

O'REILLY®



Cloud Native Java

DESIGNING RESILIENT SYSTEMS WITH SPRING BOOT,
SPRING CLOUD, AND CLOUD FOUNDRY

Josh Long & Kenny Bastani

1. [Foreword](#)
2. [Preface](#)
 1. [Conventions Used in This Book](#)
 2. [Using Code Examples](#)
 3. [Safari® Books Online](#)
 4. [How to Contact Us](#)
 5. [Acknowledgments](#)

- 3. [1. Bootcamp: Introducing Spring Boot](#)
 - 1. [Getting Started with the Spring Tool Suite](#)
 - 1. [Installing Spring Tool Suite](#)
 - 2. [Spring Boot Starters](#)
 - 3. [Getting Started with the Spring Initializr](#)
 - 1. [Generating Spring Boot applications](#)
 - 4. [The Spring Guides](#)
 - 5. [Auto-Configuration](#)
 - 1. [Spring Boot Configuration](#)
- 4. [2. The Cloud Native Application](#)
 - 1. [Platforms](#)
 - 1. [Building Platforms](#)
 - 2. [The Patterns](#)
 - 3. [Netflix's Story](#)
 - 1. [Splitting the Monolith](#)
 - 2. [Netflix OSS](#)
 - 4. [The Twelve Factors](#)
- 5. [3. 12-Factor Application Style Configuration](#)
 - 1. [The Confusing Conflation of “Configuration”](#)
 - 2. [Support in Spring framework](#)
 - 1. [The PropertyPlaceholderConfigurer](#)
 - 2. [The Environment Abstraction and @Value](#)
 - 3. [Profiles](#)
 - 3. [Bootiful Configuration](#)
 - 4. [Centralized, Journaled Configuration with the Spring Cloud Configuration Server](#)
 - 1. [Security](#)
 - 5. [Refreshable Configuration](#)
 - 6. [Next Steps](#)
- 6. [4. Testing](#)
 - 1. [Testing in Spring Boot](#)
 - 2. [Integration Testing](#)
 - 1. [Test Slices](#)
 - 2. [Mocking in Tests](#)
 - 3. [Testing Annotations](#)
 - 3. [End-to-end Testing](#)
 - 1. [Testing Distributed Systems](#)

2. [Consumer-driven Contract Testing](#)
3. [Spring Cloud Contracts](#)
4. [Continuous Integration](#)
5. [Functional Testing](#)
4. [Behavior-driven Testing](#)
7. [**5. REST APIs**](#)
 1. [Leonard Richardson's Maturity Model](#)
 2. [Simple REST APIs with Spring MVC](#)
 3. [Content Negotiation](#)
 1. [Reading and Writing Binary Data](#)
 2. [Using Google Protocol Buffers](#)
 4. [Error Handling](#)
 5. [Hypermedia](#)
 1. [Media Type and Schema](#)
 6. [API Versioning](#)
 7. [Documenting REST APIs](#)
 8. [The Client Side](#)
 1. [REST Clients for Ad-Hoc Exploration and Interaction](#)
 2. [The RestTemplate](#)
8. [**6. Managing Data**](#)
 1. [Modeling Data](#)
 1. [RDBMS](#)
 2. [NoSQL](#)
 2. [Spring Data](#)
 1. [Structure of a Spring Data Application](#)
 2. [Domain Class](#)
 3. [Repositories](#)
 4. [Organizing Java Packages for Domain Data](#)
 3. [Auto-configuration](#)
 4. [Bootstrapping Datasource Dependencies](#)
 1. [Datasource Connections](#)
 5. [JDBC Template](#)
 6. [Spring Data Examples](#)
 7. [Spring Data JPA](#)
 8. [Spring Data MongoDB](#)
 9. [Spring Data Neo4j](#)
 10. [Spring Data Redis](#)

1. [Caching](#)
11. [Next Steps](#)
9. [7. Data Integration](#)
 1. [Distributed Transactions](#)
 2. [The Saga Pattern](#)
 3. [Batch workloads with Spring Batch](#)
 4. [Scheduling](#)
 5. [Isolating Failures and Graceful Degradation](#)
 6. [Task Management](#)
 7. [Process-Centric Integration with Workflow](#)
 8. [Event Driven Architectures with Spring Integration](#)
 1. [Messaging Endpoints](#)
 2. [From Simple Components, Complex Systems](#)
 9. [Message Brokers, Bridges, the Competing Consumer Pattern and Event-Sourcing](#)
10. [Spring Cloud Stream](#)
 1. [A Stream Producer](#)
 2. [A Stream Consumer](#)
11. [Spring Cloud Data flow](#)
 1. [Streams](#)
 2. [Tasks](#)
12. [Next Steps](#)
10. [8. Using Spring Boot with Java EE](#)
 1. [Compatibility and Stability](#)
 2. [Dependency Injection with JSR 330 \(and JSR 250\)](#)
 3. [Building REST APIs with JAX-RS \(Jersey\)](#)
 4. [JTA and XA Transaction Management](#)
 1. [Resource-Local Transactions with Spring's PlatformTransactionManager](#)
 2. [Global Transactions with the Java Transaction API \(JTA\)](#)
 5. [Deployment in a Java EE Environment](#)
 6. [Final Word](#)
11. [9. Service Brokers](#)
 1. [Cloud Foundry](#)
 2. [Services Marketplace](#)
 1. [Creating Services](#)
 2. [Binding Applications](#)

- 3. [Cloud Foundry Service Brokers](#)
 - 3. [Cloud Controller](#)
 - 4. [Service Broker API](#)
 - 5. [Implementing a Service Broker with Spring Boot](#)
 - 1. [Amazon S3 Service Broker](#)
 - 2. [The Service Catalog](#)
 - 3. [Service Instances](#)
 - 4. [Service Bindings](#)
 - 6. [Deploying the Service Broker](#)
 - 1. [Releasing with BOSH](#)
 - 2. [Releasing with Cloud Foundry](#)
 - 3. [Consuming Service Instances](#)
 - 4. [Extending Spring Boot](#)
- 12. [10. The Forklifted Application](#)
 - 1. [The Contract](#)
 - 2. [Migrating Application Environments](#)
 - 1. [the Out-of-the-Box Buildpacks](#)
 - 2. [Customizing Buildpacks](#)
 - 3. [Containerized Applications](#)
 - 3. [Soft-Touch Refactoring to get your application into the cloud](#)
 - 1. [Talking to Backing Services](#)
 - 2. [Achieving Service Parity with Spring](#)
 - 4. [Next Steps](#)
- 13. [11. The Observable System](#)
 - 1. [The New Deal](#)
 - 2. [Visibility and Transparency](#)
- 14. [12. Push vs. Pull Observability and Resolution](#)
 - 1. [Capturing an Application's Present Status with Actuator](#)
 - 2. [Metrics](#)
 - 3. [Identifying Your Service with the /info Endpoint](#)
 - 4. [Health Checks](#)
 - 5. [Application Logging](#)
 - 6. [Distributed Tracing](#)
 - 1. [Finding Clues with Spring Cloud Sleuth](#)
 - 2. [How Much Data is Enough?](#)
 - 3. [OpenZipkin: a Picture is worth a Thousand Traces](#)
 - 4. [The OpenTracing Initiative](#)

