

As you can see, between these two environments, there are no significant changes except for the fact that we are configuring one environment variable in development and the folder where we deploy our application.

Deploying microservices with PM2

In Chapter 4, *Writing Your First Microservice in Node.js*, we wrote a simple e-commerce in order to show the different concepts and common catches in microservices.

Now, we are going to learn how to deploy them using PM2.

Configuring the server

First thing we need to do in order to deploy software with PM2 is to configure the remote machine and local machine to be able to talk using SSH, with a public/private key schema.

The way of doing it is easy, as follows:

- Generate one RSA key
- Install it into the remote server

Let's do it:

```
ssh-keygen -t rsa
```

That should produce something similar to the following output:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/youruser/.ssh/id_rsa):
/Users/youruser/.ssh/pm2_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in pm2_rsa.
Your public key has been saved in pm2_rsa.pub.
The key fingerprint is:
eb:bc:24:fe:23:b2:6e:2d:58:e4:5f:ab:7b:b7:ee:38 dgonzalez@yourmachine.local
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|                 |
|    .            |
|   o    S        |
|    o   ..       |
|   o o..o.       |
|  . +.+=E..      |
|   oo+**B+.      |
+-----------------+
```

Now, if we go to the folder indicated in the preceding output, we can find the following two files:

- `pm2_rsa`: The first one, `pm2_rsa`, is your private key. As you can read from the name, no one should have access to this key as they may steal your identity in the servers that trust this key.
- `pm2_rsa.pub`: The `pm2_rsa.pub` is your public key. This key can be handed over to anyone so

that using asymmetric cryptography techniques, they can verify your identity (or who you say you are).

What we are going to do now is copy the public key to the remote server so that when our local machine PM2 tries to talk to the server, it knows who we are and let's get into the shell without password:

```
cat pm2_rsa.pub | ssh youruser@yourremoteserver 'cat >> .ssh/authorized_keys'
```

The last step is to register your private key as a known identity in your local machine:

```
ssh-add pm2_rsa
```

That's about it.

From now on, whenever you SSH into the remote server using as a user `youruser`, you won't need to enter the password in order to get into the shell.

Once this configuration is done, there is very little to do in order to deploy any application into this server:

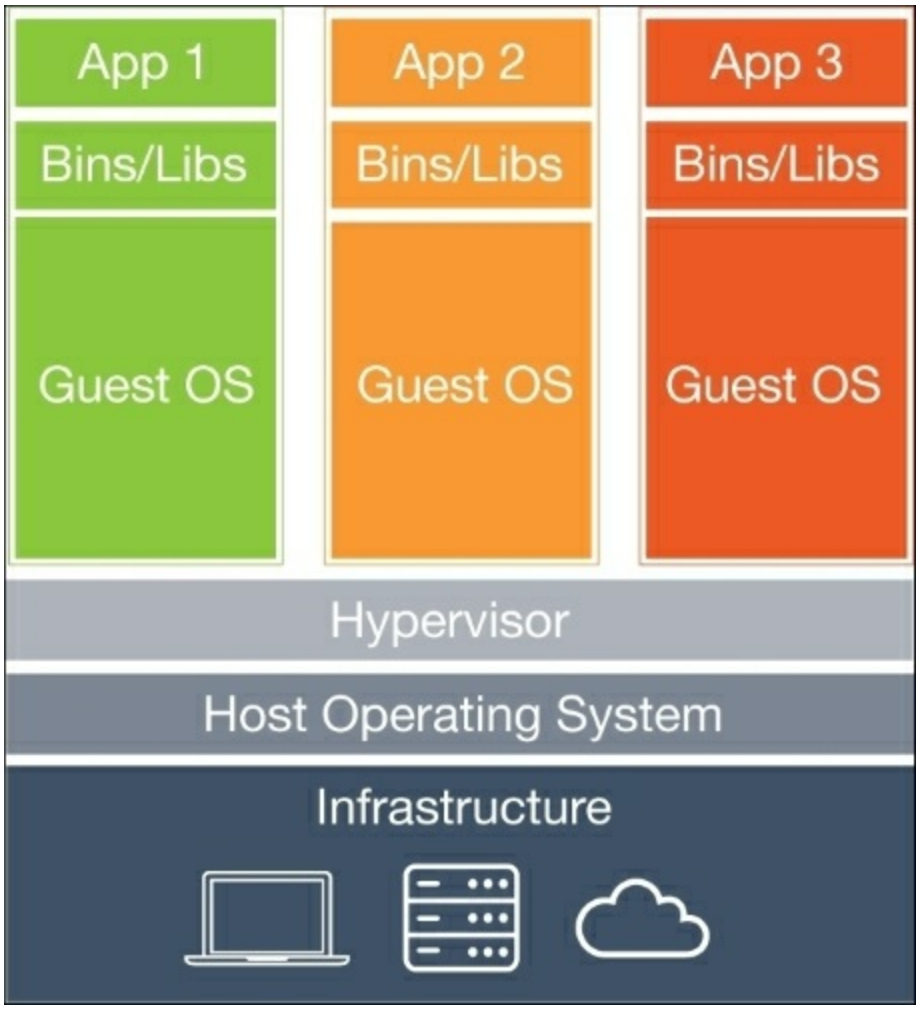
```
pm2 deploy ecosystem.json production setup
pm2 deploy ecosystem.json production
```

The first command will configure everything needed to accommodate the app. The second command will actually deploy the application itself as we configured earlier.

Docker – a container for software delivery

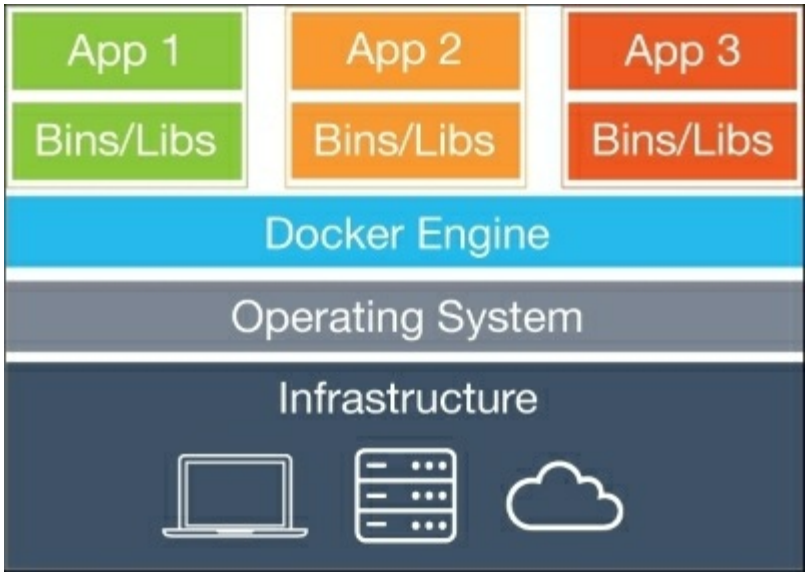
Virtualization has been one of the biggest trends in the past few years. Virtualization enables the engineer to share the hardware across different software instances. Docker is not really a virtualization software, but it is conceptually the same.

With a pure virtualization solution, a new OS runs on top of a hypervisor sitting on top of an existing operating system (host OS). Running the full OS means that we can be consuming a few gigabytes of hard drive in order to replicate the full stack from the kernel to the filesystem, which usually consumes a good chunk of resources. The structure of a virtualization solution is shown in the following image:



Layers diagram for a virtual machine environment

With Docker, we only replicate the filesystem and binaries so that there is no need to run the full stack of the OS where we don't need it. Docker images are usually a few hundreds of megabytes, instead of gigabytes, and they are quite lightweight, therefore, we can run some containers on the same machine. The previous structure using Docker is shown as follows:



Layers diagram for Docker

With Docker, we also eliminate one of the biggest problems of software deployment, that is, **the configuration management**.

We are switching a complicated per-environment configuration management, where we need to worry about how the application is deployed/configured into a container that is basically like a software package that can be installed in any Docker-ready machine.

The only Docker-ready OS nowadays is Linux, as Docker needs to make use of the advanced kernel features, forcing Windows and Mac users to run a virtual machine with Linux in order to provide support to run Docker containers.

Setting up the container

Docker comes with a very powerful and familiar way (for developers) of configuring the containers.

You can create containers based on an existing image (there are thousands of images on the Internet) and then modify the image to fulfil your needs by adding new software packages or altering the filesystem.

Once we are satisfied with it, we can use the new version of the image to create our containers using a version control system similar to **Git**.

However, we need to understand how Docker works first.

Installing Docker

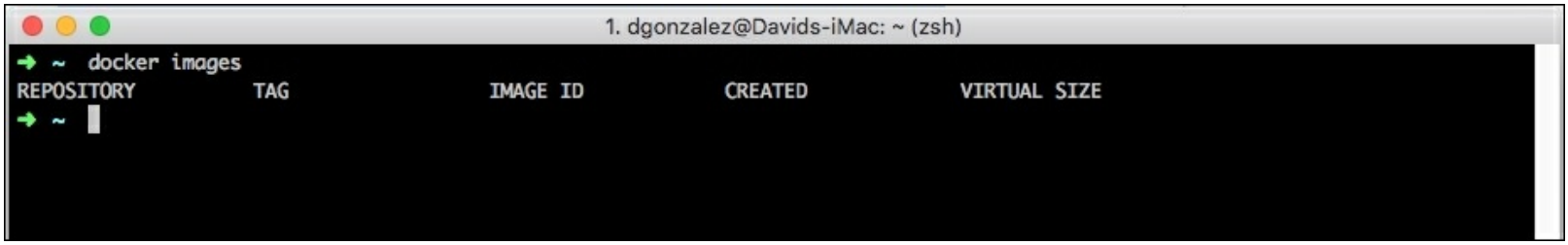
As it was mentioned before, Docker needs a virtual machine to provide support on Mac and Windows, therefore, the installation on these systems may vary. The best way to install Docker on your system is to go to the official website and follow the steps:

https://docs.docker.com/engine/installation/

At the moment, it is a very active project, so you can expect changes every few weeks.

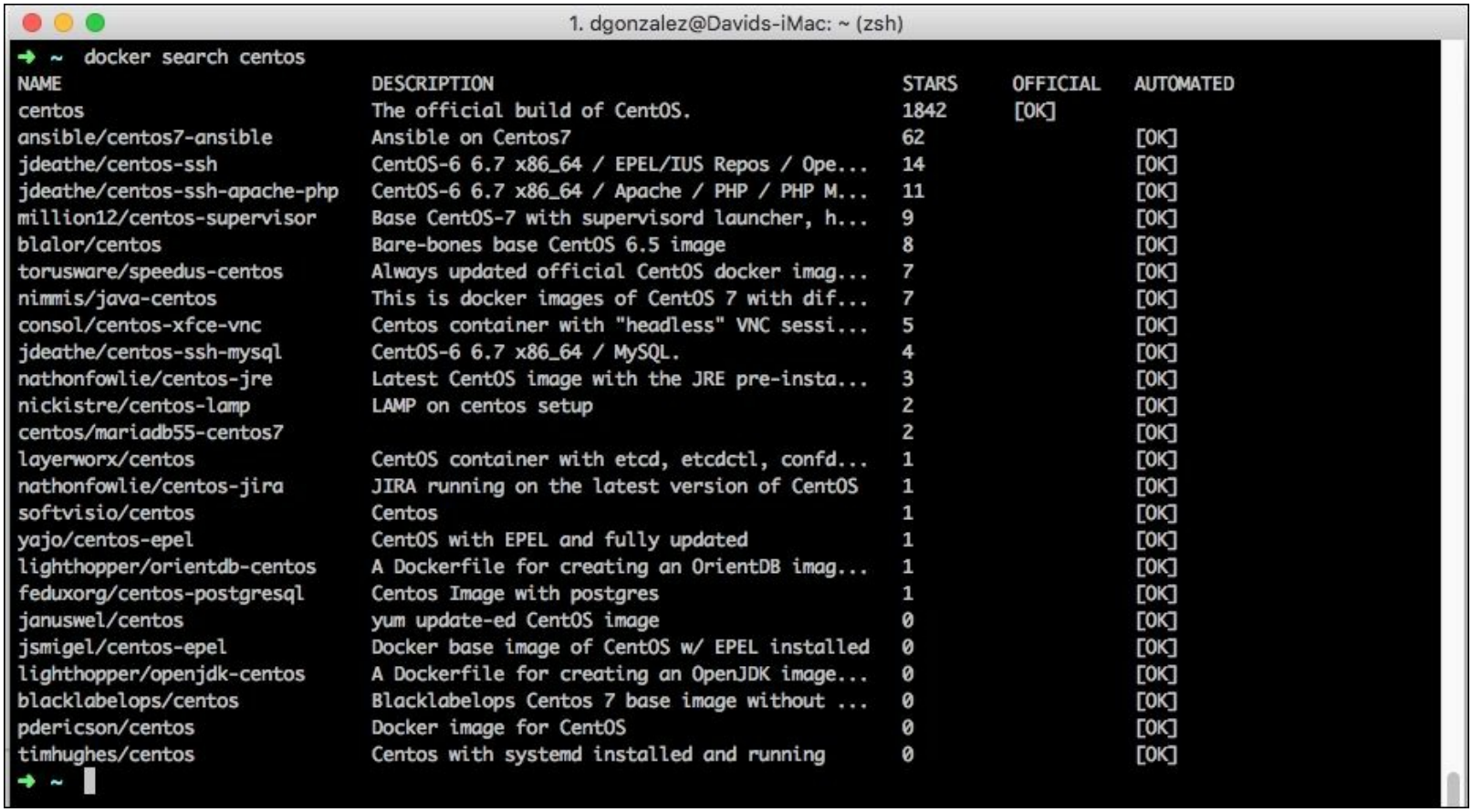
Choosing the image

By default, Docker comes with no images. We can verify this by running `docker images` on the terminal, which will produce an output very similar to the following screenshot:



It is an empty list. There are no images stored in the local machine. The first thing we need to do is search for an image. In this case, we are going to use **CentOS** as our base for creating the images. CentOS is very close to Red Hat Enterprise Linux, which seems to be one of the most extended distributions of Linux in the industry. They provide great support and there is plenty of information available on the Internet to troubleshoot problems.

Let's search for a CentOS image as follows:



As you can see, there is a long list of images based on CentOS, but only the first one is official.

This list of images is coming from something called the **Registry** in the Docker world. A Docker Registry is a simple repository of images available to the general public. You can also run your own Registry in order to prevent your images from going to the general one.

Note

More information can be found at the following link:

https://docs.docker.com/registry/

There is one column in the table in the preceding screenshot that should have caught your attention almost immediately, the column called **STARS**. This column represents the rating given by the users for a given image. We can narrow the search based on the number of stars that the users have given to an image by using the `-s` flag.

If you run the following command, you will see a list of images rated with 1000 or more stars:

```
docker search -s 1000 centos
```

Tip

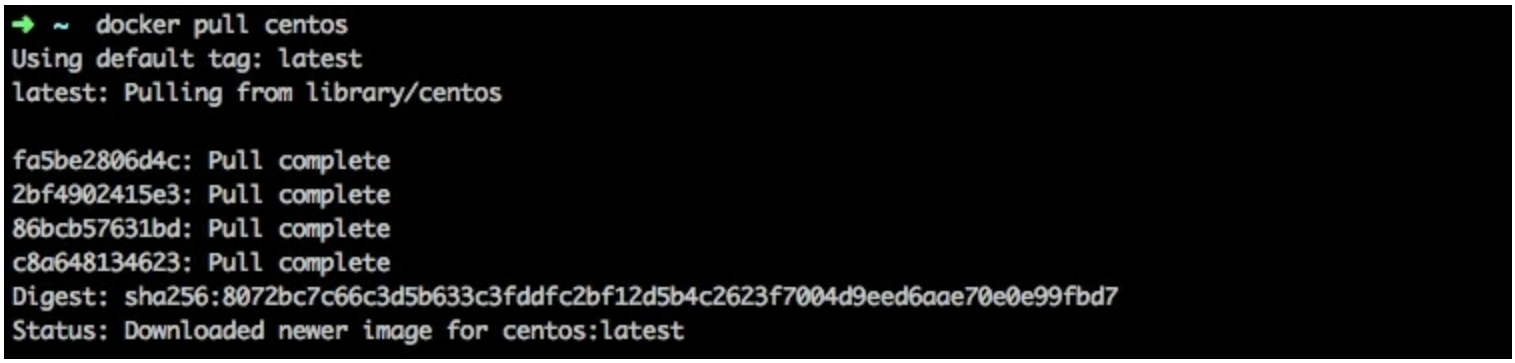
Be careful with the images you choose, there is nothing preventing a user to create an image with

malicious software.

In order to fetch the CentOS image to the local machine, we need to run the following command:

```
docker pull centos
```

The output produced will be very similar to the following image:



Once the command finishes, if we run Docker images again, we can see that **centos** is now appearing in the following list:



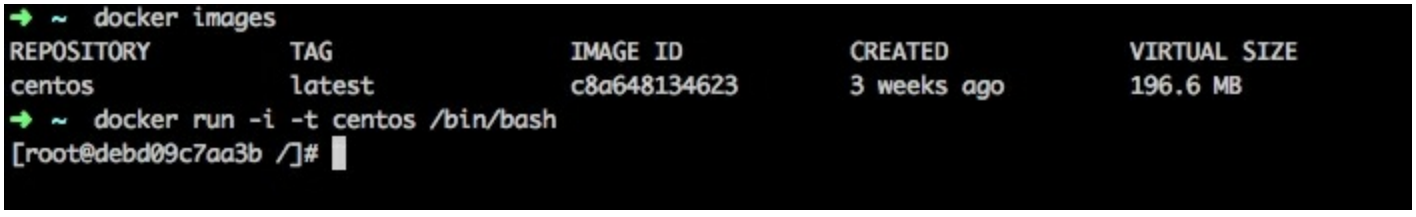
As we specified earlier, Docker does not use the full image, but it uses a reduced version of it, only virtualizing the last few layers of the OS. You can clearly see it, as the size of the image is not even 200 MB, which for a full version of CentOS, can go up to a few GB.

Running the container

Now that we have a copy of the image in our local machine, it is time to run it:

```
docker run -i -t centos /bin/bash
```

This will produce the following output:



As you can see, the prompt of the terminal has changed to something like `root@debd09c7aa3b`, which means that we are inside the container.

From now on, every single command that we run will be executed inside a contained version of CentOS Linux.

There is another interesting command in Docker:

```
docker ps
```

If we run this command in a new terminal (without exiting from the running container), we will get the following output:

```
➜ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
62e7336a4627        centos              "/bin/bash"         8 minutes ago       Up 8 minutes                            prickly_bartik
➜ ~
```

This output is self explanatory; it is an easy way to see what is going on in our Docker containers.

Installing the required software

Let's install Node.js in the container:

```
curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -
```

This command will pull and execute the setup script for Node.js.

This will produce an output very similar to the following image:

```
[root@0ab9098011c0 /]# curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -

## Installing the NodeSource Node.js 4.x LTS Argon repo...


## Inspecting system...

+ rpm -q --whatprovides redhat-release || rpm -q --whatprovides centos-release || rpm -q --whatprovides cloudlinux-release || rpm -q --whatprovides sl-release
+ uname -m

## Confirming "el7-x86_64" is supported...

+ curl -sLf -o /dev/null 'https://rpm.nodesource.com/pub_4.x/el/7/x86_64/nodesource-release-el7-1.noarch.rpm'

## Downloading release setup RPM...

+ mktemp
+ curl -sL -o '/tmp/tmp.bdco6IfiZY' 'https://rpm.nodesource.com/pub_4.x/el/7/x86_64/nodesource-release-el7-1.noarch.rpm'

## Installing release setup RPM...

+ rpm -i --nosignature --force '/tmp/tmp.bdco6IfiZY'

## Cleaning up...

+ rm -f '/tmp/tmp.bdco6IfiZY'

## Checking for existing installations...

+ rpm -qa 'node|npm' | grep -v nodesource

## Run `yum install -y nodejs` (as root) to install Node.js 4.x LTS Argon and npm.
## You may also need development tools to build native addons:
##   `yum install -y gcc-c++ make`

[root@0ab9098011c0 /]#
```

Follow the instructions, as this will install node:

```
yum install -y nodejs
```

It is highly recommended to install the development tools, as the installation process of a few modules requires a compilation step. Let's do it:

```
yum install -y gcc-c++ make
```

Once the command finishes, we are ready to run the node applications inside our container.

Saving the changes

In the Docker world, an image is the configuration for a given container. We can use the image as a template to run as many containers as we want, however first, we need to save the changes made in the previous section.

If you are a software developer, you probably are familiar with control version systems such as CVS, Subversion, or Git. Docker was built with their philosophy in mind—a container can be treated like a versionable software component and then changes can be committed.

In order to do it, run the following command:

```
docker ps -a
```

This command will show a list of containers that have run in the past, as shown in the following image:

```
➜ ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                          PORTS               NAMES
cf0725a25148        centos              "/bin/bash"         11 seconds ago      Up 10 seconds                                       reverent_ritchie
62e7336a4627        centos              "/bin/bash"         15 minutes ago      Exited (0) About a minute ago                       prickly_bartik
7bbb50bb7236        centos              "/bin/bash"         16 minutes ago      Exited (0) 16 minutes ago                           jolly_raman
482d0ef324f2        centos              "/bin/bash"         22 minutes ago      Exited (127) 17 minutes ago                         adoring_hypatia
debd09c7aa3b        centos              "/bin/bash"         47 hours ago        Exited (137) 24 minutes ago                         nostalgic_stallman
ed948a19739b        centos              "/bin/bash"         47 hours ago        Exited (0) 47 hours ago                             jolly_mcclintock
c1f8550c09bb        1d073211c498        "/bin/bash"         10 weeks ago        Exited (0) 10 weeks ago                             fervent_torvalds
```

In my case, there are few containers, but the interesting one in this case is the second; this is where Node.js is installed.

Now, we need to commit the status of the container in order to create a new image with our changes. We do it by running the following command:

```
docker commit -a dgonzalez 62e7336a4627 centos-microservices:1.0
```

Let's explain the command:

- The `-a` flag indicates the author. In this case, `dgonzalez`.
- The following parameter is `container id`. As we indicated earlier, the second container has the corresponding ID `62e7336a4627`.
- The third parameter is a combination of the name given to the new image and the tag of the image. The tagging system can be very powerful when we are dealing with quite a few images, as it can get really complicated to identify small variations between them.

It might take a few seconds, but after finishing, the output of the command must be very similar to the following image:

```
➜ ~ docker commit -a dgonzalez 62e7336a4627 centos-microservices:1.0
75d9f196b7b4181f41a09163d8177eefcc57649af1ccac9dbcc3af1e2a56bea6
```

This is the indication that we have a new image in our list with our software installed. Run `docker images` again and the output will confirm it, as shown in the following image:

```
➜ ~ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos-microservices    1.0                 75d9f196b7b4        About a minute ago  306.4 MB
centos                  latest              c8a648134623        3 weeks ago         196.6 MB
```

In order to run a container based on the new image, we can run the following command:

```
docker run -i -t centos-microservices:1.0 /bin/bash
```

This will give us access to the shell in the container, and we can confirm that Node.js is installed by running `node -v`, which should output the version of Node, in this case, 4.2.4.

Deploying Node.js applications

Now, it is time to deploy Node.js applications inside the container. In order to do it, we are going to need to expose the code from our local machine to the Docker container.

The correct way of doing it is by mounting a local folder in the Docker machine, but first, we need to create the small application to be run inside the container, as follows:

```
var express = require('express');
var myApplication = express();

app.get('/hello', function (req, res) {
  res.send('Hello Earth!');
});

var port = 80;

app.listen(port, function () {
  console.log('Listeningistening on port '+ port);
});
```

It is a simple application, using Express that basically renders `Hello Earth!` into a browser. If we run it from a terminal and we access `http://localhost:80/hello`, we can see the results.

Now, we are going to run it inside the container. In order to do it, we are going to mount a local folder as a volume in the Docker container and run it.

Docker comes from the experience of sysadmins and developers that have lately melted into a role called DevOps, which is somewhere in between them. Before Docker, every single company had its own way of deploying apps and managing configurations, so there was no consensus on how to do things the right way.

Now with Docker, the companies have a way to provide uniformity to deployments. No matter what your business is, everything is reduced to build the container, deploy the application, and run the container in the appropriate machine.

Let's assume that the application is in the `/apps/test/` folder. Now, in order to expose it to the container, we run the following command:

```
docker run -i -t -v /app/test:/test_app -p 8000:3000 centos-microservices:1.0
/bin/bash
```

As you can see, Docker can get very verbose with parameters, but let's explain them, as follows:

- The `-i` and `-t` flags are familiar to us. They capture the input and send the output to a terminal.
- The `-v` flag is new. It specifies a volume from the local machine and where to mount it in the container. In this case, we are mounting `/apps/test` from the local machine into `/test_app`. Please note the colon symbol to separate the local and the remote path.

- The `-p` flag specifies the port on the local machine that will expose the remote port in the container. In this case, we expose the port `3000` in the container through the port `8000` in the Docker machine, so accessing `docker-machine:8000` from the host machine will end up accessing the port `3000` in the container.
- The `centos-microservices:1.0` is the name and tag of the image that we have created in the preceding section.
- The `/bin/bash` is the command that we want to execute inside the container. The `/bin/bash` is going to give us access to the prompt of the system.

If everything worked well, we should have gotten access to the system prompt inside the container, as shown in the following image:

```
[root@c079d5f180da /]# cd /test_app/
[root@c079d5f180da test_app]# ls
node_modules  small-script.js
[root@c079d5f180da test_app]#
```

As you can see in the image, there is a folder called `/test_app` that contains our previous application, called `small-script.js`.

Now, it is time to access to the app, but first, let's explain how Docker works.

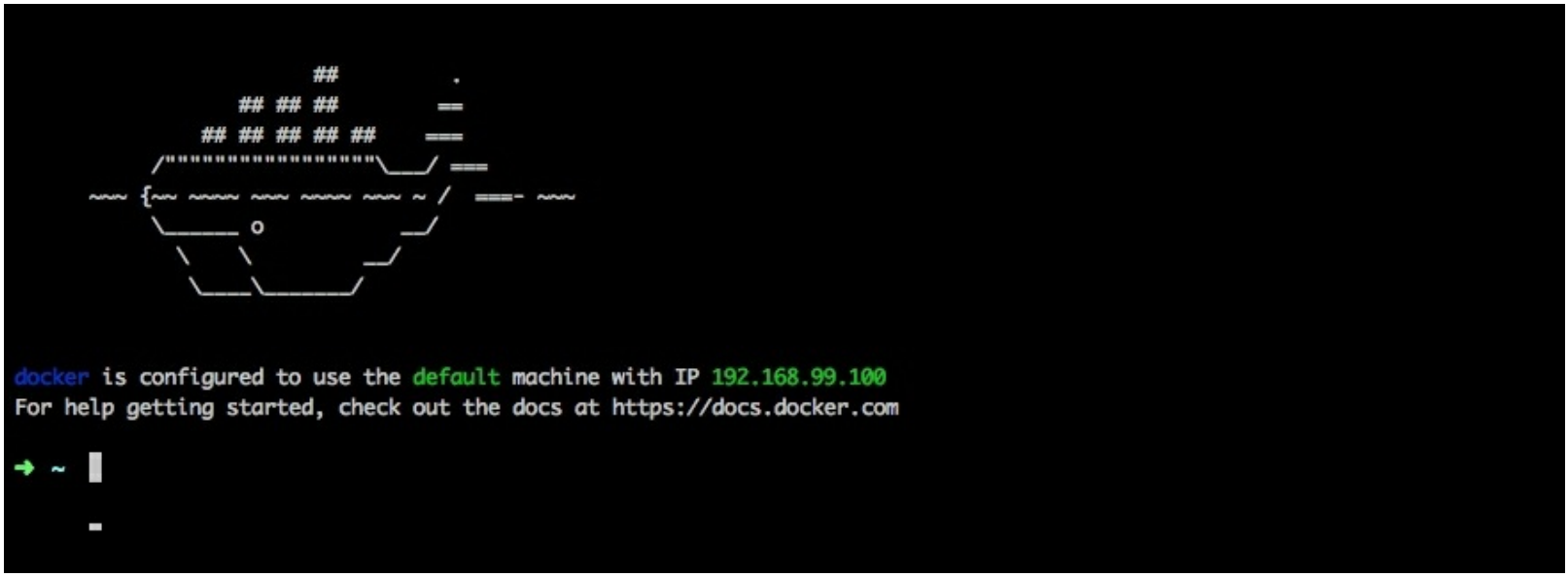
Docker is written in **Go**, which is a modern language created by Google, grouping all the benefits from a compiled language such as C++ with all the high-level features from a modern language such as Java.

It is fairly easy to learn and not hard to master. The philosophy of Go is to bring all the benefits of an interpreted language, such as reducing the compilation times (the complete language can be compiled in under a minute) to a compiled language.

Docker uses very specific features from the Linux kernel that forces Windows and Mac users to use a virtual machine to run Docker containers. This machine used to be called **boot2docker**, but the new version is called **Docker Machine**, which contains more advanced features such as deploying containers in remote virtual machines. For this example, we will only use the local capabilities.

Given that, if you run the app from within the container located in the `/test_app/` folder, and you are in Linux, accessing `http://localhost:8000/`, it would be enough to get into the application.

When you are using Mac or Windows, Docker is running either in the Docker Machine or boot2docker so that the IP is given by this virtual machine, which is shown when the Docker terminal starts, as shown in the following image:



As you can see, the IP is `192.168.99.100`, so in order to access our application, we need to visit the `http://192.168.99.100:9000/` URL.

Automating Docker container creation

If you remember, in the previous chapters, one of the most important concepts was automation. Automation is the key when working with microservices. Instead of operating one server, you probably will need to operate few dozens, reaching a point where you are almost fully booked on day-to-day tasks.

Docker designers had that in mind when allowing the users to create containers from a script written in a file called `Dockerfile`.

If you have ever worked on coding C or C++, even in college, you are probably familiar with `Makefiles`. A `Makefile` file is a script where the developer specifies the steps to automatically build a software component. Here is an example:

```
all: my-awesome-app

my-awesome-app: main.o external-module.o app-core.o
  g++ main.o external-module.o app-core.o -o my-awesome-app


main.o: main.cc
  g++ -c main.cc


external-module.o: external-module.cc
  g++ -c external-module.cc


app-core.o: app-core.cc
  g++ -c hello.cc

clean:
  rm *.o my-awesome-app
```

The preceding `Makefile` contains a list of tasks and dependencies to be executed. For example, if we execute `make clean` on the same folder where the `Makefile` file is contained, it will remove the executable and all the files ending with `o`.

`Dockerfile`, unlike `Makefile`, is not a list of tasks and dependencies (even though the concept is the same), it is a list of instructions to build a container from scratch to a ready status.

Let's see an example:

```
FROM centos
MAINTAINER David Gonzalez
RUN curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -
RUN yum -y install nodejs
```

These small few preceding lines are enough to build a container having Node.js installed.

Let's explain it in the following:

- First, we choose the base image. In this case, it is `centos` as we used before. For doing this, we use the `FROM` command and then the name of the image.
- `MAINTAINER` specifies the name of the person who created the container. In this case, it is `David Gonzalez`.
- `RUN`, as its name indicates, runs a command. In this case, we use it twice: once to add the repository to `yum`, and then to install Node.js.

Dockerfiles can contain a number of different commands. The documentation for them is pretty clear, but let's take a look at the most common (aside from the ones seen before):

- `CMD`: This is similar to run, but it actually gets executed after building the command. `CMD` is the command to be used to start an application once the container is instantiated.
- `WORKDIR`: This is to be used in conjunction with `CMD`, it is the command used to specify the work directory for the next `CMD` command.
- `ADD`: This command is used to copy files from the local filesystem to the container instance filesystem. In the previous example, we can use `ADD` to copy the application from the host machine into the container, run `npm install` with the `CMD` command, and then run the app once again with the `CMD` command. It can also be used to copy the content from a URL to a destination folder inside the container.
- `ENV`: This is used to set environment variables. For example, you could specify a folder to store files uploaded by passing an environment variable to the container, as follows:

  ```
  ENV UPLOAD_PATH=/tmp/
  ```

- `EXPOSE`: This is used to expose ports to the rest of the containers in your cluster.

As you can see, the **domain-specific language** (**DSL**) of `Dockerfiles` is quite rich and you can pretty much build every system required. There are hundreds of examples on the Internet to build pretty much everything: MySQL, MongoDB, Apache servers, and so on.

It is strongly recommended to create containers through `Dockerfiles`, as it can be used as a script to replicate and make changes to the containers in the future, as well as being able to automatically deploy our software without much manual intervention.

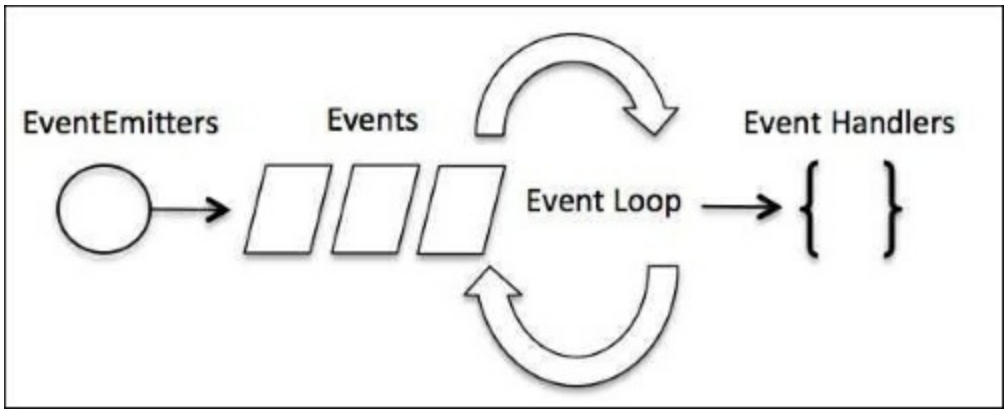
Node.js event loop – easy to learn and hard to master

We all know that Node.js is a platform that runs applications in a single-threaded way; so, why don't we use multiple threads to run applications so that we can get the benefit of multicore processors?

Node.js is built upon a library called **libuv**. This library abstracts the system calls, providing an asynchronous interface to the program that uses it.

I come from a very heavy Java background, and there, everything is synchronous (unless you are coding with some sort of non-blocking libraries), and if you issue a request to the database, the thread is blocked and resumed once the database replies with the data.

This usually works fine, but it presents an interesting problem: a blocked thread is consuming resources that could be used to serve other requests. The event loop of Node.js is shown in the following figure:



This is the Node.js event loop diagram

JavaScript is, by default, an event-driven language. It executes the program that configures a list of event handlers that will react to given events, and after that, it just waits for the action to take place.

Let's take a look at a very familiar example:

```
<div id="target">
  Click here
</div>
<div id="other">
  Trigger the handler
</div>
```

Then the JavaScript code is as follows:

```
$( "#target" ).click(function() {
  alert( "Handler for .click() called." );
});
```

As you can see, this is a very simple example. HTML that shows a button and snippet of JavaScript code that, using JQuery, shows an alert box when the button is clicked.

This is the key: *when the button is clicked*.

Clicking a button is an event, and the event is processed through the event loop of JavaScript using a handler specified in the JavaScript.

At the end of the day, we only have one thread executing the events, and we never talk about parallelism in JavaScript, the correct word is concurrency. So, being more concise, we can say that Node.js programs are highly concurrent.

Your application will always be executed in only one thread, and we need to keep that in mind while coding.

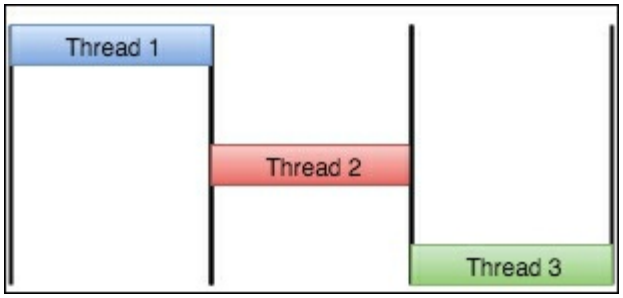
If you have been working in Java or .NET or any other language/frameworks designed and implemented with thread-blocking techniques, you might have observed that Tomcat, when running an application, spawns a number of threads listening to the requests.

In the Java world, each of these threads are called **workers**, and they are responsible to handle the request from a given user from the beginning to the end. There is one type of data structure in Java that takes the benefit of it. It is called **ThreadLocal** and it stores the data in the local thread so that it can be recovered later on. This type of storage is possible because the thread that starts the request is also responsible to finish it, and if the thread is executing any blocking operation (such as reading a file or accessing a database), it will wait until it is completed.

This is usually not a big deal, but when your software relies heavily on I/O, the problems can become serious.

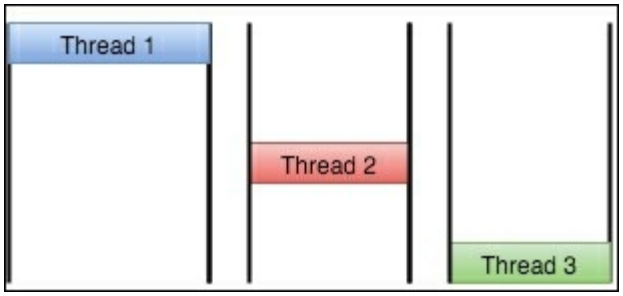
Another big point in favor of the non-blocking model of Node.js is the lack of context switch.

When the CPU switches one thread with another, what happens is that all the data in the registers, and other areas of the memory, is stacked and allows the CPU to switch the context with a new process that has its own data to be placed in there, as shown in the following image:



This is a diagram showing context switching in threads from the theoretical point of view.

This operation takes time, and this time is not used by the application. It simply gets lost. In Node.js, your application runs in only one thread, so there is no such context switching while running (it is still present in the background, but hidden to your program). In the following image, we can see what happens in the real world when a CPU switches a thread:



This is a diagram showing context switching in threads from the practical (shows the dead times) point of view.

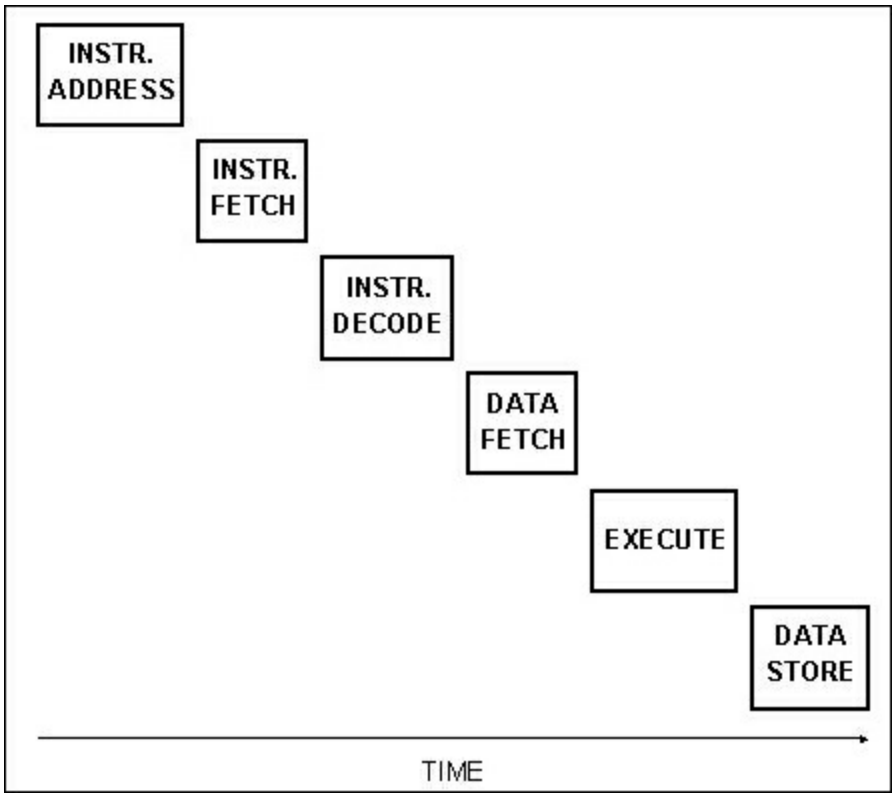
Clustering Node.js applications

By now, you know how Node.js applications work, and certainly, some of the readers may have a question that if the app runs on a single thread, then what happens with the modern multicore processors?

Before answering this question, let's take a look at the following scenario.

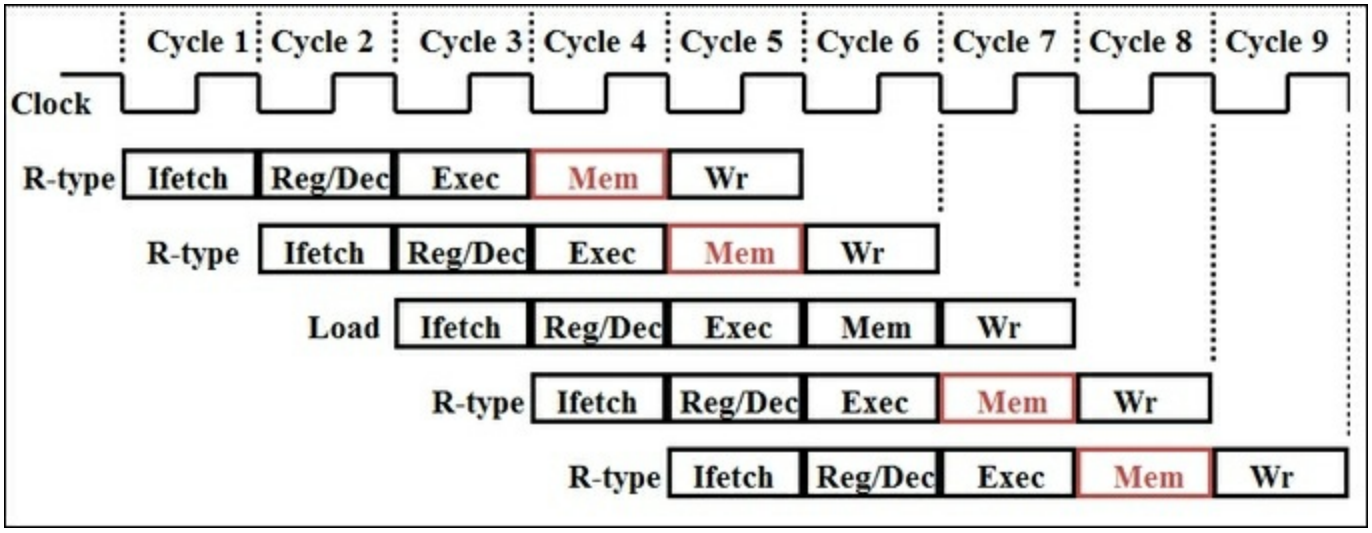
When I was in high school, there was a big technology leap in CPUs: the segmentation.

It was the first attempt to introduce parallelism at the instruction level. As you probably are aware, the CPU interprets assembler instructions and each of these instructions are composed of a number of phases, as shown in the following diagram:



Before the Intel 4x86, the CPUs were executing one instruction at the time, so taking the instruction model from the preceding diagram, any CPU could only execute one instruction every six CPU cycles.

Then, the segmentation came into play. With a set of intermediate registers, the CPU engineers managed to parallelize the individual phases of instructions so that in the best-case scenario, the CPUs are able to execute one instruction per cycle (or nearly), as shown in the following diagram:



The image describes the execution of instructions in a CPU with a segmented pipeline

This technical improvement led to faster CPUs and opened the door to native hardware multithreading, which led to the modern n-core processors that can execute a large number of parallel tasks, but when we are running Node.js applications, we only use one core.

If we don't cluster our app, we are going to have a serious performance degradation when compared to other platforms that take the benefit of the multiple cores of a CPU.

However, this time we are lucky, PM2 already allows you to cluster Node.js apps to maximize the usage of your CPUs.

Also, one of the important aspects of PM2 is that it allows you to scale applications without any downtime.

Let's run a simple app in the cluster mode:

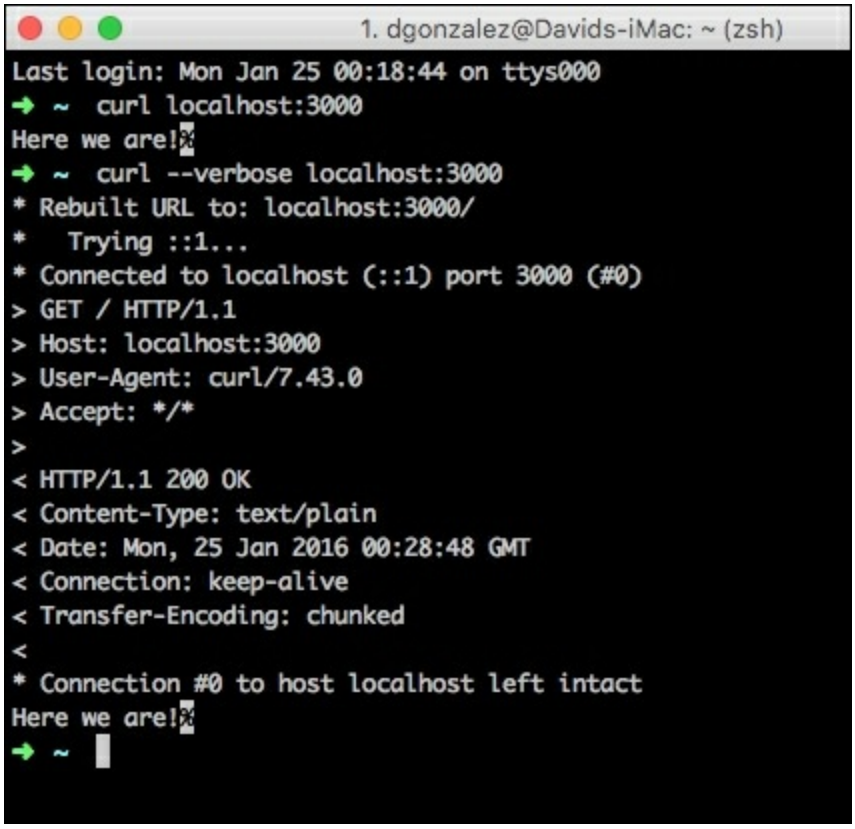
```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  response.write('Here we are!')
  response.end();
}).listen(3000);
```

This time we have used the native HTTP library for Node.js in order to handle the incoming HTTP requests.

Now we can run the application from the terminal and see how it works:

```
node app.js
```

Although it does not output anything, we can curl to the `http://localhost:3000/` URL in order to see how the server responds, as shown in the following screenshot:



As you can see, Node.js has managed all the HTTP negotiation and it has also managed to reply with the `Here we are!` phrase as it was specified in the code.

This service is quite trivial, but it is the principle on which more complex web services work, so we need to cluster the web service to avoid bottlenecks.

Node.js has one library called `cluster` that allows us to programmatically cluster our application, as follows:

```
var cluster = require('cluster');
var http = require('http');
var cpus = require('os').cpus().length;

// Here we verify if the we are the master of the cluster: This is the root
process
// and needs to fork al the childs that will be executing the web server.
if (cluster.isMaster) {
  for (var i = 0; i < cpus; i++) {
    cluster.fork();
  }

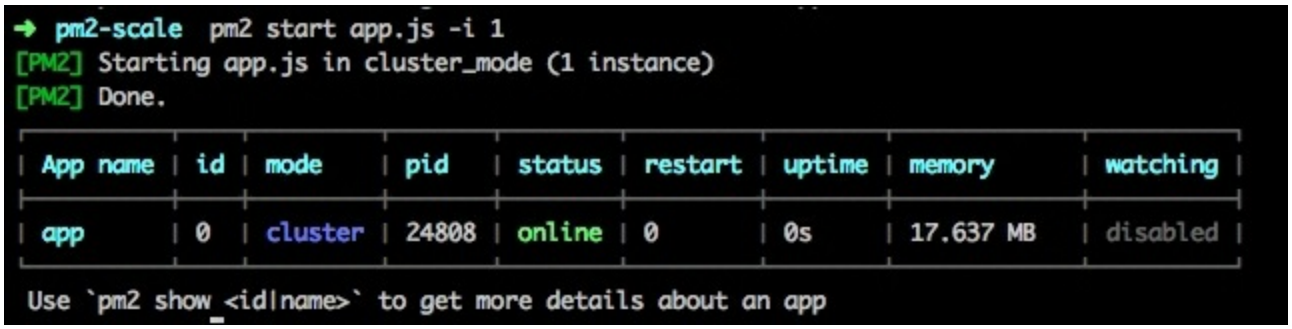
  cluster.on('exit', function (worker, code, signal) {
    console.log("Worker " + worker.proces.pid + " has finished.");
  });
```

```
} else {
  // Here we are on the child process. They will be executing the web server.
  http.createServer(function (request, response) {
    response.writeHead(200);
    response.end('Here we are!d\n');
  }).listen(80);
}
```

Personally, I find it much easier to use specific software such as PM2 to accomplish effective clustering, as the code can get really complicated while trying to handle the clustered instances of our app.

Given this, we can run the application through PM2 as follows:

```
pm2 start app.js -i 1
```



The `-i` flag in PM2, as you can see in the output of the command, is used to specify the number of cores that we want to use for our application.

If we run `pstree`, we can see the process tree in our system and check whether PM2 is running only one process for our app, as shown in the following image:



In this case, we are running the app in only one process, so it will be allocated in one core of the CPU.

In this case, we are not taking advantage of the multicore capabilities of the CPU that is running the

app, but we still get the benefit of restarting the app automatically if one exception bubbles up from our algorithm.

Now, we are going to run our application using all the cores available in our CPU so that we maximize the usage of it, but first, we need to stop the cluster:

```
pm2 stop all
```

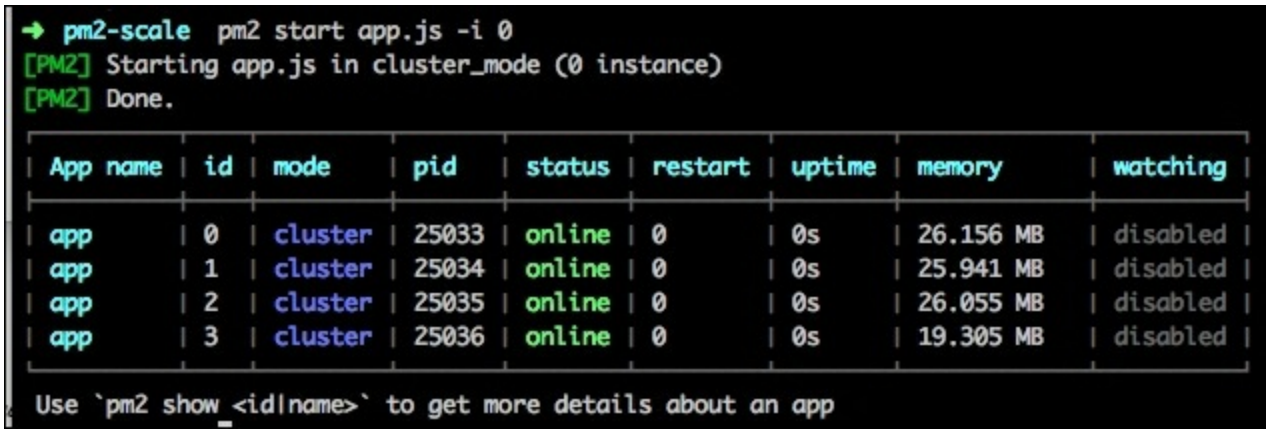


PM2, after stopping all the services