- 7. [Dashboards](#)
 - 1. [Monitoring Potentially Risky Service Calls with the Hystrix Dashboard](#)
 - 2. [Codecentric's Spring Boot Admin](#)
 - 3. [Ordina Microservices Dashboard](#)
 - 8. [Remediation](#)
 - 9. [Next Steps](#)
- 15. [13. The Application Centric Cloud](#)
 - 1. [Portable Applications](#)
 - 2. [Cattle](#)
 - 3. [Containerized Workloads](#)
 - 1. [Scheduler](#)
 - 2. [Service Discovery](#)
 - 4. [The Application Framework](#)
 - 1. [Spring Boot](#)
 - 2. [Spring Cloud](#)
 - 16. [14. Continuous Delivery](#)
 - 1. [Start Here](#)
 - 2. [Every Build is a Release Candidate](#)
 - 3. [Version Control Everything](#)
 - 17. [15. Edge Services](#)
 - 1. [Greetings](#)
 - 2. [A Simple Edge Service](#)
 - 3. [Netflix Feign](#)
 - 4. [Reactive Programming](#)
 - 5. [Proxies with Netflix Zuul](#)
 - 1. [A Custom Zuul Filter](#)
 - 6. [Security on the Edge](#)
 - 1. [OAuth](#)
 - 2. [Building an OAuth Authorization Server](#)
 - 3. [Building an Implicit OAuth Client with Angular.js](#)
 - 4. [Building a Social OAuth Authorization Server](#)
 - 18. [16. Routing](#)
 - 1. [Locational Decoupling with Service Registration and Discovery](#)
 - 2. [The DiscoveryClient Abstraction](#)
 - 3. [Cloud Foundry Route Services](#)
 - 4. [Next Steps](#)

19. [Index](#)

Cloud Native Java

First Edition

Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry

Josh Long & Kenny Bastani

Cloud Native Java

by Josh Long, Kenny Bastani

Copyright © 2016 Josh Long, Kenny Bastani. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Editor: Brian Foster
- Developmental Editor: Nan Barber
- Month Year: First Edition

Revision History for the First Edition

- 2015-11-15: First Early Release
- 2015-12-14: Second Early Release
- 2016-01-21: Third Early Release
- 2016-03-01: Fourth Early Release
- 2016-04-20: Fifth Early Release
- 2016-05-13: Sixth Early Release
- 2016-05-31: Seventh Early Release
- 2016-09-23: Eighth Early Release
- 2016-10-31: Ninth Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449370787> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Cloud Native Java*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-4493-7464-8

[???

Foreword

Forward goes here

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

Tip

This element signifies a tip or suggestion.

Note

This element signifies a general note.

Warning

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Note

Safari Books Online is an on-demand digital library that delivers expert [content](#) in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of [plans and pricing](#) for [enterprise](#), [government](#), [education](#), and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds [more](#). For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at
<http://www.oreilly.com/catalog/0636920038252>.

To comment or ask technical questions about this book, send email to
bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Chapter 1. Bootcamp: Introducing Spring Boot

Spring Boot provides a way to create production-ready Spring applications with minimal setup time. The primary goals behind the creation of the Spring Boot project are central to the idea that users should be able to get up and running quickly with Spring. Spring Boot also takes an opinionated view of the Spring platform and third-party libraries.

An opinionated view means that Spring Boot lays out a framework of abstractions that are common to all Spring projects. This opinionated view provides the plumbing that all projects need but without getting in the way of the developer. By doing this, Spring Boot makes it simple to swap components when project requirements change.

This chapter will introduce you to building Spring Boot applications. The topics we will go over are:

- The Spring Tool Suite
- Spring Initializr
- Starter Projects
- The Spring Guides
- Auto-configuration

Getting Started with the Spring Tool Suite

The Spring Tool Suite (STS) is an Eclipse distribution that is customized for developing Spring applications. STS is a freely available IDE under the terms of the Eclipse Public License. While there are many IDE options available for developing your Spring applications, STS provides a set of features that are tailored for Spring-based development.

Installing Spring Tool Suite

Let's get started with downloading and installing the Spring Tool Suite, available from <http://www.spring.io>.

- Go to <https://spring.io/tools/sts>
- Choose **Download STS**
- Download, extract, and run STS

After you have downloaded, extracted, and have run the STS program, you will be prompted to choose a workspace location.

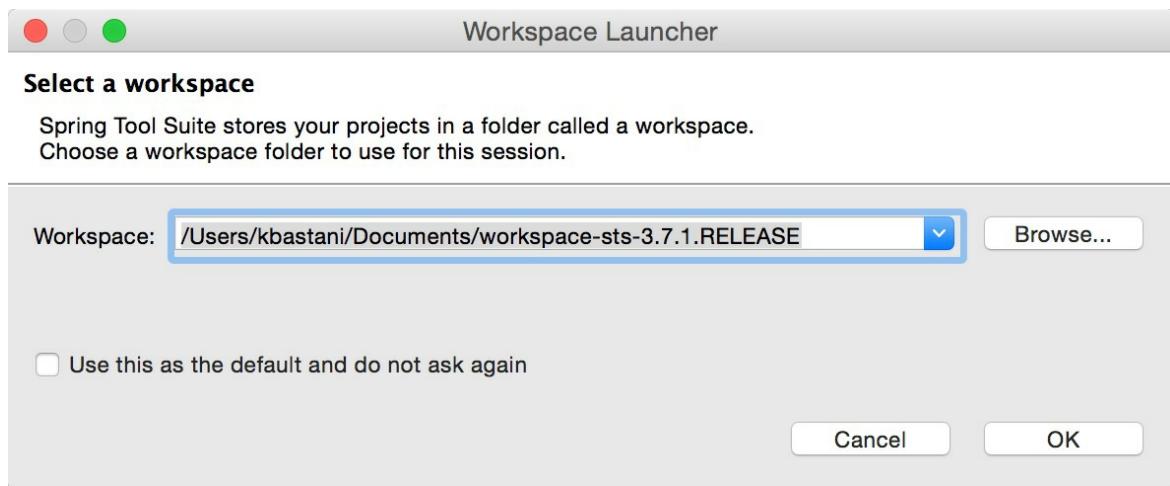


Figure 1-1. Choose your desired workspace location

Choose your desired workspace location and click **OK**. If you plan to use the same workspace location each time you run STS, click on the option "*Use this as the default and do not ask again*".

After you have provided a workspace location and clicked **OK**, the STS IDE will load for the first time.

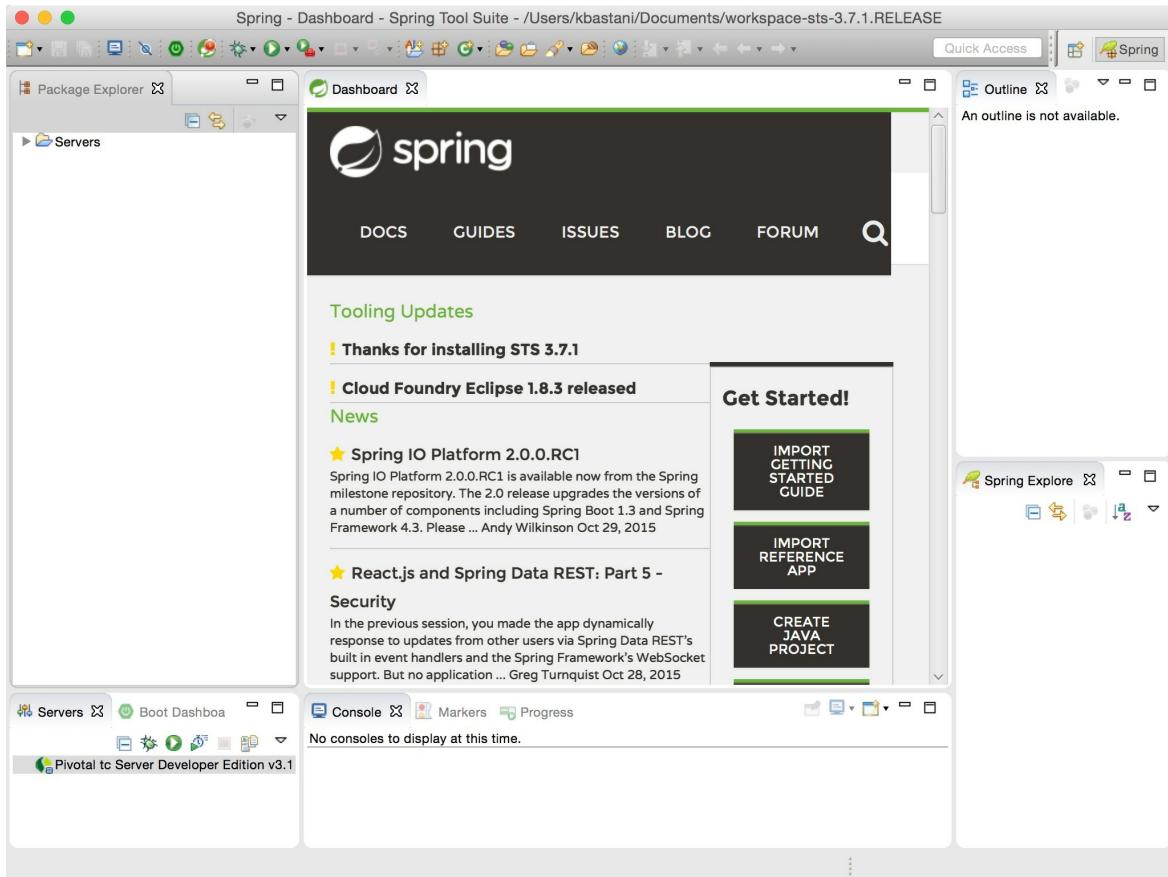


Figure 1-2. The STS dashboard

Your first Spring Boot application

Let's now use STS to create our first Spring Boot application. We're going to create a simple *Hello World* web service using the Spring Boot starter project for web.

To create a new Spring Boot application using a Spring Boot Starter project, choose from the menu **File > New > Spring Boot Starter Project**.

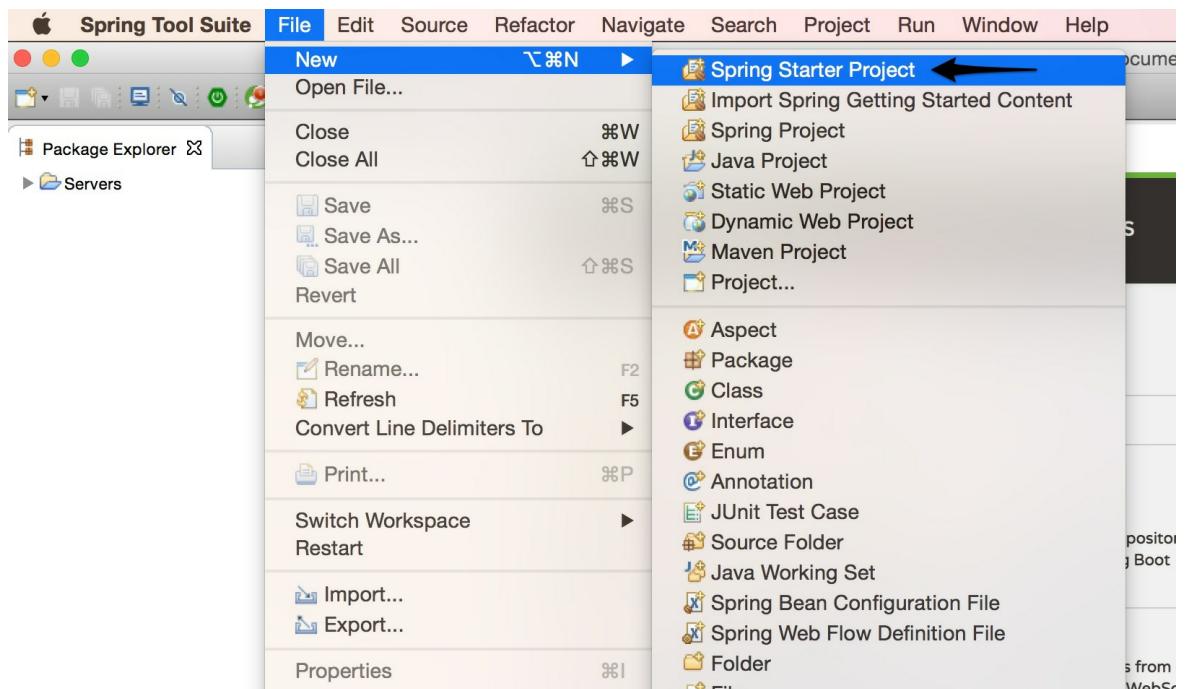


Figure 1-3. Create a new Spring Boot Starter Project

After choosing to create a new Spring Boot Starter Project, you will be presented with a dialog to configure your new Spring Boot application.