```
pm2 delete all
```

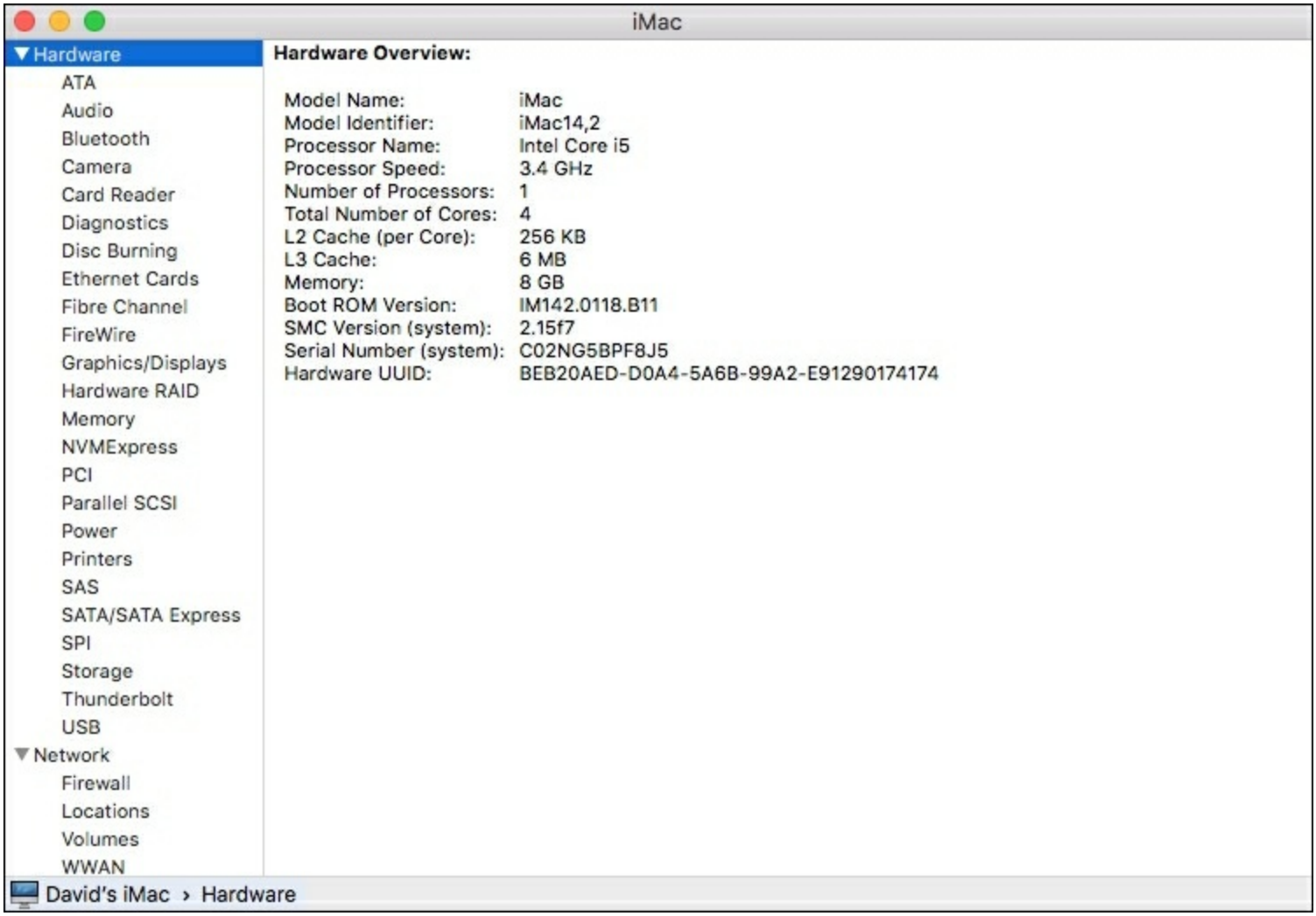
Now, we are in a position to rerun the application using all the cores of our CPU:

```
pm2 start app.js -i 0
```



PM2 showing four services running in a cluster mode

PM2 has managed to guess the number of CPUs in our computer, in my case, this is an iMac with four cores, as shown in the following screenshot:



As you can see in `pstree`, PM2 started four threads at the OS level, as shown in the following image:



When clustering an application, there is an unwritten rule about the number of cores that an application should be using and this number is the number of cores minus one.

The reason behind this number is the fact that the OS needs some CPU power so that if we use all the CPUs in our application, once the OS starts carrying on with some other tasks, it will force context switching as all the cores will be busy and this will slow down the application.

Load balancing our application

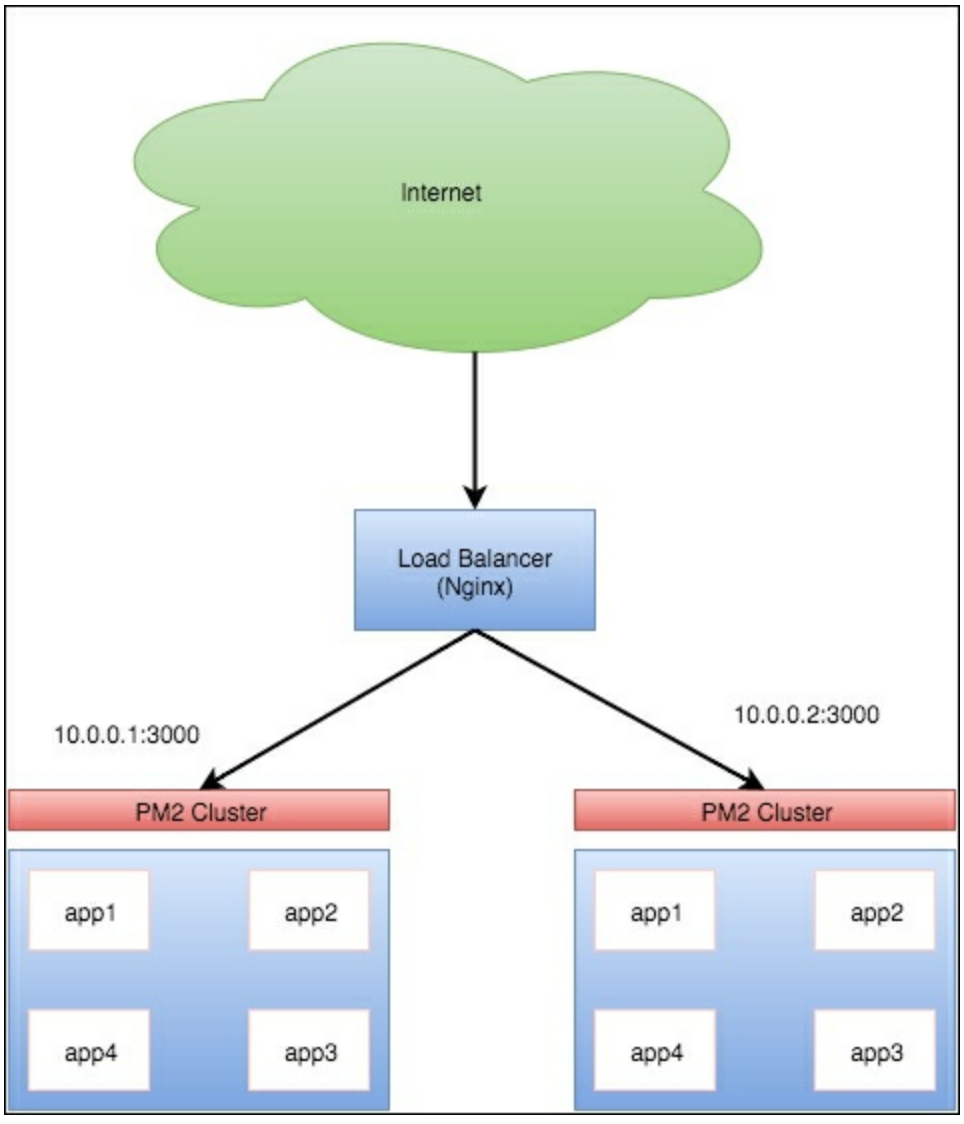
Sometimes, clustering our app is not enough and we need to scale our application horizontally.

There are a number of ways to horizontally scale an app. Nowadays, with cloud providers such as Amazon, every single provider has implemented their own solution with a number of features.

One of my preferred ways of implementing the load balancing is using **NGINX**.

NGINX is a web server with a strong focus on the concurrency and low memory usage. It is also the perfect fit for Node.js applications as it is highly discouraged to serve static resources from within a Node.js application. The main reason is to avoid the application from being under stress due to a task that could be done better with another software, such as NGINX (which is another example of specialization).

However, let's focus on load balancing. The following figure shows how NGINX works as a load balancer:



As you can see in the preceding diagram, we have two PM2 clusters load balanced by an instance of NGINX.

The first thing we need to do is know how NGINX manages the configuration.

On Linux, NGINX can be installed via `yum`, `apt-get`, or any other package manager. It can also be built from the source, but the recommended method, unless you have very specific requirements, is to use a package manager.

By default, the main configuration file is `/etc/nginx/nginx.conf`, as follows:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
  worker_connections  1024;
}

http {
  include                   /etc/nginx/mime.types;
  default_type              application/octet-stream;

  log_format          main '$remote_addr - $remote_user [$time_local] "$request"
'
     '$status $body_bytes_sent "$http_referer" '
     '"$http_user_agent" "$http_x_forwarded_for" '
     '$request_time';

  access_log                /var/log/nginx/access.log  main;
  server_tokens             off;
  sendfile                  on;
  #tcp_nopush               on;
  keepalive_timeout         65s;
  send_timeout              15s;
  client_header_timeout     15s;
  client_body_timeout       15s;
  client_max_body_size      5m;
  ignore_invalid_headers    on;
  fastcgi_buffers           16 4k;
  #gzip                     on;
  include                   /etc/nginx/sites-enabled/*.conf;
}
```

This file is pretty straightforward, it specifies the number of workers (remember, processes to serve requests), the location of error logs, number connections that a worker can have active at the time, and finally, the HTTP configuration.

The last line is the most interesting one: we are informing NGINX to use `/etc/nginx/sites-`

`enabled/*.conf` as potential configuration files.

With this configuration, every file ending in `.conf` under the specified folder is going to be part of the NGINX configuration.

As you can see, there is a default file already existing there. Modify it to look as follows:

```
http {
  upstream app {
    server 10.0.0.1:3000;
    server 10.0.0.2:3000;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://app;
    }
  }
}
```

This is all the configuration we need to build a load balancer. Let's explain it in the following:

- The `upstream app` directive is creating a group of services called `app`. Inside this directive, we specify two servers as we've seen in the previous image.
- The `server` directive specifies to NGINX that it should be listening to all the requests from port `80` and passing them to the group of upstream called `app`.

Now, how does NGINX decide to send the request to which computer?

In this case, we could specify the strategy used to spread the load.

By default, NGINX, when there is not a balancing method specifically configured, uses **Round Robin**.

One thing to bear in mind is that if we use round robin, our application should be stateless as we won't be always hitting the same machine, so if we save the status in the server, it might not be there in the following call.

Round Robin is the most elementary way of distributing load from a queue of work into a number of workers; it rotates them so that every node gets the *same amount of requests*.

There are other mechanisms to spread the load, as follows:

```
  upstream app {
    least_conn;
    server 10.0.0.1:3000;
    server 10.0.0.2:3000;
  }
```

Least connected, as its name indicates, sends the request to the least connected node, equally

distributing the load between all the nodes:

```
upstream app {
  ip_hash;
  server 10.0.0.1:3000;
  server 10.0.0.2:3000;
}
```

IP hashing is an interesting way of distributing the load. If you have ever worked with any web application, the concept of sessions is something present in almost any application. In order to remember who the user is, the browser sends a cookie to the server, which has stored who the user is in memory and what data he/she needs/can be accessed by that given user. The problem with the other type of load balancing is that we are not guaranteed to always hit the same server.

For example, if we are using Least connected as a policy for balancing, we could hit the server one in the first load, but then hit a different server on subsequent redirections that will result in the user not being displayed with the right information as the second server won't know who the user is.

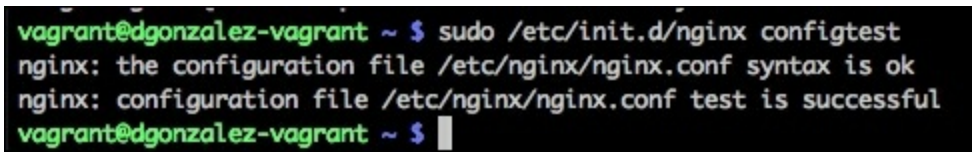
With IP hashing, the load balancer will calculate a hash for a given IP. This hash will somehow result in a number from 1 to *N*, where *N* is the number of servers, and then, the user will always be redirected to the same machine as long as they keep the same IP.

We can also apply a weight to the load balancing, as follows:

```
upstream app {
  server 10.0.0.1:3000 weight=5;
  server 10.0.0.2:3000;
}
```

This will distribute the load in such way that, for every six requests, five will be directed to the first machine and one will be directed to the second machine.

Once we have chosen our preferred load balancing method, we can restart NGINX for the changes to take effect, but first, we want to validate them as shown in the following image:



As you can see, the configuration test can be really helpful in order to avoid configuration disasters.

Once NGINX has passed `configtest`, it is guaranteed that NGINX will be able to `restart/start/reload` without any syntax problem, as follows:

```
sudo /etc/init.d/nginx reload
```

Reload will gracefully wait until the old threads are done, and then, reload the configuration and route the new requests with the new configuration.

If you are interested in learning about NGINX, I found the following official documentation of NGINX quite helpful:

http://nginx.org/en/docs/

Health check on NGINX

Health checking is one of the important activities on a load balancer. What happens if one of the nodes suffers a critical hardware failure and is unable to serve more requests?

In this case, NGINX comes with two types of health checks: **passive** and **active**.

Passive health check

Here, NGINX is configured as a reverse proxy (as we did in the preceding section). It reacts to a certain type of response from the upstream servers.

If there is an error coming back, NGINX will mark the node as faulty, removing it from the load balancing for a certain period of time before reintroducing it. With this strategy, the number of failures will be drastically reduced as NGINX will be constantly removing the node from the load balancer.

There are a few configurable parameters, such as `max_fails` or `fail_timeout`, where we can configure the amount of failures required to mark a node as invalid or the time out for requests.

Active health check

Active health checks, unlike passive health checks, actively issue connections to the upstream servers to check whether they are responding correctly to the experiencing problems.

The most simplistic configuration for active health checks in NGINX is the following one:

```
http {
  upstream app {
    zone app test;
    server 10.0.0.1:3000;
    server 10.0.0.2:3000;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://app;
      health_check;
    }
  }
}
```

There are two new lines in this config file, as follows:

- `health_check`: This enables the active health check. The default configuration is to issue a connection every five seconds to the host and port specified in the `upstream` section.
- `zone app test`: This is required by the NGINX configuration when enabling the health check.

There is a wide range of options to configure more specific health checks, and all of them are available in NGINX configuration that can be combined to satisfy the needs of different users.

Summary

In this chapter, you learned a wide range of technologies that we can use to deploy microservices. By now you know how to build, deploy, and configure software components in such a way that we are able to homogenize a very diverse range of technologies. The objective of this book is to provide you the concepts required to start working with microservices and enable the reader to know how to look for the needed information.

Personally, I have struggled to find a book that provides a summary of all the aspects of the life cycle of microservices and I really hope that this book covers this empty space.

Appendix A. Bibliography

This learning path has been prepared for you to build enterprise-ready implementations of microservices and explore the domain-driven design with its adoption in microservices. It comprises of the following Packt products:

- Mastering Microservices with Java, Sourabh Sharma
- Spring Microservices, Rajesh RV
- Developing Microservices with Node.js,David Gonzalez

Index

A

- ACID
  - atomicity / Decentralization
  - consistency / Decentralization
  - isolation / Decentralization
  - durability / Decentralization
- active health check
  - about / Active health check
- active monitoring
  - about / Simian Army – the active monitoring from Spotify
- ADD command
  - about / Automating Docker container creation
- Advance Messaging Queue Protocol (AMQP) / Monitoring
- agents
  - about / Fleet
- Agile software development
  - URL / The Agile practice
- Airbnb
  - URL / Microservices early adopters
  - about / Microservices early adopters
- Amazon
  - about / Microservices early adopters
  - URL / Microservices early adopters
- Amazon EC2 Container Service (ECS) / Introduction to Docker
- Amazon Machine Image (AMI) / Deployment and delivery – Spinnaker with Aminator
- Amazon Web Services (AWS) / Deployment and delivery – Spinnaker with Aminator
- Aminator / Deployment and delivery – Spinnaker with Aminator
- Angel
  - about / Spring Cloud releases
- AngularJS framework
  - overview / AngularJS framework overview
  - MVC (Model View Controller) / AngularJS framework overview
  - MVVM (Model-View-ViewModel) / MVVM
  - modules / Modules
  - providers / Providers and services
  - services / Providers and services
  - scopes / Scopes
  - controllers / Controllers
  - filters / Filters
  - directives / Directives

    - UI-Router / UI-Router
- AngularJS module
    - config() / Modules
    - run() / Modules
- Apache Camel for Microservice Architectures
    - reference link / References
- Apache Mesos
    - about / Apache Mesos
- API aggregation
    - about / API aggregation
- API versions
    - about / Dependencies and versions
    - exploring / Let's explore more
    - references / Let's explore more
- App / Registration and Discovery service (Eureka service)
- application
    - autoscaling / Autoscaling an application
- Application Binary Interface (ABI) / Overview and mindset
- application build
    - setting up / Setting up the application build
- application build, setting up
    - about / Setting up the application build
    - Maven tool, running / Running the Maven tool
    - execution, Java command used / Executing with the Java command
- application exceptions
    - monitoring, Keymetrics used / Monitoring application exceptions
- application information
    - configuring / Configuring application information
- application metrics
    - about / Monitoring services
- application security
    - about / Application security
- appropriate microservice boundaries
    - establishing / Establishing appropriate microservice boundaries
- Archaius
    - about / Distributed configuration management – Archaius
- architecture, Mesos
    - about / The Mesos architecture
    - master / The Mesos architecture
    - slave / The Mesos architecture
    - ZooKeeper / The Mesos architecture
    - framework / The Mesos architecture
- artifacts, domain-driven design (DDD)

    - entities / Entities
    - value objects (VOs) / Value objects
    - services / Services
    - aggregates / Aggregates
    - repository / Repository
    - factory / Factory
    - modules / Modules
- asymmetric cryptography / SSH – encrypting the communications
- Asynchronous JavaScript and XML (AJAX)
    - about / The future of Node.js
- Atlas / Operational monitoring – Atlas
- auditing
    - about / Auditing
- authentication
    - providing / Authentication and authorization
    - OAuth 2.0 / OAuth 2.0
- authorization
    - providing / Authentication and authorization
    - OAuth 2.0 / OAuth 2.0
- authorization code grant, OAuth 2.0 grant types
    - about / Authorization code grant
    - code requests and responses / Authorization code requests and responses
- autoconfig
    - about / Spring Boot actuators
- automated testing
    - about / The pyramid of automated testing
    - unit tests / Unit tests
    - integration tests / Integration tests
    - end-to-end tests / End-to-end tests
    - need for / How much testing is too much?
- autoscaling
    - about / Understanding the concept of autoscaling
    - benefits / The benefits of autoscaling
    - different autoscaling models / Different autoscaling models
    - in cloud / Autoscaling in the cloud
- autoscaling approaches
    - about / Autoscaling approaches
    - scaling, with resource constraints / Scaling with resource constraints
    - scaling, during specific time periods / Scaling during specific time periods
    - scaling, based on message queue length / Scaling based on the message queue length
    - scaling, based on business parameters / Scaling based on business parameters
    - predictive autoscaling / Predictive autoscaling
- autoscaling ecosystem