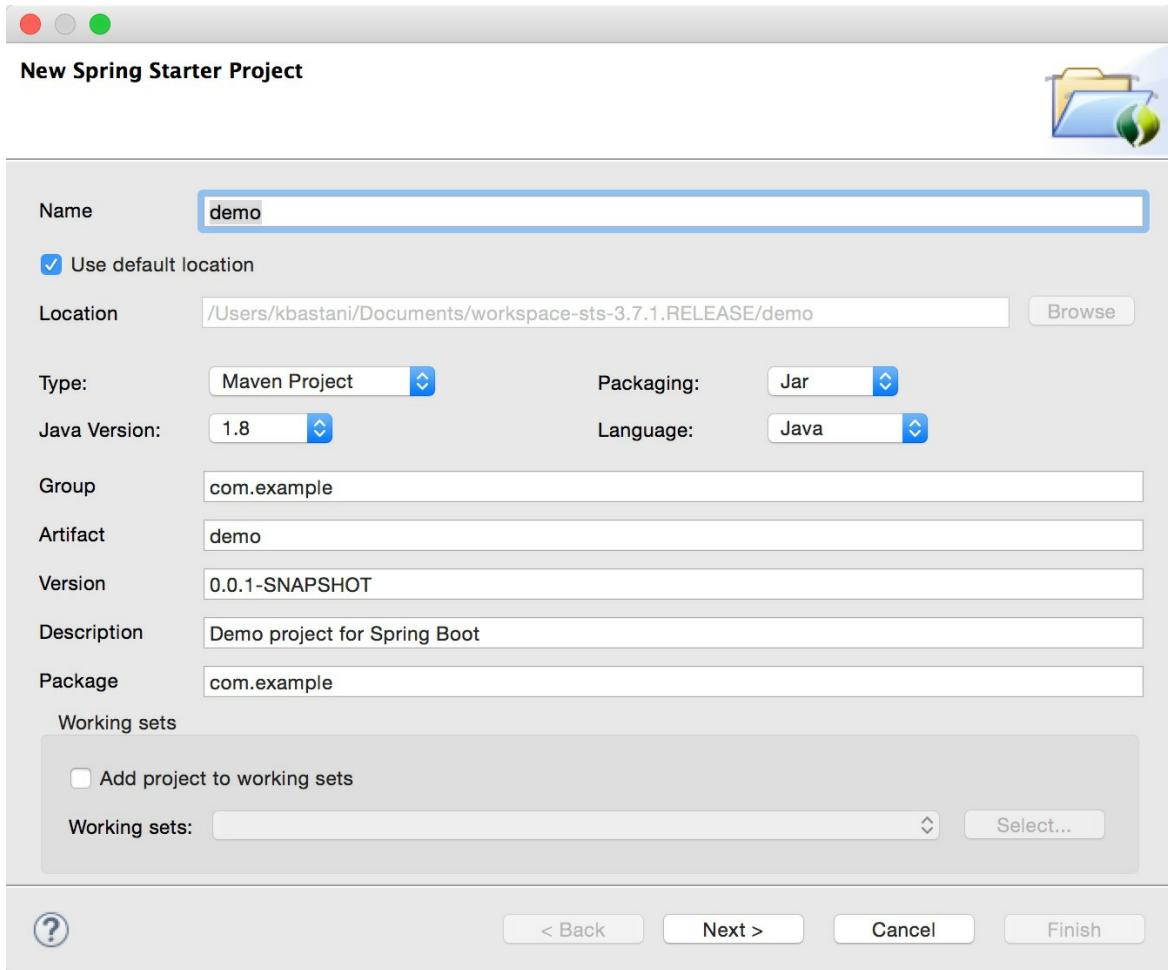


Figure 1-4. Configure your new Spring Boot Starter Project

You can configure your options, but for the purposes of this simple walkthrough, let's use the defaults and click **Next**. After clicking **Next**, you will be provided with a set of Spring Boot Starter projects that you can choose for your new Spring Boot application. For our first application we're going to choose the Spring Boot Starter Web project.

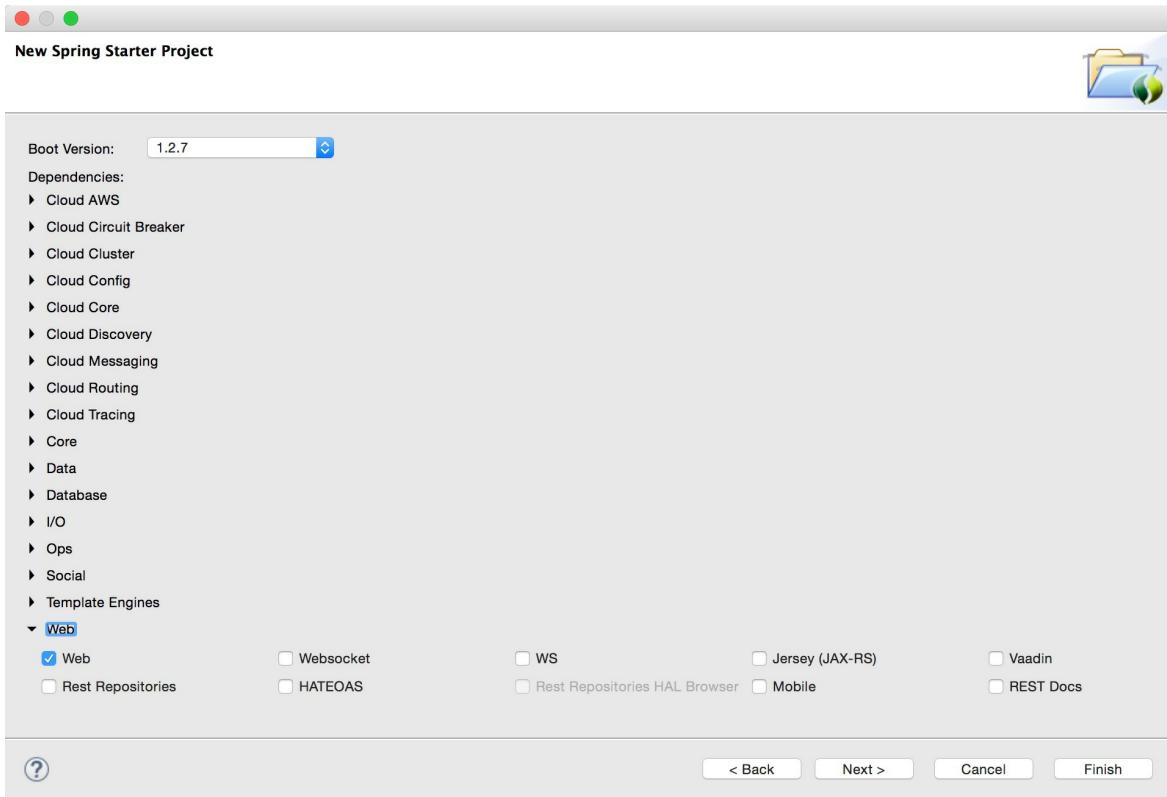


Figure 1-5. Choose your Spring Boot Start Project

From the list of sections, choose **Web** to expand its list of options. Now from the **Web** section, choose the option **Web** and click **Finish**. After you click **Finish**, your Spring Boot application with the *Spring Boot Starter Web* project dependency will be created.

Now that your project has been created and imported into the STS IDE, you will notice that a new project is available in your package explorer area.

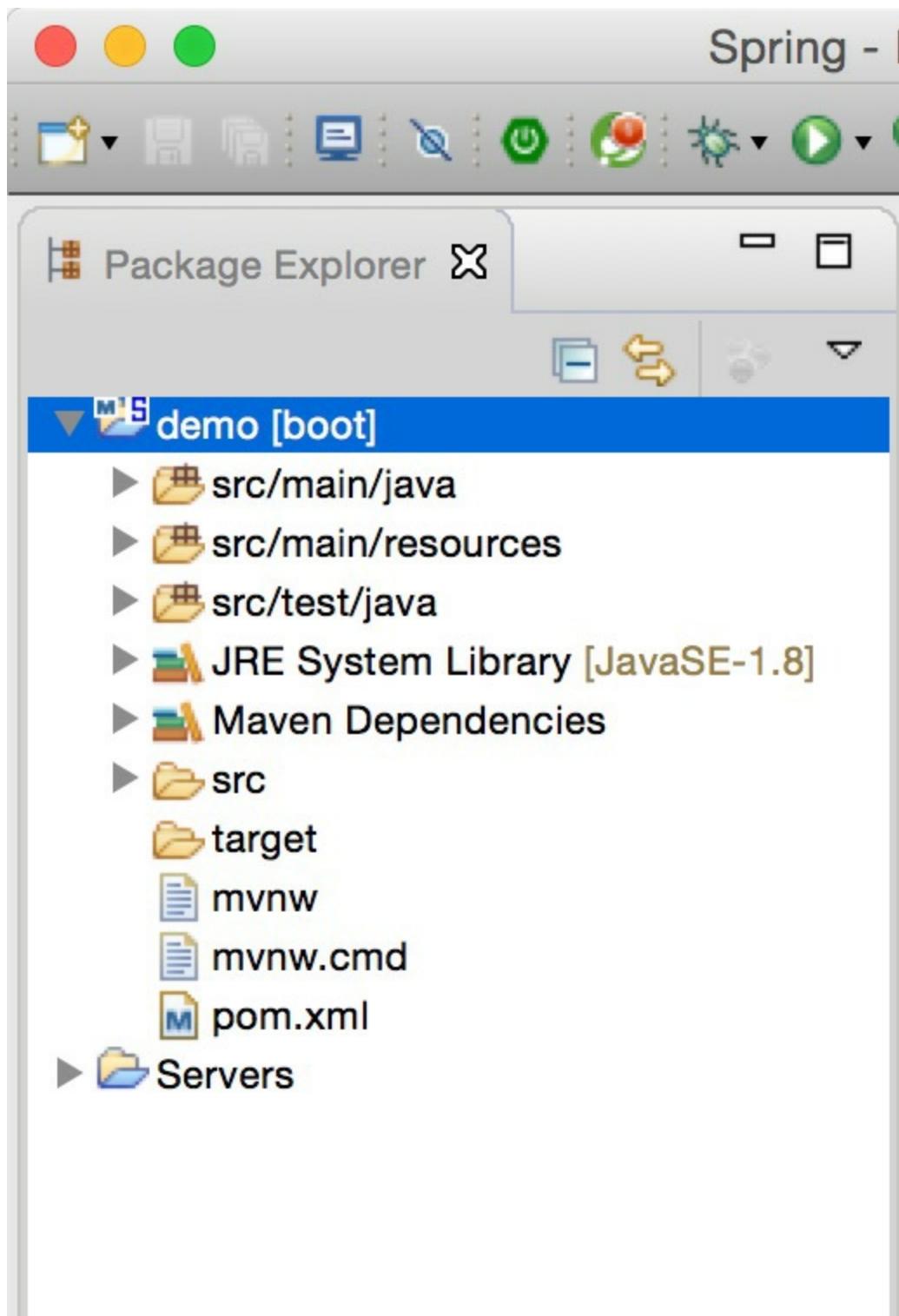


Figure 1-6. Expand the demo project from the package explorer

If you haven't already, expand the **demo [boot]** project and view the project

contents as shown in the screenshot above. From the expanded project files, navigate to **src/main/java/com/example/DemoApplication.java**.