    - about / The capabilities required for an autoscaling system
    - Microservices / The capabilities required for an autoscaling system
    - service registry / The capabilities required for an autoscaling system
    - load balancer / The capabilities required for an autoscaling system
    - life cycle manager / The capabilities required for an autoscaling system
- average calculated values metric
    - about / Average calculated values
- AWS
    - setting up / Setting up AWS
- AWS Lambda
    - reference / Imperative architecture evolution

B

- Backend as a Service (BaaS) / Microservice use cases
- behavior-driven development (BDD) / End-to-end tests
- behavior-driven development testing (BDDT) / Mocha
- benefits, microservices
  - about / Microservices benefits
  - polyglot architecture support / Supports polyglot architecture
  - experimentation, enabling / Enabling experimentation and innovation
  - innovation, enabling / Enabling experimentation and innovation
  - elastically scalable / Elastically and selectively scalable
  - selectively scalable / Elastically and selectively scalable
  - substitution, allowing / Allowing substitution
  - build organic systems, enabling / Enabling to build organic systems
  - technology debt, reducing / Helping reducing technology debt
  - coexistence of different versions, allowing / Allowing the coexistence of different versions
  - self-organizing systems, building / Supporting the building of self-organizing systems
  - event-driven architecture, supporting / Supporting event-driven architecture
  - DevOps, enabling / Enabling DevOps
- best-of-the-breed components
  - log shippers / Log shippers
  - log storage / Log storage
  - dashboards / Dashboards
- best practices and principals
  - about / Best practices and principals
  - nanoservice / Nanoservice (not recommended), size, and monolithic
  - size / Nanoservice (not recommended), size, and monolithic
  - monolithic / Nanoservice (not recommended), size, and monolithic
  - continuous integration / Continuous integration and deployment
  - deployment / Continuous integration and deployment
  - system/end-to-end test automation / System/end-to-end test automation
  - self-monitoring / Self-monitoring and logging
  - logging / Self-monitoring and logging
  - separate data store,for microservice / A separate data store for each microservice
  - transaction boundaries / Transaction boundaries
- bill of materials (BOM) / Examining the POM file
- Bin packing algorithm / What does cluster management do?
- black listing
  - about / Input validation
- Bluebird
  - URL / The code, Seneca and promises
- bluebird / Single-threaded applications and exceptions
- boot2docker

    - about / Deploying Node.js applications
- Boot2Docker / Introduction to Docker
- bounded context
    - about / Establishing appropriate microservice boundaries, Replaceability
- Brixton
    - about / Spring Cloud releases
- BrownField PSS
    - environment, setting up for / Setting up the environment for BrownField PSS
- BrownField PSS architecture
    - defining / Summarizing the BrownField PSS architecture
- BrownField PSS microservices
    - limitations / Understanding the gaps in BrownField PSS microservices
- BrownField PSS microservices, autoscaling
    - about / Autoscaling BrownField PSS microservices
    - capabilities / The capabilities required for an autoscaling system
    - custom life cycle manager, implementing with Spring Boot / Implementing a custom life cycle manager using Spring Boot
    - deployment topology / Understanding the deployment topology
    - execution flow / Understanding the execution flow
    - life cycle manager code / A walkthrough of the life cycle manager code
    - life cycle manager, running / Running the life cycle manager
- BrownField PSS services
    - preparing / Preparing BrownField PSS services
    - deploying / Deploying BrownField PSS services
    - reviewing / Reviewing the deployment
- BrownField services
    - running, on EC2 / Running BrownField services on EC2
- building blocks, domain-driven design (DDD)
    - about / Building blocks
    - ubiquitous language / Ubiquitous language
    - multilayered architecture / Multilayered architecture
- business case
    - defining / The business case
- Business Process Execution Language (BPEL)
    - about / Smart services, dumb communication pipes
- Business Process Management (BPM)
    - about / Role of BPM and workflows
    - use cases / Role of BPM and workflows
- business units
    - about / Business units, no components

C

- CalculationController / @PathVariable
- Calico
  - about / What are containers?
- callback / Our first program – Hello World
- calling services
  - New Business / How to tackle organic growth?
  - Renewals / How to tackle organic growth?
- candidate tests, for automation
  - about / Different candidate tests for automation
  - automated sanity tests / Automated sanity tests
  - regression testing / Regression testing
  - automated functional testing / Automated functional testing
  - automated acceptance testing / Automated acceptance testing
  - performance testing / Performance testing
  - real user flow simulation or journey testing / Real user flow simulation or journey testing
  - automated security testing / Automated security testing
  - exploratory testing / Exploratory testing
  - A/B testing / A/B testing, canary testing, and blue-green deployments
  - canary testing / A/B testing, canary testing, and blue-green deployments
  - blue-green deployments / A/B testing, canary testing, and blue-green deployments
  - nonfunctional tests / Other nonfunctional tests
  - Testing in Production (TiP) / Testing in production
  - antifragility testing / Antifragility testing
- CDNs (Content Delivery Networks)
  - about / Microservices to the rescue
- centralized logging solution
  - about / A centralized logging solution
  - log streams / A centralized logging solution
  - log shippers / A centralized logging solution
  - log store / A centralized logging solution
  - log stream processor / A centralized logging solution
  - log dashboard / A centralized logging solution
- Certificate Authority (CA) / Enabling Secure Socket Layer
- cgroups / Docker containers, Diving deep into Mesos
- Chai / Testing microservices in Node.js, Chai
  - BDD-style interfaces / BDD-style interfaces
  - assertions interface / Assertions interface
- Chaos Gorilla
  - about / Simian Army – the active monitoring from Spotify
- Chaos Monkey
  - about / Simian Army – the active monitoring from Spotify

- characteristics, of microservices
  - about / Characteristics of microservices
  - services are first-class citizens / Services are first-class citizens
  - characteristics of services / Characteristics of services in a microservice
  - lightweight / Microservices are lightweight
  - polyglot architecture / Microservices with polyglot architecture
  - automation / Automation in a microservices environment
  - supporting ecosystem / Microservices with a supporting ecosystem
  - distributed and dynamic / Microservices are distributed and dynamic
  - antifragility / Antifragility, fail fast, and self-healing
  - fail fast / Antifragility, fail fast, and self-healing
  - self-healing / Antifragility, fail fast, and self-healing
- characteristics of services, microservices
  - service contract / Characteristics of services in a microservice
  - loose coupling / Characteristics of services in a microservice
  - service abstraction / Characteristics of services in a microservice
  - service reuse / Characteristics of services in a microservice
  - statelessness / Characteristics of services in a microservice
  - services are discoverable / Characteristics of services in a microservice
  - service interoperability / Characteristics of services in a microservice
  - service composeability / Characteristics of services in a microservice
- Chef
  - URL / Installing Node.js, npm, Seneca, and PM2
- circuit breaker
  - about / Circuit breaker and monitoring
  - monitoring / Circuit breaker and monitoring, Monitoring
  - Hystrix's fallback methods, implementing / Using Hystrix's fallback methods
  - Hystrix Dashboard, setting up / Setting up the Hystrix Dashboard
  - Turbine, setting up / Setting up Turbine
- circuit breaker pattern
  - reference / Spring Cloud Hystrix for fault-tolerant microservices
- circuit breakers
  - URL / Circuit breakers
- client-side load balancing
  - about / Client-side load balancing
  - features / Client-side load balancing
  - DiscoveryClient sample / Client-side load balancing
  - RESTTemplate sample / Client-side load balancing
  - FeignClient sample / Client-side load balancing
- client types, OAuth 2.0 client registration
  - about / Client types
  - confidential / Confidential client type
  - public / Public client type

    - web application / Web application
    - user agent-based application / User agent-based application
    - native application / Native application
- cloud
    - as self-service infrastructure, for microservices / Cloud as the self-service infrastructure for Microservices
- cluster management
    - about / Why cluster management is important
    - features / Why cluster management is important
    - functioning / What does cluster management do?
    - relationship with microservices / Relationship with microservices
    - relationship with virtualization / Relationship with virtualization
    - with Mesos and Marathon / Cluster management with Mesos and Marathon
- cluster management solutions
    - about / Cluster management solutions
    - Docker Swarm / Docker Swarm
    - Kubernetes / Kubernetes
    - Apache Mesos / Apache Mesos
    - Nomad / Nomad
    - Fleet / Fleet
- CMD command
    - about / Automating Docker container creation
- code reviews
    - effective ways / Effective code reviews
- Command Query Responsibility Segregation (CQRS) / Single responsibility
- Command Query Responsibility segregations (CQRS) / Microservice use cases
- common name (CN) / Enabling Secure Socket Layer
- communications service
    - about / Then the microservices appeared
- communication styles, designing
    - synchronous style communication / Synchronous style communication
    - asynchronous style communication / Asynchronous style communication
    - style, selecting / How to decide which style to choose?
- components, Spring Cloud
    - Netflix OSS / Spring Cloud and Netflix OSS
- comprehensive microservice example
    - developing / Developing a comprehensive microservice example
- Config server URL
    - defining / Understanding the Config server URL
    - references / Understanding the Config server URL
    - Config Server, accessing from clients / Accessing the Config Server from clients
- configurable items (CIs) / Operation overheads
- configuration management

    - about / Docker – a container for software delivery
- Configuration Management Database (CMDB) / Operation overheads
- container
    - used, for microservice deployment / Microservice deployment using containers
    - setting up / Setting up the container
    - executing / Running the container
    - creation, automating / Automating Docker container creation
- containerization
    - about / The future of containerization – unikernels and hardened security
    - future / The future of containerization – unikernels and hardened security
    - unikernels / The future of containerization – unikernels and hardened security
    - security issues / The future of containerization – unikernels and hardened security
- containers
    - about / What are containers?
    - versus, virtual machines (VMs) / The difference between VMs and containers
    - benefits / The benefits of containers
    - self-contained / The benefits of containers
    - lightweight / The benefits of containers
    - scalable / The benefits of containers
    - portable / The benefits of containers
    - lower license cost / The benefits of containers
    - DevOps / The benefits of containers
    - version controlled / The benefits of containers
    - reusable / The benefits of containers
    - immutable containers / The benefits of containers
- Content Management System (CMS) / An example of a travel agent portal
- context map
    - about / Context map
    - shared kernel / Shared kernel
    - customer-supplier / Customer-supplier
    - conformist / Conformist
    - anticorruption layer / Anticorruption layer
    - separate ways / Separate ways
    - open host service / Open host service
    - distillation / Distillation
- continuous delivery (CD)
    - about / Continuous delivery
- continuous delivery pipeline, automating
    - about / Automating the continuous delivery pipeline
    - development stage / Development
    - continuous integration / Continuous integration
    - automated testing / Automated testing
    - continuous deployment / Continuous deployment

    - monitoring and feedback / Monitoring and feedback
- Continuous Deployment (CD) / Alignment with Agile practices
- continuous integration (CI)
    - about / Pain points, Continuous integration
- Continuous Integration (CI) / Alignment with Agile practices
- Continuous Integration (CI) tools / Automation in a microservices environment
- continuous integration workflow
    - about / Continuous integration
    - build and QA / Continuous integration
    - packaging / Continuous integration
    - integration tests / Continuous integration
- Conveys law
    - reference / Microservice use cases
- Conway's Law
    - about / How to tackle organic growth?
- core capabilities, microservices capability model
    - about / Core capabilities
    - service listeners / Core capabilities
    - storage capability / Core capabilities
    - business capability definition / Core capabilities
    - event sourcing / Core capabilities
    - service endpoints and communication protocols / Core capabilities
    - API gateway / Core capabilities
    - user interfaces / Core capabilities
- Corporate Customer microservice
    - about / Enabling to build organic systems
- correlation ID
    - using, for service calls / Use of correlation ID for service calls
    - used, for tracking request / Let's see how we can tackle this problem
- cost of change curve
    - about / Concepts in software deployment
- counter metric
    - about / Counter
- create, read, update, and delete (CRUD) / Developing a comprehensive microservice example
- cross-origin
    - enabling, for microservices / Enabling cross-origin access for microservices
- Cross-site request forgery (CSRF)
    - about / Cross-site request forgery
- cross-site scripting
    - about / Cross-site scripting
    - output encoding / Output encoding
- CRUD (Create Read Update Delete)
    - about / Technology heterogeneity

- cryptography algorithm / SSH – encrypting the communications
- curl
  - about / Diagnosing problems
- Customer microservices
  - about / Enabling to build organic systems
- Customer Profile microservice
  - about / Services are first-class citizens
  - Register Customer / Services are first-class citizens
  - Get Customer / Services are first-class citizens
- Customer service
  - about / Allowing the coexistence of different versions
- custom health module
  - adding / Adding a custom health module
  - custom metrics, building / Building custom metrics
- custom metrics
  - about / Custom metrics
  - simple metric / Simple metric
  - counter metric / Counter
  - average calculated values metric / Average calculated values
- cyclic dependencies
  - about / Dependencies and versions
  - impact / Cyclic dependencies and their impact
  - analyzing, while designing system / It needs to be analyzed while designing the system
  - different versions, maintaining / Maintaining different versions

D

- data access object (DAO) pattern / Modular organization best practices
- data access objects (DAO) / Monolithic architecture overview
- data analysis
  - data lakes, used / Data analysis using data lakes
- Database Lifecycle Management (DLM) / Addressing challenges around databases
- data lake
  - about / Data analysis using data lakes
- debugging
  - about / Debugging
- decentralization
  - about / Decentralization
- default embedded web server
  - changing / Changing the default embedded web server
- definition function / Writing plugins
- deliver early
  - reference link / First, there was the monolith
  - about / First, there was the monolith
- deliver often
  - reference link / First, there was the monolith
  - about / First, there was the monolith
- dependencies, microservice boundaries
  - events, as opposed to query / Events as opposed to query
  - events, as opposed to synchronous updates / Events as opposed to synchronous updates
  - requirements, challenging / Challenge requirements
  - service boundaries, challenging / Challenge service boundaries
  - dependency graph / Final dependency graph
- development environment
  - setting up / Setting up a development environment
- development philosophy
  - selecting / Choosing a development philosophy
  - design thinking / Design thinking
  - start-up model / The start-up model
  - Agile practice / The Agile practice
- development stage
  - about / Development
  - source code / Development
  - unit test cases / Development
  - consumer-driven contracts / Development
  - mock testing / Development
  - behavior driven design (BDD) / Development
  - source code repository / Development

    - build tool / Development
    - artifact repository / Development
    - database schemas / Development
- DevOps
    - about / What are microservices?
    - URL / What are microservices?
    - defining / The new mantra of lean IT – DevOps
    - wastage, reducing / Reducing wastage
    - tasks, automating / Automating every possible step
    - value-driven delivery / Value-driven delivery
    - development and operations, bridging / Bridging development and operations
    - as practice and process, for microservices / DevOps as the practice and process for microservices
- DevOps-style microservice life cycle process
    - defining / Defining a DevOps-style microservice life cycle process
    - continuous integration (CI) / Defining a DevOps-style microservice life cycle process, Continuous integration
    - continuous delivery (CD) / Defining a DevOps-style microservice life cycle process
    - continuous deployment / Defining a DevOps-style microservice life cycle process
    - Application Release Automation (ARA) / Defining a DevOps-style microservice life cycle process
    - value-driven planning / Value-driven planning
    - Agile development / Agile development
    - continuous testing / Continuous testing
    - continuous release / Continuous release
    - continuous monitoring and feedback / Continuous monitoring and feedback
- directives
    - ng-app / Directives
    - ng-model / Directives
    - ng-bind / Directives
    - ng-submit / Directives
    - ng-repeat / Directives
- Docker
    - used, for µServices deployment / Deployment using a container such as Docker
    - about / Docker, What are containers?, Introduction to Docker, Continuous delivery, Docker – a container for software delivery
    - architecture / Docker's architecture
    - image / Docker image
    - configuration / Installation and configuration
    - installation / Installation and configuration
    - 4 GB memory / Docker Machine with 4 GB
    - images, building with Maven / Building Docker images with Maven
    - running, with Maven / Running Docker using Maven

    - integrating with / Integration testing with Docker
    - images, pushing to registry / Pushing the image to a registry
    - containers, managing / Managing Docker containers
    - reference link / References
    - key components / The key components of Docker
    - concepts / Docker concepts
    - microservices, deploying / Deploying microservices in Docker
    - URL / Deploying microservices in Docker, Installing Docker
    - RabbitMQ, running on / Running RabbitMQ on Docker
    - container, setting up / Setting up the container
    - installing / Installing Docker
    - image, selecting / Choosing the image
    - container, executing / Running the container
    - Node.js, installing / Installing the required software
    - modifications, saving / Saving the changes
    - Node.js applications, deploying / Deploying Node.js applications
    - container creation, automating / Automating Docker container creation
- Docker client
    - about / The Docker client
- Docker concepts
    - Docker images / Docker images
    - Docker containers / Docker containers
    - Docker registry / The Docker registry
    - Dockerfile / Dockerfile
- Docker container
    - about / Docker container
    - deployment / Deployment
- Docker daemon
    - about / The Docker daemon
- Docker file
    - contents / Deploying microservices in Docker
- Docker Hub
    - reference link / Docker's architecture
- Docker Hub account
    - reference / Setting up the Docker Hub
- Dockerization / Introduction to Docker
- Docker Machine
    - about / Deploying Node.js applications
- Docker maven plugins
    - reference links / Building Docker images with Maven
- Docker registry
    - using / Using the Docker registry
    - Docker hub, setting up / Setting up the Docker Hub

    - microservices, publishing to Docker hub / Publishing microservices to the Docker Hub
- Docker Swarm
    - about / Docker Swarm
- Doctor Monkey
    - about / Simian Army – the active monitoring from Spotify
- Document Object Model (DOM) / Javascript
- domain-driven design (DDD)
    - fundamentals / Domain-driven design fundamentals
    - building blocks / Building blocks
    - artifacts / Artifacts of domain-driven design
    - strategic design and principles / Strategic design and principles
    - sample domain service / Sample domain service
    - about / Establishing appropriate microservice boundaries
    - URL / Establishing appropriate microservice boundaries
- domain-specific language (DSL) / Mocha
    - about / Automating Docker container creation
- Drawbridge
    - about / What are containers?
- Drools
    - about / Rules engine – shared or embedded?
- dump
    - about / Spring Boot actuators
- dynamic discovery
    - defining / Understanding dynamic service registration and discovery
- dynamic resource provisioning and deprovisioning / Understanding the concept of autoscaling
- dynamic service registration
    - defining / Understanding dynamic service registration and discovery

E

- e-mailer
  - about / The e-mailer – a common problem
  - e-mails, sending / How to send e-mails
  - interface, defining / Defining the interface
  - Mandrill, setting up / Setting up Mandrill
  - Mandrill, integrating in microservice / Hands on – integrating Mandrill in your microservice
  - fallback strategy / The fallback strategy
- eBay
  - URL / Microservices early adopters
  - about / Microservices early adopters
- ECMAScript 6
  - about / The future of Node.js
- ecosystem capabilities
  - building / Building ecosystem capabilities
- Edda / AWS resource monitoring – Edda
- Edda, features
  - dynamic querying / AWS resource monitoring – Edda
  - history/changes / AWS resource monitoring – Edda
  - configuration / AWS resource monitoring – Edda
- elasticity / Understanding the concept of autoscaling
- Elasticsearch
  - installing / Installing Elasticsearch
  - reference link / Installing Elasticsearch, References
  - URL / A custom logging implementation
- ELK stack
  - and logging / Logging and ELK stack
  - overview / A brief overview
  - Elasticsearch / Elasticsearch
  - Logstash / Logstash
  - Kibana / Kibana
  - setting up / ELK stack setup
  - implementation, tips / Tips for ELK stack implementation
- ELK stack setup
  - about / ELK stack setup
  - Elasticsearch, installing / Installing Elasticsearch
  - Logstash, installing / Installing Logstash
  - Kibana, installing / Installing Kibana
- emailer
  - about / Micromerce – the big picture
- end-to-end testing

    - about / End-to-end testing
- end-to-end tests / End-to-end tests
- engine
    - about / Fleet
- Enterprise Application integration (EAI)
    - about / Integration with other systems
- enterprise archive (EAR)/web archive (WAR) / Monolithic architecture overview
- Enterprise Integration Patterns (EIP) / Service-oriented integration
- Enterprise Service Bus (ESB) / Evolution of µServices, Microservices are distributed and dynamic
    - about / SOA versus microservices
- ENV command
    - about / Automating Docker container creation
- Eureka / Service registration and discovery – Eureka
    - defining / Understanding Eureka
    - high availability / High availability for Eureka
- Eureka server
    - setting up / Setting up the Eureka server
- European Computer Manufacturers Association (ECMA)
    - about / The future of Node.js
- event loop, Node.js
    - about / Node.js event loop – easy to learn and hard to master
- evolution
    - planning / Plan the evolution
- evolution, microservices
    - about / The evolution of microservices
    - business demand, as catalyst / Business demand as a catalyst for microservices evolution
    - technology, as catalyst / Technology as a catalyst for the microservices evolution
    - imperative architecture evolution / Imperative architecture evolution
- examples, microservices
    - about / Microservices examples
    - holiday portal / An example of a holiday portal
    - microservice-based order management system / A microservice-based order management system
    - travel agent portal / An example of a travel agent portal
- exceptions / Single-threaded applications and exceptions
- EXPOSE command
    - about / Automating Docker container creation
- Express / Need for Node.js, Web server integration
    - integrating with / Integrating with Express – how to create a REST API

F

- factors
  - defining / Prioritizing microservices for migration
- FakeSMTP
  - URL / Developing a comprehensive microservice example
- Feign
  - defining / Feign as a declarative REST client
- FeignClient / Client-side load balancing
- Fenzo
  - features / Scheduler for Apache Mesos – Fenzo
- Fleet
  - about / Fleet
- Fly By Points / An example of a holiday portal
- followers
  - about / Nomad
- forever / Need for Node.js, Single-threaded applications and exceptions
- Fully Integrated Defense Operation (FIDO) / Other security tools – Scumblr and FIDO, Fully Integrated Defence Operation (FIDO)
- functional testing
  - about / Functional testing
  - automated testing / The pyramid of automated testing

G

- gateway / Docker containers
- Gilt
  - URL / Microservices early adopters
  - about / Microservices early adopters
- Glue Conference (Glue Con) / Self-monitoring and logging
- Gradle
  - download link / Summary
- Guest OS
  - about / The difference between VMs and containers

H

- HAL (Hypertext Application Language) browser / Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example
- hardware resource metrics
  - memory metrics / Monitoring services
  - CPU utilization / Monitoring services
  - disk utilization / Monitoring services
- HATEOAS example / Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example
- health
  - about / Spring Boot actuators
- health check
  - about / Health check on NGINX
  - passive health check / Passive health check
  - active health check / Active health check
- Hello World program / Our first program – Hello World
- hexagonal architecture, microservices
  - reference / What are microservices?
- high availability
  - URL / Setting up high availability for the Config server
- high availability, of Zuul
  - about / High availability of Zuul
  - Eureka client / High availability of Zuul when the client is also a Eureka client
  - not a Eureka client / High availability when the client is not a Eureka client
- highly cohesive business unit
  - about / Business units, no components
- home page/restaurant list page
  - header section / Home page/restaurant list page
  - content or middle section / Home page/restaurant list page
  - footer section / Home page/restaurant list page
  - designing, file requisites / Home page/restaurant list page
  - index.html file / Home page/restaurant list page, index.html
  - app.js file / Home page/restaurant list page, app.js
  - restaurants.js file / Home page/restaurant list page, restaurants.js
  - restaurants.html file / Home page/restaurant list page, restaurants.html
- honeycomb analogy
  - about / Microservices – the honeycomb analogy
- HTTP codes
  - about / HTTP codes
  - 1xx - informational / 1xx – informational
  - 2xx - success codes / 2xx – success codes
  - 3xx - redirection / 3xx – redirection

    - 4xx - client errors / 4xx – client errors
    - 5xx - server errors / 5xx – server errors
    - in microservices, importance / Why HTTP codes matter in microservices
- Hyper Text Transfer Protocol (HTTP) / Enabling Secure Socket Layer
    - about / Smart services, dumb communication pipes
- Hyper Text Transfer Protocol over SSL (HTTPS) / Enabling Secure Socket Layer
- Hyper Text Transfer Protocol Secure (HTTPS) / Enabling Secure Socket Layer
- Hystrix
    - reference link / References
  / Circuit breaker – Hystrix
- Hystrix's fallback methods
    - circuit breaker, enabling / Using Hystrix's fallback methods
    - configuring / Using Hystrix's fallback methods
    - defining / Using Hystrix's fallback methods

I

- Ice
  - about / Cost and cloud utilization – Ice
- implicit grant, OAuth 2.0 grant types
  - about / Implicit grant
  - requests and responses / Implicit grant requests and responses, Resource owner password credentials grant
- in-memory data grid (IMDG) / Elastically and selectively scalable
- independence
  - about / Independence
- info
  - about / Spring Boot actuators
- infrastructure
  - autoscaling / Autoscaling the infrastructure
- infrastructure as code / Cloud as the self-service infrastructure for Microservices
- infrastructure capabilities, microservices capability model
  - about / Infrastructure capabilities
  - cloud / Infrastructure capabilities
  - containers or virtual machines / Infrastructure capabilities
  - cluster control and provisioning / Infrastructure capabilities
  - application life cycle management / Infrastructure capabilities
- infrastructure logical security
  - about / Infrastructure logical security
- injection
  - about / Injection
  - input validation / Input validation
- INPUT queue / Supporting the building of self-organizing systems
- input validation
  - about / Input validation
- instance ID / Preparing BrownField PSS services
- Integration Platform as a Service (iPaaS) / Technology as a catalyst for the microservices evolution
- Integration Testing / Continuous integration and deployment
- integration tests / Integration tests
- Intelligent Business Process Management (iBPM) / Role of BPM and workflows
- Interface Segregation Principle (ISP) / Sample domain service
- Internet Engineering Task Force (IETF) / OAuth 2.0
- IP hashing
  - about / Load balancing our application

J

- Java Remote Method Invocation (RMI)
  - about / Technology alignment
- JavaScript / Javascript
- JavaScript Object Notation (JSON)
  - about / Product Manager – the two-faced core
- JDK 1.8
  - reference / Setting up a development environment
- JDK 8
  - download link / NetBeans IDE installation and setup
- Jenkins
  - reference link / Summary
- jobs
  - about / Nomad
- JSch (Java Secure Channel) / A walkthrough of the life cycle manager code
- JS modules
  - bower / Setting up the web app
  - gulp / Setting up the web app
  - node / Setting up the web app
- JSON / Documenting APIs with Swagger
- JSON structure
  - id / Deploying BrownField PSS services
  - cpus / Deploying BrownField PSS services
  - mem / Deploying BrownField PSS services
  - instances / Deploying BrownField PSS services
  - container / Deploying BrownField PSS services
  - image / Deploying BrownField PSS services
  - network / Deploying BrownField PSS services
  - portMappings / Deploying BrownField PSS services

K

- kernel namespace / Docker containers
- key benefits
  - resilience / Resilience
  - scalability / Scalability
  - technology heterogeneity / Technology heterogeneity
  - replaceability / Replaceability
  - independence / Independence
  - easy to deploy / Easy to deploy
- key capabilities, cluster management software
  - about / What does cluster management do?
  - cluster management / What does cluster management do?
  - deployments / What does cluster management do?
  - scalability / What does cluster management do?
  - health / What does cluster management do?
  - infrastructure abstraction / What does cluster management do?
  - resource optimization / What does cluster management do?
  - resource allocation / What does cluster management do?
  - service availability / What does cluster management do?
  - agility / What does cluster management do?
  - isolation / What does cluster management do?
- key components, automated microservices deployment topology
  - about / Scaling microservices with Spring Cloud
  - Eureka / Scaling microservices with Spring Cloud
  - Eureka client / Scaling microservices with Spring Cloud
  - Ribbon client / Scaling microservices with Spring Cloud
- key components, Docker
  - about / The key components of Docker
  - Docker daemon / The key components of Docker
  - Docker client / The key components of Docker
- key design principles
  - about / Key design principles
  - business units / Business units, no components
  - smart services / Smart services, dumb communication pipes
  - decentralization / Decentralization
  - technology alignment / Technology alignment
  - small size / How small is too small?
- Keymetrics
  - used, for monitoring services / Monitoring using PM2 and Keymetrics
  - about / Monitoring using PM2 and Keymetrics
  - URL / Monitoring using PM2 and Keymetrics
  - used, for diagnosing problems / Diagnosing problems

    - used, for monitoring application exceptions / Monitoring application exceptions
    - custom metrics / Custom metrics
- Kibana
    - download link / Installing Kibana
    - installing / Installing Kibana
    - reference link / Installing Kibana, References
- Kubernetes
    - about / Kubernetes
- Kurma
    - about / What are containers?

L

- Lambda architecture
  - reference / A centralized logging solution
- latency
  - about / Throughput and performance degradation
- Latency Monkey
  - about / Simian Army – the active monitoring from Spotify
- leader
  - about / Nomad
- least connected node
  - about / Load balancing our application
- libraries
  - about / Microservices development governance, reference architectures, and libraries
- libuv library
  - about / Node.js event loop – easy to learn and hard to master
- life cycle manager / Autoscaling BrownField PSS microservices
  - about / The capabilities required for an autoscaling system, A place for the life cycle manager
  - metrics collection / The capabilities required for an autoscaling system
  - scaling policies / The capabilities required for an autoscaling system
  - decision engine / The capabilities required for an autoscaling system
  - deployment rules / The capabilities required for an autoscaling system
  - deployment engine / The capabilities required for an autoscaling system
  - updating / Updating the life cycle manager
  - rewriting, with Mesos and Marathon / Rewriting the life cycle manager with Mesos and Marathon
- line of code (LOC) / Nanoservice (not recommended), size, and monolithic
- Lmctfy
  - about / What are containers?
- load balancing
  - about / Load balancing
  - client-side load balancing / Client-side load balancing
  - client-side load balancing, features / Client-side load balancing
  - server-side load balancing / Server-side load balancing
  - of Node.js applications / Load balancing our application
- logging
  - and ELK stack / Logging and ELK stack
  - about / Logging
- logging, levels / Auditing
  - DEBUG / Auditing
  - INFO / Auditing
  - WAR / Auditing

    - FATAL / Auditing
- logging solutions
    - selecting / The selection of logging solutions
    - cloud services / Cloud services
    - off-the-shelf solutions / Off-the-shelf solutions
    - best-of-the-breed integration / Best-of-breed integration
    - custom logging implementation / A custom logging implementation
    - distributed tracing, with Spring Cloud Sleuth / Distributed tracing with Spring Cloud Sleuth
- Loggly
    - reference link / Summary
- login page
    - about / Login page
    - login.html template / login.html
    - login.js / login.js
- log management challenges
    - about / Understanding log management challenges
- Logstash
    - download link / Installing Logstash
    - installing / Installing Logstash
    - documentation, reference link / Installing Logstash
    - reference link / References
- loosely coupled business unit
    - about / Business units, no components
- Loyalty Points microservice
    - about / Enabling to build organic systems
- LXD
    - about / What are containers?

M

- 10-18 Monkey
  - about / Simian Army – the active monitoring from Spotify
- manager
  - about / Docker Swarm
- Mandrill
  - setting up / Setting up Mandrill
  - URL / Setting up Mandrill, Hands on – integrating Mandrill in your microservice
  - integrating, in microservice / Hands on – integrating Mandrill in your microservice
- manual testing
  - about / Manual testing – the necessary evil
  - proxy, building to debug microservices / Building a proxy to debug our microservices
  - Postman / Postman
- mappings
  - about / Spring Boot actuators
- Marathon
  - about / Marathon
  - features / Marathon
  - implementing, for BrownField microservices / Implementing Mesos and Marathon for BrownField microservices
  - reference / Installing ZooKeeper, Mesos, and Marathon
  - installing / Installing ZooKeeper, Mesos, and Marathon
  - running as services / Running Mesos, Marathon, and ZooKeeper as services
- Marathon autoscale
  - about / A place for the life cycle manager
- master
  - about / Kubernetes
- Master Data Management (MDM) / A separate data store for each microservice
- Maven
  - download link / Summary
  - used, for building Docker images / Building Docker images with Maven
  - used, for running Docker / Running Docker using Maven
- Maven 3.3.1
  - reference / Setting up a development environment
- Mean Time Between Failures (MTBF) / Antifragility, fail fast, and self-healing
- Mean Time To Recover (MTTR) / Antifragility, fail fast, and self-healing
- memory leak
  - about / Diagnosing problems
- Mesos
  - about / Diving deep into Mesos
  - architecture / The Mesos architecture
  - workflow diagram / The Mesos architecture

    - implementing, for BrownField microservices / Implementing Mesos and Marathon for BrownField microservices
    - reference / Installing ZooKeeper, Mesos, and Marathon
    - installing / Installing ZooKeeper, Mesos, and Marathon
    - configuring / Configuring Mesos
    - running as services / Running Mesos, Marathon, and ZooKeeper as services
- Mesos slave
    - running, in command line / Running the Mesos slave in the command line
- metrics
    - about / Spring Boot actuators
- micromerce
    - about / Micromerce – the big picture
    - deploying, PM2 used / Deploying microservices with PM2
- microservice
    - testing / Testing a real microservice
- Microservice A / Establishing self-organizing teams
- microservice architecture
    - demonstrating, Netflix Open Source Software Center (OSS) used / An overview of microservice architecture using Netflix OSS
    - Edge Server / An overview of microservice architecture using Netflix OSS
    - load balancing / An overview of microservice architecture using Netflix OSS
    - circuit breaker / An overview of microservice architecture using Netflix OSS
    - service discovery and registration / An overview of microservice architecture using Netflix OSS
    - monitoring dashboard / An overview of microservice architecture using Netflix OSS
- Microservice B / Establishing self-organizing teams
- microservice boundaries
    - autonomous functions / Autonomous functions
    - size, of deployable unit / Size of a deployable unit
    - subdomain / Most appropriate function or subdomain
    - most appropriate function / Most appropriate function or subdomain
    - polyglot architecture / Polyglot architecture
    - selective scaling / Selective scaling
    - agile teams / Small, agile teams
    - single responsibility / Single responsibility
    - replicability / Replicability or changeability
    - changeability / Replicability or changeability
    - coupling / Coupling and cohesion
    - microservice, as product / Think microservice as a product
    - identifying / Identification of microservices boundaries
    - dependencies, analyzing / Analyze dependencies
- microservice capability model
    - reviewing / Reviewing the microservice capability model, Reviewing the microservice

capability model, Reviewing the microservice capability model, Reviewing the microservice capability model, Reviewing the microservice capability model
- microservice deployment
  - container, using / Microservice deployment using containers
- microservice monitoring
  - about / Monitoring microservices
  - objectives / Monitoring microservices
  - challenges / Monitoring challenges
  - metrics sources and data collectors / Monitoring challenges
  - aggregation and correlation of metrics / Monitoring challenges
  - processing metrics and actionable insights / Monitoring challenges
  - alerts / Monitoring challenges
  - actions / Monitoring challenges
  - dashboards / Monitoring challenges
  - Application Performance Monitoring (APM) approach / Monitoring challenges
  - synthetic monitoring / Monitoring challenges
  - real user monitoring (RUM) / Monitoring challenges
  - user experience monitoring / Monitoring challenges
  - tools / Monitoring tools
  - microservice dependencies, monitoring / Monitoring microservice dependencies
  - Spring Cloud Hystrix, for fault-tolerant microservices / Spring Cloud Hystrix for fault-tolerant microservices
  - Hystrix streams, aggregating with Turbine / Aggregating Hystrix streams with Turbine
- microservices
  - evolution / The evolution of microservices, Then the microservices appeared
  - about / What are microservices?
  - hexagonal architecture / What are microservices?
  - architecture / What are microservices?
  - honeycomb analogy / Microservices – the honeycomb analogy
  - principles / Principles of microservices
  - Order / Single responsibility per service
  - Customer / Single responsibility per service
  - Product / Single responsibility per service
  - characteristics / Characteristics of microservices
  - examples / Microservices examples
  - benefits / Microservices benefits, The business case
  - relationship, with other architecture / Relationship with other architecture styles
  - use cases / Microservice use cases
  - documenting / Documenting microservices, Documenting microservices
  - defining / Microservices to the rescue
  - prioritizing, for migration / Prioritizing microservices for migration
  - data synchronization, during migration / Data synchronization during migration
  - reference data, managing / Managing reference data