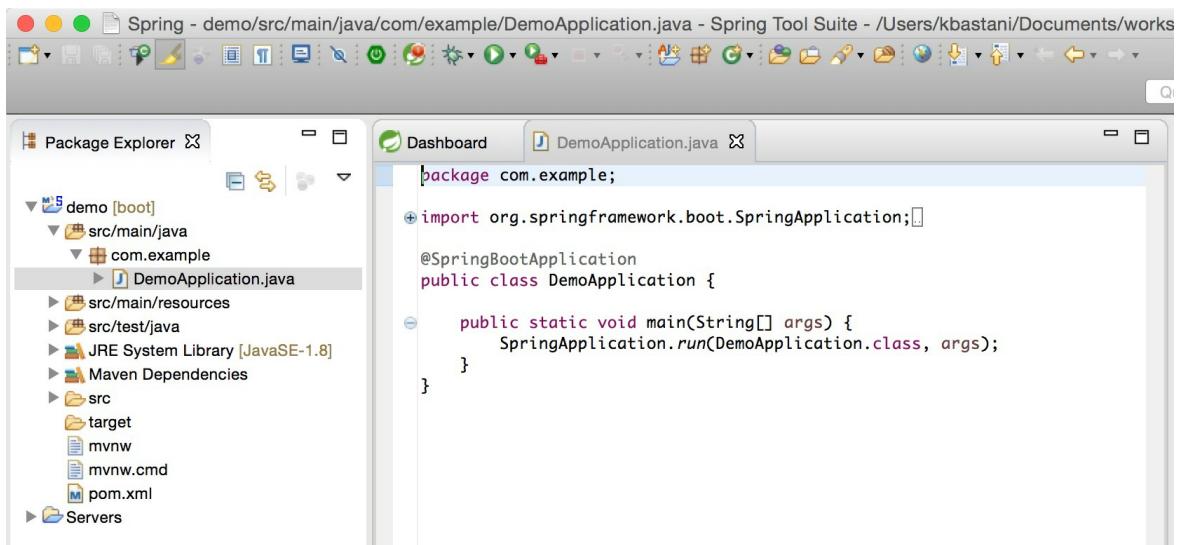


Figure 1-7. Navigate to DemoApplication.java

This is your main application class for your new Spring Boot application. STS has configured this project as a Spring Boot starter project with the **spring-boot-starter-web** dependency. If you navigate to your `pom.xml` file in your *Package Explorer* you will find that the following dependencies have been added.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>①
    </dependency>

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

①

The starter project for Web

②

A starter project for unit testing

Now that we understand the makeup of our Spring Boot application, let's create our first RESTful service by modifying `DemoApplication.java`. In `DemoApplication.java`, modify the contents so that it looks like the following example.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication ①
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args); ②
    }

    @RestController ③
    public static class Hello {
        @RequestMapping(value = "/hello") ④
        public String hello(
            @RequestParam(value = "name", defaultValue = "World")
                return "Hello, " + name;
        }
    }
}
```

①

Annotates a class as a Spring Boot application

②

This starts the Spring Boot application

③

Annotates a class as a REST controller

④

This maps a URL route to a controller method

⑤

This method has a query string parameter with a default value

Now that we have our code for a basic RESTful web service, let's go ahead and run the application. Run the application from the **Run > Run** menu.

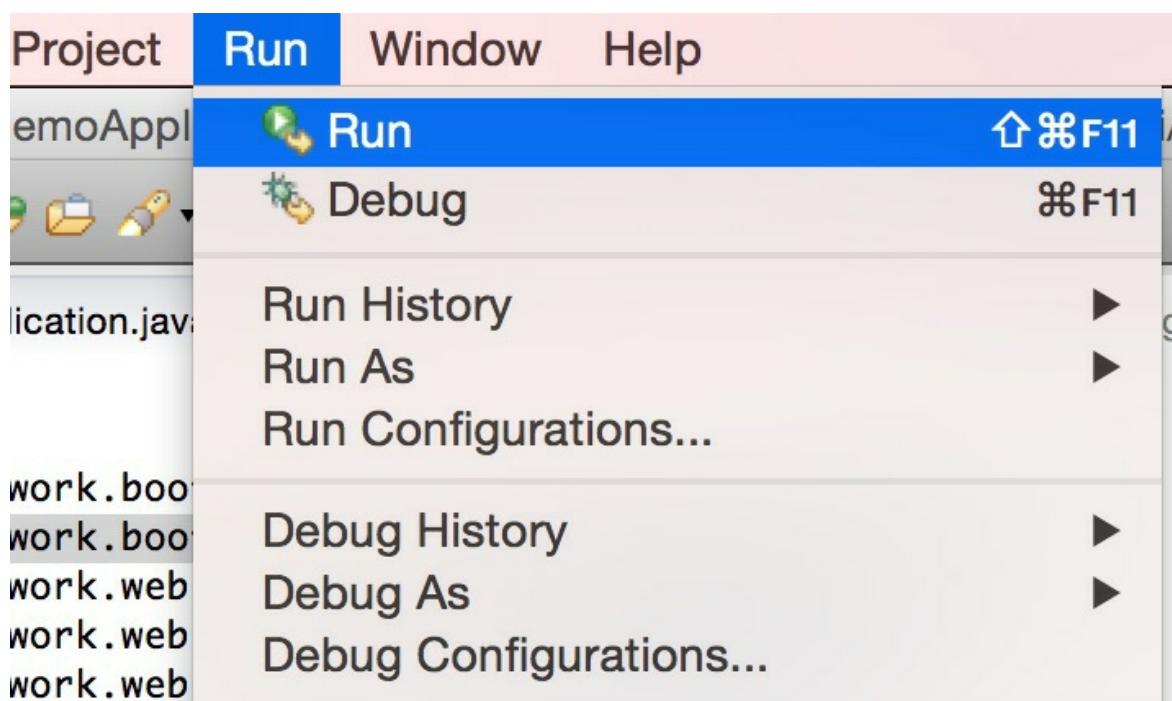


Figure 1-8. Run the Spring Boot application

After choosing the **Run** option from the menu, you'll be presented with a *Save and Launch* dialog. Choose the **DemoApplication.java** option and click **OK**.

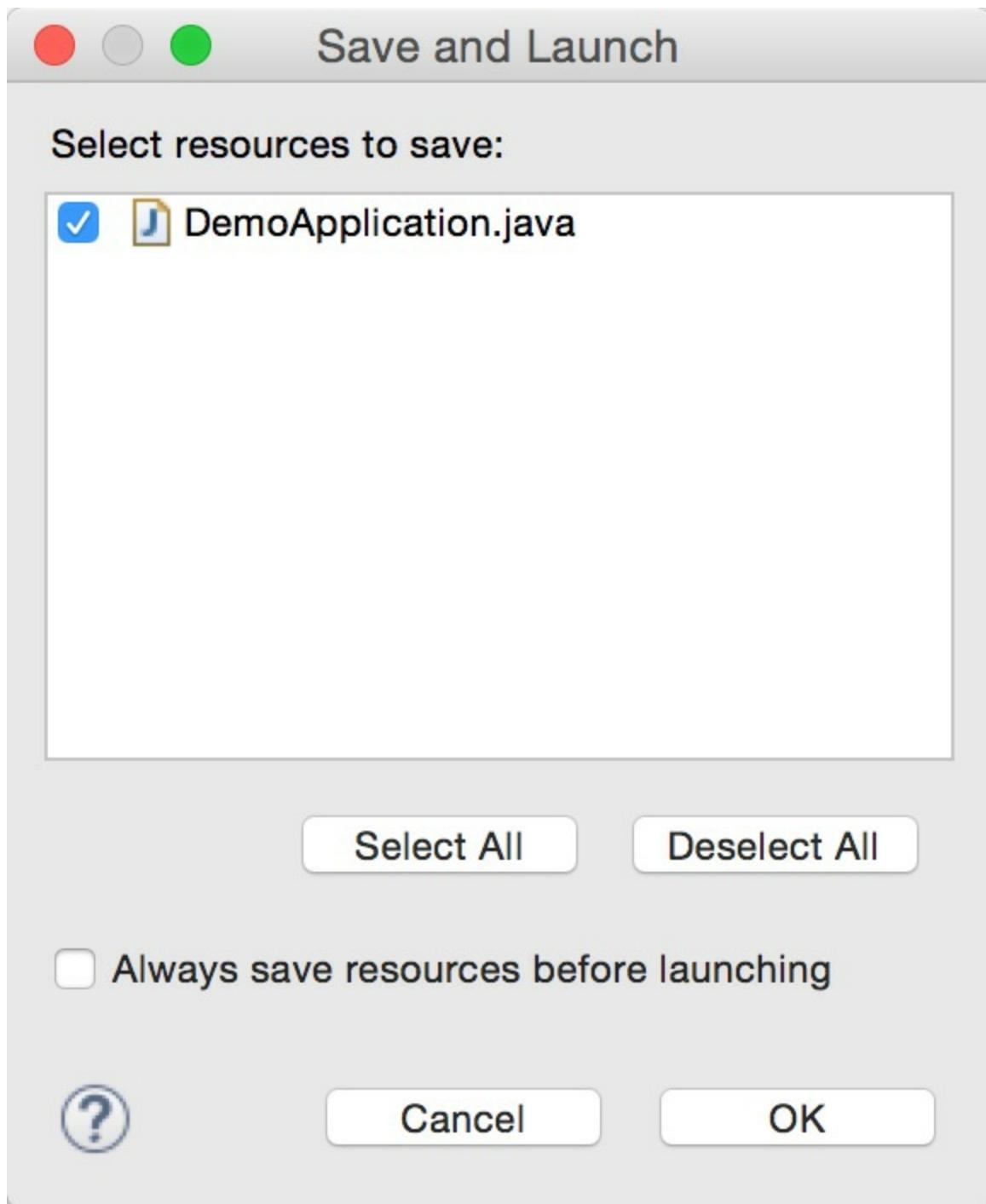


Figure 1-9. Choose DemoApplication.java and launch the application

Your Spring Boot application will now start up. If you look at your STS console, you should see the iconic Spring Boot ASCII art and the version of Spring Boot as it starts up. The log output of the Spring Boot application can

be seen here. You'll see that an embedded Tomcat server is being started up and launched on the default port of 8080. You can access your Spring Boot web service from <http://localhost:8080>.

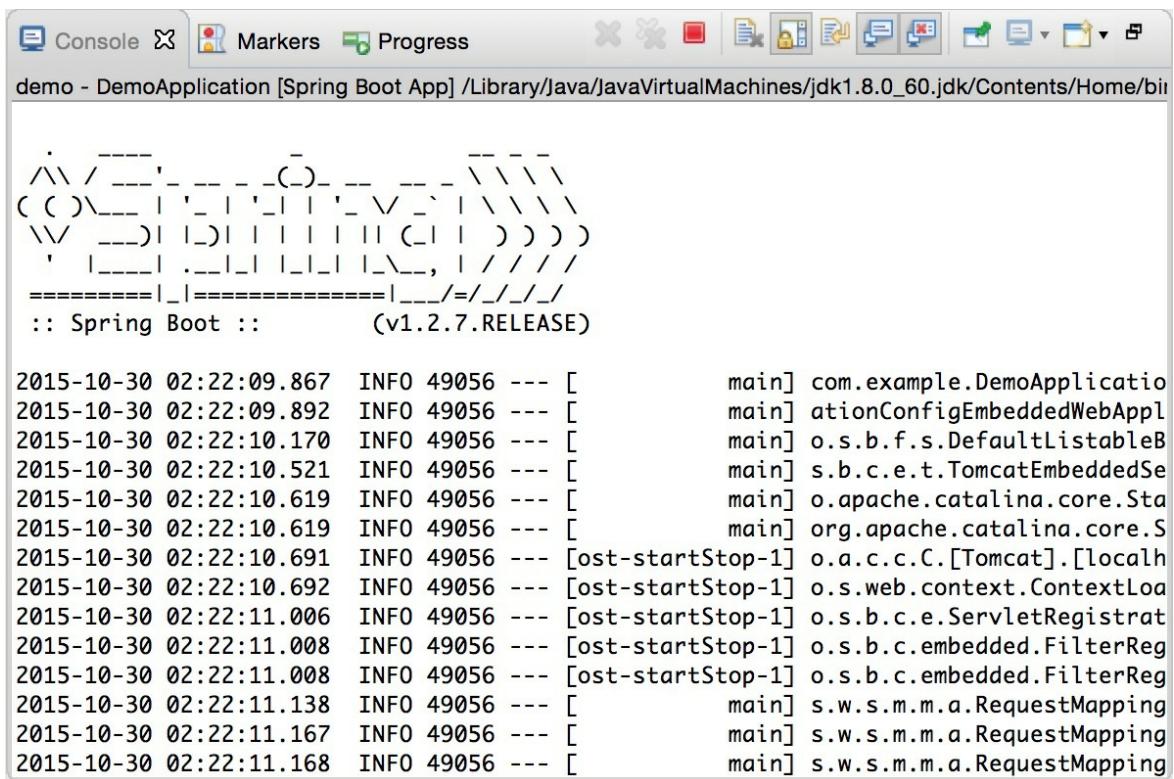


Figure 1-10. See the Spring Boot log output in the STS console

Now let's issue a `curl` request to our new web service from the command line.

Tip

If you do not have `curl` installed, you can install it from <http://curl.haxx.se/download.html>. We will be using `curl` extensively throughout the book to test our Spring Boot web applications.

From a command line interface, issue the following `curl` request:

```
curl -X GET -i http://localhost:8080/hello
```

You will see the following output.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/plain;charset=UTF-8
Content-Length: 12
Date: Fri, 30 Oct 2015 09:29:40 GMT
```

Hello, World

Now try appending a query string for `name` at the end of your request URL.

```
curl -X GET -i http://localhost:8080/hello?name=Kenny
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/plain;charset=UTF-8
Content-Length: 12
Date: Fri, 30 Oct 2015 09:32:20 GMT
```

Hello, Kenny

Congratulations! You've just created your first Spring Boot application as a RESTful service.