    - scaling, with Spring Cloud / Scaling microservices with Spring Cloud
    - and containers / Microservices and containers
    - deploying, in Docker / Deploying microservices in Docker
    - need for / Need for microservices
    - in real world / Microservices in the real world
    - versus Service-Oriented Architectures (SOA) / SOA versus microservices
    - disadvantages / Disadvantages
    - monolith, splitting / Splitting the monolith
    - testing, in Node.js / Testing microservices in Node.js
    - APIs, documenting with Swagger / Documenting APIs with Swagger
    - project, generating from Swagger definition / Generating a project from the Swagger definition
- microservices, on cloud
    - about / Microservices on the cloud
    - Docker, installing on AWS EC2 / Installing Docker on AWS EC2
- microservices-based architecture
    - examining / What are microservices?
- microservices-based design
    - overview / Overview and mindset
    - mindet / Overview and mindset
- microservices-oriented architectures
    - about / Microservice-oriented architectures
    - benefits / How is it better?
    - shortcomings / Shortcomings
- microservices capabilities
    - reviewing / Reviewing microservices capabilities
- microservices capability model
    - about / The microservices capability model
    - core capabilities / The microservices capability model, Core capabilities
    - supporting capabilities / The microservices capability model, Supporting capabilities
    - infrastructure capabilities / The microservices capability model, Infrastructure capabilities
    - governance capabilities / The microservices capability model
    - process and governance capabilities / Process and governance capabilities
    - reviewing / Reviewing the microservices capability model
- microservices challenges
    - about / Microservices challenges
    - data islands / Data islands
    - logging / Logging and monitoring
    - monitoring / Logging and monitoring
    - dependency management / Dependency management
    - organization culture / Organization culture
    - governance challenges / Governance challenges
    - operation overheads / Operation overheads

    - microservices, testing / Testing microservices
    - infrastructure provisioning / Infrastructure provisioning
- microservices development
    - practice points / Practice points for microservices development
- microservices development teams
    - about / Microservices development governance, reference architectures, and libraries
- Microservices frameworks and tools
    - about / Microservices frameworks and tools
    - Netflix Open Source Software (OSS) / Netflix Open Source Software (OSS)
- Minimum Viable Product (MVP)
    - using / Using the concept of Minimum Viable Product
    - about / Using the concept of Minimum Viable Product
- Minimum Viable Products (MVPs) / Replicability or changeability
- Mocha / Testing microservices in Node.js
    - about / Mocha
- Model-View-ViewModel (MVVM)
    - about / MVVM
    - Model / MVVM
    - View / MVVM
    - ViewModel / MVVM
- Model View Controller (MVC)
    - about / MVC
    - Model / MVC
    - View / MVC
    - Controller / MVC
- modules
    - migrating / Migrate modules only if required
- modules, advantages
    - modularity / Modules
    - simplicity / Modules
    - testing / Modules
- mongo-store
    - URL / Wiring everything up
- MongoDB
    - URL / Product Manager – the two-faced core
- Monkeys
    - Chaos Monkey / Reliability monitoring service – Simian Army
    - Janitor Monkey / Reliability monitoring service – Simian Army
    - Conformity Monkey / Reliability monitoring service – Simian Army
    - Security Monkey / Reliability monitoring service – Simian Army
- monolith
    - about / First, there was the monolith
    - organic growth, tackling / How to tackle organic growth?

    - abstraction capacity / How abstract is too abstract?
    - splitting / Splitting the monolith
    - problems, in splitting / Problems splitting the monolith – it is all about the data
- Monolithic (Etsy)
    - versus Microservices (Netflix) Twitter, reference link / References
- monolithic architecture
    - overview / Monolithic architecture overview
    - versus solution, limitations with µServices / Limitation of monolithic architecture versus its solution with µServices
- monolithic architecture, limitations
    - one dimension scalability / One dimension scalability
    - release rollback / Release rollback in case of failure
    - new technology adoption, issues / Problems in adopting new technologies
    - alignment with Agile processes / Alignment with Agile practices
    - ease of development / Ease of development – could be done better
    - microservices build pipeline / Microservices build pipeline
    - deployment, with Docker / Deployment using a container such as Docker
    - container / Containers
    - Docker / Docker
- monolithic software
    - about / Monolithic software
- monolithic software architecture
    - about / Monolithic software
- multilayered architecture, domain-driven design (DDD)
    - about / Multilayered architecture
    - presentation layer / Presentation layer
    - application layer / Application layer
    - domain layer / Domain layer
    - infrastructure layer / Infrastructure layer

N

- Nanoservice Antipattern
  - reference link / References
- Nebula / Build – Nebula
- NetBeans
  - download link / References
- NetBeans IDE
  - installation / NetBeans IDE installation and setup
  - download link / NetBeans IDE installation and setup
  - setup / NetBeans IDE installation and setup
  - reference link / NetBeans IDE installation and setup
- Netflix
  - URL / Microservices early adopters
  - about / Microservices early adopters, Spring Cloud and Netflix OSS, Microservices in the real world
- Netflix Eureka
  - reference link / References
- Netflix Open Source Software (Netflix OSS)
  - about / Spring Cloud and Netflix OSS
- Netflix Open Source Software (OSS)
  - about / Netflix Open Source Software (OSS)
  - Build - Nebula / Build – Nebula
  - Nebula / Build – Nebula
  - Spinnaker / Deployment and delivery – Spinnaker with Aminator
  - Aminator / Deployment and delivery – Spinnaker with Aminator
  - Eureka / Service registration and discovery – Eureka
  - Ribbon / Service communication – Ribbon
  - Hystrix / Circuit breaker – Hystrix
  - Zuul / Edge (proxy) server – Zuul
  - Atlas / Operational monitoring – Atlas
  - Simian Army / Reliability monitoring service – Simian Army
  - Edda / AWS resource monitoring – Edda
  - Vector / On-host performance monitoring – Vector
  - Archaius / Distributed configuration management – Archaius
  - Fenzo / Scheduler for Apache Mesos – Fenzo
  - Ice / Cost and cloud utilization – Ice
  - Fully Integrated Defence Operation (FIDO) / Other security tools – Scumblr and FIDO
  - Scumblr / Other security tools – Scumblr and FIDO
- Netflix Open Source Software Center (OSS) / An overview of microservice architecture using Netflix OSS
- Netflix Ribbon
  - reference link / References

- Netflix Zuul
  - reference link / References
- network latency
  - about / Shortcomings
- NGINX
  - about / Load balancing our application
  - using / Load balancing our application
  - URL / Load balancing our application
  - health check / Health check on NGINX
- Nike
  - URL / Microservices early adopters
  - about / Microservices early adopters
- Node.js
  - need for / Why Node.js?, Need for Node.js
  - API aggregation / API aggregation
  - future / The future of Node.js
  - installing / Installing Node.js, npm, Seneca, and PM2, Installing the required software
  - URL / Installing Node.js, npm, Seneca, and PM2
  - threading model / Node.js threading model
  - modular organization best practices / Modular organization best practices
  - JavaScript / Javascript
  - SOLID / SOLID design principles
  - microservices, testing / Testing microservices in Node.js
  - Chai / Chai
  - BDD-style interfaces / BDD-style interfaces
  - assertions interface / Assertions interface
  - Mocha / Mocha
  - Sinon / Sinon.JS – a mocking framework
  - real microservice, testing / Testing a real microservice
  - Test-driven development (TDD) / TDD – Test-driven development
  - event loop / Node.js event loop – easy to learn and hard to master
- Node.js applications
  - deploying / Deploying Node.js applications
  - clustering / Clustering Node.js applications
  - load balancing / Load balancing our application
- Node.js package manager (npm)
  - about / Setting up the web app
  - reference link / Setting up the web app
- nodemon / Need for Node.js, Single-threaded applications and exceptions
- Node Package Manager (npm) / Installing Node.js, npm, Seneca, and PM2
- nodes
  - about / Docker Swarm, Kubernetes
- Nomad

    - about / Nomad
- non-persistent XSS attacks / Cross-site scripting
- npm
    - about / Why Node.js?
    - installing / Installing Node.js, npm, Seneca, and PM2
    - learning / Learning npm
- NServiceBus
    - URL / Seneca – a simple puzzle that makes our lives easier

O

- OAuth
  - reference / Securing a microservice with OAuth2
- OAuth 2.0
  - about / OAuth 2.0
  - usage / Usage of OAuth
  - specification / OAuth 2.0 specification – concise details
  - roles / OAuth 2.0 roles
  - client registration / OAuth 2.0 client registration
  - protocol endpoints / OAuth 2.0 protocol endpoints
  - grant types / OAuth 2.0 grant types
- OAuth 2.0 Authorization Framework
  - reference link / References
- OAuth 2.0 client registration
  - about / OAuth 2.0 client registration
  - client types / Client types
  - client identifier / Client identifier
  - client authentication / Client authentication
- OAuth 2.0 grant types
  - about / OAuth 2.0 grant types
  - authorization code grant / Authorization code grant
  - implicit grant / Implicit grant
  - resource owner password credentials grant / Resource owner password credentials grant
  - client credentials grant / Client credentials grant
- OAuth 2.0 protocol endpoints
  - about / OAuth 2.0 protocol endpoints
  - authorization endpoint / Authorization endpoint
  - token endpoint / Token endpoint
  - redirection endpoint / Redirection endpoint
- OAuth 2.0 roles
  - about / OAuth 2.0 roles
  - resource owner / Resource owner
  - resource server / Resource server
  - client / Client
  - authorization server / Authorization server
- OAuth implementation
  - Spring Security, using / OAuth implementation using Spring Security
  - authorization code grant / Authorization code grant
  - implicit grant / Implicit grant
  - resource owner password credential grant / Resource owner password credential grant
  - client credentials grant / Client credentials grant
- object-oriented programming (OOP) / Value objects

    - restaurant service / OTRS overview
    - user service / OTRS overview
    - booking service / OTRS overview
- Object-relational mapping (ORM) / Data storage
- open redirects
    - about / Open redirects
- Open Source Software (OSS) / Microservices frameworks and tools
- Open Web Application Security (OWASP) / Common threats – how to be up to date
- Orbitz
    - about / Microservices early adopters
    - URL / Microservices early adopters
- Order Event / Supporting event-driven architecture
- order manager
    - about / Micromerce – the big picture, The order manager
    - microservice, defining / Defining the microservice – how to gather non-local data
    - code / The order manager – the code
    - remote services, calling / Calling remote services
    - resilience, over perfection / Resilience over perfection
- organic growth
    - about / Need for microservices, First, there was the monolith
    - tackling / How to tackle organic growth?
- organizational alignment
    - about / Organizational alignment
- OTRS
    - overview / OTRS overview
    - features, developing / Development of OTRS features
- OTRS features development
    - about / Development of OTRS features
    - home page/restaurant list page / Home page/restaurant list page
    - Search Restaurants functionality / Search Restaurants
    - restaurant details with reservation option / Restaurant details with reservation option
    - login page / Login page
    - reservation confirmation / Reservation confirmation
- OUTPUT queue / Supporting the building of self-organizing systems
- oversubscription / What does cluster management do?

P

- passive health check
  - about / Passive health check
- Patrun
  - about / Inversion of control done right, Patrun – a pattern-matching library
  - Pclosest match / Patrun – a pattern-matching library
  - patterns, reusing / Reusing patterns
- patterns and common design decisions
  - about / Patterns and common design decisions
  - appropriate microservice boundaries, establishing / Establishing appropriate microservice boundaries
  - communication styles, designing / Designing communication styles
  - orchestration / Orchestration of microservices
  - number of endpoints / How many endpoints in a microservice?
  - microservice, on VM / One microservice per VM or multiple?
  - microservice, on multiple VMs / One microservice per VM or multiple?
  - rules engine / Rules engine – shared or embedded?
  - role, of BPM and workflows / Role of BPM and workflows
  - data store, sharing / Can microservices share data stores?
  - transaction boundaries, setting up / Setting up transaction boundaries
  - service endpoint design consideration / Service endpoint design consideration
  - shared libraries, handling / Handling shared libraries
  - user interfaces, in microservices / User interfaces in microservices
  - API gateways, using in microservices / Use of API gateways in microservices
  - ESB and iPaaS, using with microservices / Use of ESB and iPaaS with microservices
  - service versioning considerations / Service versioning considerations
  - design, for cross origin / Design for cross origin
  - shared reference data, handling / Handling shared reference data
  - bulk operations, in microservices / Microservices and bulk operations
- Performance Co-Pilot (PCP) / On-host performance monitoring – Vector
- persistent XSS attacks / Cross-site scripting
- Plain Old Java Object (POJO) / Sample REST program
- Plain Old Java Objects (POJO) services / Service-oriented application
- Platform as a Services (PaaS) / Technology as a catalyst for the microservices evolution
- PM2
  - installing / Installing Node.js, npm, Seneca, and PM2
  - about / PM2 – a task runner for Node.js, Monitoring using PM2 and Keymetrics
  - single-threaded applications and exceptions / Single-threaded applications and exceptions
  - using / Using PM2 – the industry-standard task runner
  - used, for monitoring services / Monitoring using PM2 and Keymetrics
  - used, for software deployment / Deployments with PM2
  - ecosystems / PM2 – ecosystems

    - used, for deploying micromerce / Deploying microservices with PM2
    - server, configuring / Configuring the server
- pmx
    - about / Monitoring application exceptions
- pods
    - about / Kubernetes
- POJO (Plain Old Java Object)
    - about / Components of Spring Cloud
- polymorphism of actions / Hands on – integrating Mandrill in your microservice
- pom file
    - reference / Examining the POM file
- Postman / Postman
- Postman Chrome extension
    - used, for REST API testing / REST API testing using the Postman Chrome extension
    - positive test scenarios / Some more positive test scenarios
    - negative test scenarios / Negative test scenarios
- practice points, microservices development
    - about / Practice points for microservices development
    - business motivation and value / Understanding business motivation and value
    - mindset, changing from project to product development / Changing the mindset from project to product development
    - development philosophy, selecting / Choosing a development philosophy
    - Minimum Viable Product (MVP), using / Using the concept of Minimum Viable Product
    - legacy hotspot / Overcoming the legacy hotspot
    - challenges around databases, addressing / Addressing challenges around databases
    - self-organizing teams, establishing / Establishing self-organizing teams
    - self-service cloud, building / Building a self-service cloud
    - microservices ecosystem, building / Building a microservices ecosystem
    - DevOps-style microservice life cycle process, defining / Defining a DevOps-style microservice life cycle process
    - continuous delivery pipeline, automating / Automating the continuous delivery pipeline
    - automated configuration management / Automated configuration management
- prerequisites, for bootstrapping app
    - app module / Modules
    - Angular library , loading / Modules
    - App DOM configuration / Modules
- PRG pattern
    - URL / Need for frontend microservice
- principles, of microservices
    - about / Principles of microservices
    - single responsibility principle / Single responsibility per service
    - autonomous services / Microservices are autonomous
- probes

    - about / Custom metrics
- process and governance capabilities, microservices capability model
    - about / Process and governance capabilities
    - DevOps / Process and governance capabilities
    - DevOps tools / Process and governance capabilities
    - microservices repositor / Process and governance capabilities
    - microservice documentation / Process and governance capabilities
    - reference architecture and libraries / Process and governance capabilities
- Product Manager
    - about / Micromerce – the big picture, Product Manager – the two-faced core
    - products, fetching / Fetching products
    - fetching, by category / Fetching by category
    - fetching, by ID / Fetching by ID
    - product, adding / Adding a product
    - product, removing / Removing a product
    - product, editing / Editing a product
    - wiring / Wiring everything up
    - REST API, creating / Integrating with Express – how to create a REST API
    - Express, integrating with / Integrating with Express – how to create a REST API
- protocol selection
    - about / Protocol selection
    - message-oriented services / Message-oriented services
    - HTTP and REST endpoints / HTTP and REST endpoints
    - optimized communication protocols / Optimized communication protocols
    - API documentations / API documentations
- PSS application
    - defining / Understanding the PSS application
    - business process view / Business process view
    - functional view / Functional view
    - architectural view / Architectural view
    - design view / Design view
    - implementation view / Implementation view
    - deployment view / Deployment view
    - about / Death of the monolith
    - limitation / Pain points
    - stop gap fix / Stop gap fix
    - shared data / Shared data
    - single database / Single database
    - domain boundaries / Domain boundaries
- PSS Implementation, BrownField
    - reviewing / Reviewing BrownField's PSS implementation
- pyramid of testing / How much testing is too much?

Q

- queueing
  - about / Throughput and performance degradation
- Quora
  - reference link / Authentication and authorization

R

- RabbitMQ
  - reference link / References
  - running, on Docker / Running RabbitMQ on Docker
  - URL / Seneca – a simple puzzle that makes our lives easier
  - about / Throughput and performance degradation
- Rabbit MQ
  - reference / Implementing Spring Boot messaging
  - URL / Setting up high availability for the Config server
- RabbitMQ transport plugin
  - URL / Throughput and performance degradation
- Random algorithm / What does cluster management do?
- reactive programming
  - reference / A microservice-based order management system
- Real Application Cluster (RAC)
  - about / Stop gap fix
- Real User Monitoring (RUM) tools / Testing in production
- recipe types
  - constant / Providers and services
  - provider / Providers and services
  - value / Providers and services
  - factory / Providers and services
  - service / Providers and services
- reference architectures
  - about / Microservices development governance, reference architectures, and libraries
- references / References, References, References
- Registry
  - about / Choosing the image
  - URL / Choosing the image
- relationship, microservices
  - about / Relationship with other architecture styles
  - relations, with SOA / Relations with SOA
  - relations, with Twelve-Factor Apps / Relations with Twelve-Factor apps
- relationship, microservices with SOA
  - about / Relations with SOA
  - service-oriented integration / Service-oriented integration
  - legacy modernization / Legacy modernization
  - service-oriented application / Service-oriented application
  - monolithic application / Monolithic migration using SOA
- relationship, microservices with Twelve-Factor Apps
  - about / Relations with Twelve-Factor apps
  - single codebase / A single code base

    - dependencies, bundling / Bundling dependencies
    - configurations, externalizing / Externalizing configurations
    - backing services / Backing services are addressable
    - isolation, between build, release, run stages / Isolation between build, release, and run
    - stateless and share nothing processes / Stateless, shared nothing processes
    - services, exposing through port bindings / Exposing services through port bindings
    - concurrency, scaling out / Concurrency to scale out
    - disposability, with minimal overhead / Disposability with minimal overhead
    - development and production parity / Development and production parity
    - logs, externalizing / Externalizing logs
    - package admin processes / Package admin processes
- replaceability
    - about / Replaceability
    - importance / Why is replaceability important?
- Representational State Transfer (REST) / Writing the REST controller class
    - download link / Summary
- resilience
    - about / Resilience
- restaurant details with reservation option
    - about / Restaurant details with reservation option
    - restaurant.html file / restaurant.html
- Restaurant µService
    - about / Restaurant µService
    - Controller class / Controller class
    - Controller class, API versioning / API versioning
    - service classes / Service classes
    - repository classes / Repository classes
    - entity classes / Entity classes
- REST Client extension for Chrome
    - reference link / REST API testing using the Postman Chrome extension
- REST controller class
    - writing / Writing the REST controller class
    - @RestController / @RestController
    - @RequestMapping / @RequestMapping
    - @RequestParam / @RequestParam
    - @PathVariable / @PathVariable
- RESTful microservices
    - building, Spring Boot used / Using Spring Boot to build RESTful microservices
- RESTful service
    - developing / Developing a RESTful service – the legacy approach
- restify / Need for Node.js
- RestTemplate / Client-side load balancing
- Ribbon / Service communication – Ribbon

    - defining, for load balancing / Ribbon for load balancing
    - about / Ribbon for load balancing
- Rocket
    - about / What are containers?
- Round Robin
    - about / Load balancing our application
- RPM (Requests Per Minute) / Understanding the execution flow
- RSA / SSH – encrypting the communications
- run command, parameters
    - -t flag / Deploying Node.js applications
    - -i flag / Deploying Node.js applications
    - -v flag / Deploying Node.js applications
    - -p flag / Deploying Node.js applications
    - /bin/bash command / Deploying Node.js applications

S

- sales closing transactions / Scaling based on business parameters
- sample domain service, domain-driven design (DDD)
  - about / Sample domain service
  - entity implementation / Entity implementation
  - repository implementation / Repository implementation
  - service implementation / Service implementation
- sample REST program
  - about / Sample REST program
  - REST controller class, writing / Writing the REST controller class
  - app, executing / Making a sample REST app executable
- scalability
  - about / Scalability
- Scale Cube
  - about / Elastically and selectively scalable
  - reference / Elastically and selectively scalable
- scopes
  - $rootScope / Scopes
  - $scope / Scopes
- Scumblr / Scumblr
- Search API Gateway microservice / Deploying microservices in Docker
- Search microservice / Deploying microservices in Docker
- Search Providers / Scumblr
- Secure Shell (SSH)
  - about / SSH – encrypting the communications
- security
  - infrastructure logical security / Infrastructure logical security
- Seed project
  - reference link / Setting up the web app
- Seneca
  - about / API aggregation, Seneca – a microservices framework, How abstract is too abstract?, Seneca – a simple puzzle that makes our lives easier
  - installing / Installing Node.js, npm, Seneca, and PM2
  - Inversion of control / Inversion of control done right
  - pattern matching / Pattern matching in Seneca
  - plugins, writing / Writing plugins
  - web server integration / Web server integration
  - as Express middleware / Seneca as Express middleware
  - data storage / Data storage
  - and promises / Seneca and promises
- seneca-web / Seneca as Express middleware
- servers

    - about / Nomad
- service-oriented architecture (SOA) / Evolution of µServices
- Service-Oriented Architectures (SOA)
    - about / SOA versus microservices
    - versus microservices / SOA versus microservices
- Service-Oriented Integration (SOI) / Service-oriented integration
- service availability
    - registering / Eureka for registration and discovery
    - and discovery / Eureka for registration and discovery
- service endpoint design consideration
    - about / Service endpoint design consideration
    - contract design / Contract design
    - protocol selection / Protocol selection
- Service Oriented Architecture (SOA) / Best practices and principals
- services
    - monitoring / Monitoring services
    - monitoring, PM2 used / Monitoring using PM2 and Keymetrics
    - monitoring, Keymetrics used / Monitoring using PM2 and Keymetrics
- session handling
    - and security / Session handling and security
- shopping logic / Microservices are distributed and dynamic
- Simian Army / Reliability monitoring service – Simian Army
    - about / Simian Army – the active monitoring from Spotify
    - URL / Simian Army – the active monitoring from Spotify
    - throughput / Throughput and performance degradation
    - performance degradation / Throughput and performance degradation
- simple metric
    - about / Simple metric
- Simple Object Access Protocol (SOAP) / Evolution of µServices
- Single-Page Application (SPA)
    - about / Why Node.js?
  / Micromerce – the big picture
- single-threaded applications / Single-threaded applications and exceptions
- single database
    - native queries / Native queries
    - stored procedures / Stored procedures
- Single Euro Payments Area (SEPA)
    - about / How small is too small?
- single page applications (SPAs) / UI-Router
- single point of failure
    - about / Simian Army – the active monitoring from Spotify
- single sign-on (SSO)
    - about / User interfaces and web applications

- Sinon / Sinon.JS – a mocking framework
- Sinon.JS / Testing microservices in Node.js
- Slack
  - about / Monitoring application exceptions
- small size microservices
  - about / How small is too small?
- smart services
  - about / Smart services, dumb communication pipes
- SMTP server / Supporting the building of self-organizing systems
- SOAP
  - about / Why is replaceability important?
- SOA principles
  - reference / Characteristics of services in a microservice
- software-defined infrastructure / Cloud as the self-service infrastructure for Microservices
- Software as a service (SaaS)
  - about / Monitoring using PM2 and Keymetrics
- software deployment
  - concepts / Concepts in software deployment
  - continuous integration (CI) / Continuous integration
  - continuous delivery (CD) / Continuous delivery
  - with PM2 / Deployments with PM2
- Software Development Life Cycle (SDLC)
  - about / Concepts in software deployment
- SOLID
  - about / SOLID design principles
  - principles / SOLID design principles
- SOLID design pattern
  - reference / Single responsibility per service
- SOLID design principles
  - single-responsibility principle / Then the microservices appeared
  - open for extension, closed for modification / Then the microservices appeared
  - Liskov substitution / Then the microservices appeared
  - interface segregation / Then the microservices appeared
  - dependency injection / Then the microservices appeared
- span
  - about / Distributed tracing with Spring Cloud Sleuth
- Spinnaker / Deployment and delivery – Spinnaker with Aminator
- Spotify
  - about / Microservices in the real world, Simian Army – the active monitoring from Spotify
- SPR-9888
  - reference link / Spring Boot overview
- Spread algorithm / What does cluster management do?
- Spring Boot

    - configuration / Spring Boot configuration
    - overview / Spring Boot overview
    - adding, to rest sample / Adding Spring Boot to the rest sample
    - Jetty-embedded server, adding / Adding a Jetty-embedded server
    - reference link / NetBeans IDE installation and setup, References
    - configuration, reference link / NetBeans IDE installation and setup
    - used, for building RESTful microservices / Using Spring Boot to build RESTful microservices
    - about / Getting started with Spring Boot
- Spring Boot actuators
    - about / Spring Boot actuators
    - monitoring, JConsole used / Monitoring using JConsole
    - monitoring, SSH used / Monitoring using SSH
- Spring Boot command-line tool
    - reference / Developing the Spring Boot microservice using the CLI
- Spring Boot configuration
    - about / The Spring Boot configuration
    - autoconfiguration / Understanding the Spring Boot autoconfiguration
    - default configuration values, overriding / Overriding default configuration values
    - configuration file location, changing / Changing the location of the configuration file
    - custom properties, reading / Reading custom properties
    - .yaml file, using / Using a .yaml file for configuration
    - multiple configuration profiles, using / Using multiple configuration profiles
    - properties, reading / Other options to read properties
- Spring Boot Java microservice
    - developing, STS used / Developing the Spring Boot Java microservice using STS
    - pom file, examining / Examining the POM file
    - Application.java, examining / Examining Application.java
    - application.properties, examining / Examining application.properties
    - ApplicationTests.java, examining / Examining ApplicationTests.java
    - testing / Testing the Spring Boot microservice
- Spring Boot messaging
    - implementing / Implementing Spring Boot messaging
- Spring Boot microservice
    - developing, CLI used / Developing the Spring Boot microservice using the CLI
    - developing, Spring Initializr used / Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example
- Spring Boot security
    - implementing / Implementing Spring Boot security
    - microservice, securing with basic security / Securing microservices with basic security
    - microservice, securing with OAuth2 / Securing a microservice with OAuth2
- Spring Cloud
    - reference link / References

    - defining / What is Spring Cloud?
    - and Cloud Foundry, comparing / What is Spring Cloud?
    - releases / Spring Cloud releases
    - components / Components of Spring Cloud
    - capabilities / Components of Spring Cloud
    - for scaling microservices / Scaling microservices with Spring Cloud
- Spring Cloud Config
    - defining / Spring Cloud Config
    - Config server, using / What's next?
    - Config server, setting up / Setting up the Config server
    - Config server URL, defining / Understanding the Config server URL
    - configuration changes, handling / Handling configuration changes
    - Spring Cloud Bus, for propagating configuration changes / Spring Cloud Bus for propagating configuration changes
    - high availability, setting up for Config server / Setting up high availability for the Config server
    - Config server health, monitoring / Monitoring the Config server health
    - Config server, for configuration files / Config server for configuration files
    - changes, completing for Config server usage / Completing changes to use the Config server
- Spring Cloud examples
    - references / Setting up the Config server
- Spring Initializer
    - reference link / Spring Boot overview
- Spring Initializr project
    - reference / Getting started with Spring Boot
- Spring Initilizr
    - reference / Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example
- Spring OAuth2
    - reference link / References
- Spring Security
    - used, for OAuth implementation / OAuth implementation using Spring Security
    - reference link / References
- Spring Starter project
    - URL / Feign as a declarative REST client
- Spring Tool Suite 3.7.2 (STS)
    - reference / Setting up a development environment
- Spring Tool Suite IDE
    - reference link / NetBeans IDE installation and setup
- SSL
    - enabling / Enabling Secure Socket Layer
    - about / Enabling Secure Socket Layer
- STARS

    - about / Choosing the image
- strategic design and principles
    - about / Strategic design and principles
    - bounded context / Bounded context
    - continuous integration / Continuous integration
    - context map / Context map
- streams
    - for reactive microservices / Streams for reactive microservices
- Stubs / Sinon.JS – a mocking framework
- Subject Alternative Names (SAN) / Enabling Secure Socket Layer
- subnet / Docker containers
- supporting capabilities, microservices capability model
    - about / Supporting capabilities
    - software defined load balancer / Supporting capabilities
    - central log management / Supporting capabilities
    - service registry / Supporting capabilities
    - security service / Supporting capabilities
    - service configuration / Supporting capabilities
    - testing tools / Supporting capabilities
    - monitoring and dashboards / Supporting capabilities
    - dependency and CI management / Supporting capabilities
    - Data lake / Supporting capabilities
    - reliable messaging / Supporting capabilities
- Swagger
    - APIs, documenting with / Documenting APIs with Swagger
- SystemdNspawn
    - about / What are containers?
- µService
    - developing / Developing and implementing µServices
    - implementing / Developing and implementing µServices
    - entities / Developing and implementing µServices
    - services / Developing and implementing µServices
    - Repository object / Developing and implementing µServices
    - Restaurant µService / Restaurant µService
    - booking / Booking and user services
    - user services / Booking and user services
    - registration / Registration and Discovery service (Eureka service)
    - discovery service (Eureka service) / Registration and Discovery service (Eureka service)
    - Maven dependency / Registration and Discovery service (Eureka service)
    - startup class / Registration and Discovery service (Eureka service)
    - spring configurations / Registration and Discovery service (Eureka service)
    - execution / Execution
- µServices

- evolution / Evolution of µServices
- monolithic architecture overview / Monolithic architecture overview
- used, for specifying limitations of monolithic architecture versus solution / Limitation of monolithic architecture versus its solution with µServices

T

- t2.large / Installing Docker on AWS EC2
- t2.micro EC2 instances / Setting up AWS
- target architecture
  - about / Target architecture
  - internal layering, of microservices / Internal layering of microservices
  - microservices, orchestrating / Orchestrating microservices
  - integration, with other systems / Integration with other systems
  - shared libraries, managing / Managing shared libraries
  - exceptions, handling / Handling exceptions
- target implementation view
  - about / Target implementation view
  - implementation projects / Implementation projects
  - project, running / Running and testing the project
  - project, testing / Running and testing the project
- target test environments
  - about / Target test environments
  - development environment / Target test environments
  - integration test environment / Target test environments
  - performance and diagnostics / Target test environments
  - functional test environment / Target test environments
  - UAT environment / Target test environments
  - staging / Target test environments
  - production / Target test environments
- task groups
  - about / Nomad
- tasks
  - about / Nomad
- Teamcity
  - reference link / References
- technical debt
  - about / Why is replaceability important?
- technology alignment
  - about / Technology alignment
- technology heterogeneity
  - about / Technology alignment, Technology heterogeneity
- technology metamodel
  - about / The technology metamodel
- Test-driven development (TDD) / Functional testing
  - about / TDD – Test-driven development
- Test Driven Development (TDD) / System/end-to-end test automation
- testing

    - enabling / Testing
- Testing in Production (TiP) / Testing in production
- test strategy
    - about / Test strategy
- ThreadLocal
    - about / Node.js event loop – easy to learn and hard to master
- TPM (Transactions Per Minute) / Understanding the execution flow
- trace
    - about / Distributed tracing with Spring Cloud Sleuth
- traceability
    - about / Traceability
    - logging / Logging
    - tracing requests / Tracing requests
    - auditing / Auditing
    - HTTP codes / HTTP codes
- tracing requests
    - about / Tracing requests
- traditional web application
    - moving, to microservices / Moving from traditional web applications to microservices
- transaction boundaries
    - setting up / Setting up transaction boundaries
    - use cases, altering / Altering use cases to simplify transactional requirements
    - distributed transaction scenarios / Distributed transaction scenarios
- transaction processing facility (TPF) / Overcoming the legacy hotspot
- transition plan
    - establishing / Evolutionary approach
- transition point of view
    - queries / Evolutionary approach
- Transport Security Layer (TSL) / Enabling Secure Socket Layer, OAuth 2.0
- Turbine
    - reference link / References
- Twitter
    - URL / Enabling Secure Socket Layer, OAuth 2.0 specification – concise details, Microservices early adopters
    - about / Microservices early adopters

U

- Uber
  - about / Microservices early adopters
  - URL / Microservices early adopters
- UI - API aggregation
  - about / Micromerce – the big picture, The UI – API aggregation
  - frontend microservice, need for / Need for frontend microservice
  - code / The code
  - service degradation / Service degradation – when the failure is not a disaster
  - circuit breakers / Circuit breakers
  - Seneca puzzle / Seneca – a simple puzzle that makes our lives easier
  - Seneca, and promises / Seneca and promises
- Unified Model Language (UML) / Ubiquitous language
- unique identity
  - creating, for objects / Entities
- unit testing
  - about / Unit testing
- unit tests / Unit tests
- use cases, microservices
  - about / Microservice use cases
  - microservices early adopters / Microservices early adopters
  - monolithic migrations / The common theme is monolithic migrations
- User Interface (UI) / Multilayered architecture
- user interfaces
  - and web applications / User interfaces and web applications

V

- Vagrant
  - URL / SSH – encrypting the communications
- value objects (VOs) / Value objects
- Veritas Risk Advisor (VRA) / Monitoring microservice dependencies
- virtual engine (VE)
  - about / What are containers?
- virtual machines (VMs) / Deployment using a container such as Docker
  - versus containers / The difference between VMs and containers
  - Zen / The difference between VMs and containers
  - Hyper-V / The difference between VMs and containers

W

- web app
  - setting up / Setting up the web app
  - reference links / Setting up the web app
- web applications
  - and user interfaces / User interfaces and web applications
- white box testing / Unit tests
- white list
  - about / Input validation
- willdurand/elk ELK Docker image
  - reference link / References
- WORKDIR command
  - about / Automating Docker container creation
- workers
  - about / Node.js event loop – easy to learn and hard to master

X

- XebiaLabs periodic table
  - reference / Automating the continuous delivery pipeline
- XSS attacks
  - persistent / Cross-site scripting
  - non-persistent / Cross-site scripting

Y

- Yet Another Markup Language (YAML) / Documenting APIs with Swagger

Z

- ZooKeeper
  - installing / Installing ZooKeeper, Mesos, and Marathon
  - reference / Installing ZooKeeper, Mesos, and Marathon
  - configuring / Configuring ZooKeeper
  - running as services / Running Mesos, Marathon, and ZooKeeper as services
- Zuul / Edge (proxy) server – Zuul
  - setting up / Setting up Zuul
  - using / Setting up Zuul
  - high availability / High availability of Zuul
  - completing, for all services / Completing Zuul for all other services
- Zuul proxy
  - defining, as API gateway / Zuul proxy as the API gateway