Spring Boot Starters

Spring Boot uses *Starter Projects* to help developers compose the ingredients that are needed for their applications. If we were to create a Spring application from scratch, we would need to worry about including a reference to each of the project dependencies for each of the features we need from the Spring ecosystem of libraries. Further, we'd have to define compatible versions for each of these dependencies, which can often cause problems due to transitive dependencies having a versioning conflict.

Spring Boot provides an easy way to declaratively choose a set of specific starters that we need for our applications.

Let's take a look at what a Spring Boot application's `pom.xml` file might look like for a Maven project that needs to use the following Spring projects.

- Spring Data JPA
- Spring MVC

Here we would like to create a Spring Boot application that uses Spring Data's repository-based data management capabilities for a relational database. We'll also need to create a REST controller interface to expose an API for managing data over HTTP. We'll create a `pom.xml` that looks like [Example 1-1](#).

Example 1-1. A example Maven project's pom.xml for a Spring Boot application

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

    <modelVersion>4.0.0</modelVersion>

    <!-- Project Metadata -->
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

```

<packaging>jar</packaging>

<!-- Spring Boot Starter Parent BOM -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactI
    <version>1.3.2.RELEASE</version> ❶
    <relativePath/>
</parent>

...

<!-- Spring Boot Starter Dependencies -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupI
        <artifactId>spring-boot-starter-data-jpa<
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupI
        <artifactId>spring-boot-starter-web</arti
    </dependency>
</dependencies>

</project>

```

❶

The Spring Boot release version for all referenced starters

❷

The starter that includes *Spring Data JPA*

❸

The starter that includes *Spring MVC*

We can see in [Example 1-1](#) that we have our basic `pom.xml` that includes the starters we need for *Spring Data JPA* and *Spring MVC*. Notice that we do not need to provide a version for each of these dependencies. Because we are referencing a parent release for Spring Boot starters, one that has a specified version, all transitive dependencies for the Spring Boot starters we need will be automatically described in the `spring-boot-starter-parent`.

Getting Started with the Spring Initializr

Spring Initializr is an open source project and tool in the Spring ecosystem that helps you create ready-to-go project templates as Spring Boot applications. Spring Boot is an opinionated framework that reduces the resistance of on-boarding new applications by giving developers an easy way to compose the various modules provided by the Spring Framework project and other ecosystem projects, such as Spring Data, Spring Security, Spring Cloud, and others.

When building microservices, it is essential that the friction of on-boarding new applications is reduced to a minimum. If I am a lead developer who has been tasked with the creation of a new microservice, either by decomposing functionality from an existing monolith, or as a green field project, the amount of time spent to support the new application as a part of the existing architecture can be quite costly if a process is not in place to minimize it.

Let's assume that we are a developer tasked with creating a new microservice in an online banking application that will be decomposed from an existing service.

In [Figure 1-11](#) we can see what our current architecture looks like for the application. There are a few components here that we should go over. Each hexagon is a Spring Boot application that already exists as a part of our online banking application. We see that we have an *Online Banking Web* application to the left, in blue. This application will be the user interface for our online banking application and depends on a data service, which is called the *Customer Service*.

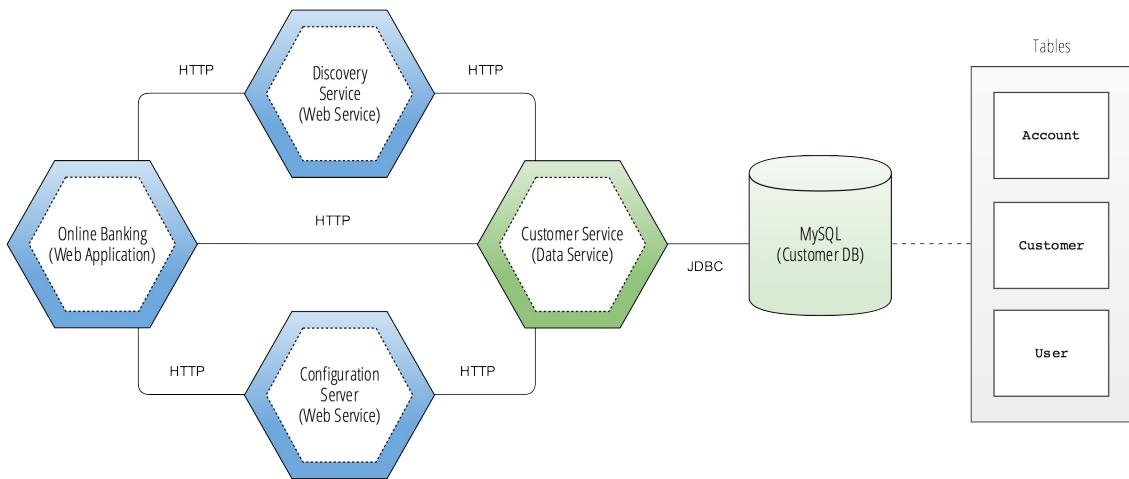


Figure 1-11. Spring Boot services for online banking

We also see in [Figure 1-11](#) that we have two other Spring Boot applications, the *Configuration Server* and the *Discovery Service*. These two services were recently added by the platform team in a push to start building cloud native applications. The platform services will help our new microservice communicate with other services and vice versa. We also will be able to source the configurations for the new microservice in the environment that it is deployed to. This means that we will not store configurations for different environments in the source code of the new service, they will instead be sourced from the environment.

The last component of the diagram is a MySQL database, named the *Customer DB*. We can see that the *Customer Service* has a dependency on the *Customer DB* and is using a JDBC connection. We also see that there are a set of tables that have been added to the *Customer DB*. We see in the list of tables we have: **Account**, **Customer**, and **User**.

Over time, the *Customer Service* has served as the full backend for our application. The development teams working on these different features are starting to collide and there is increased coordination that is delaying the frequency of releases. We can consider the *Customer Service* to be a monolithic application which needs to be split into separate services to help improve the release velocity for developers.

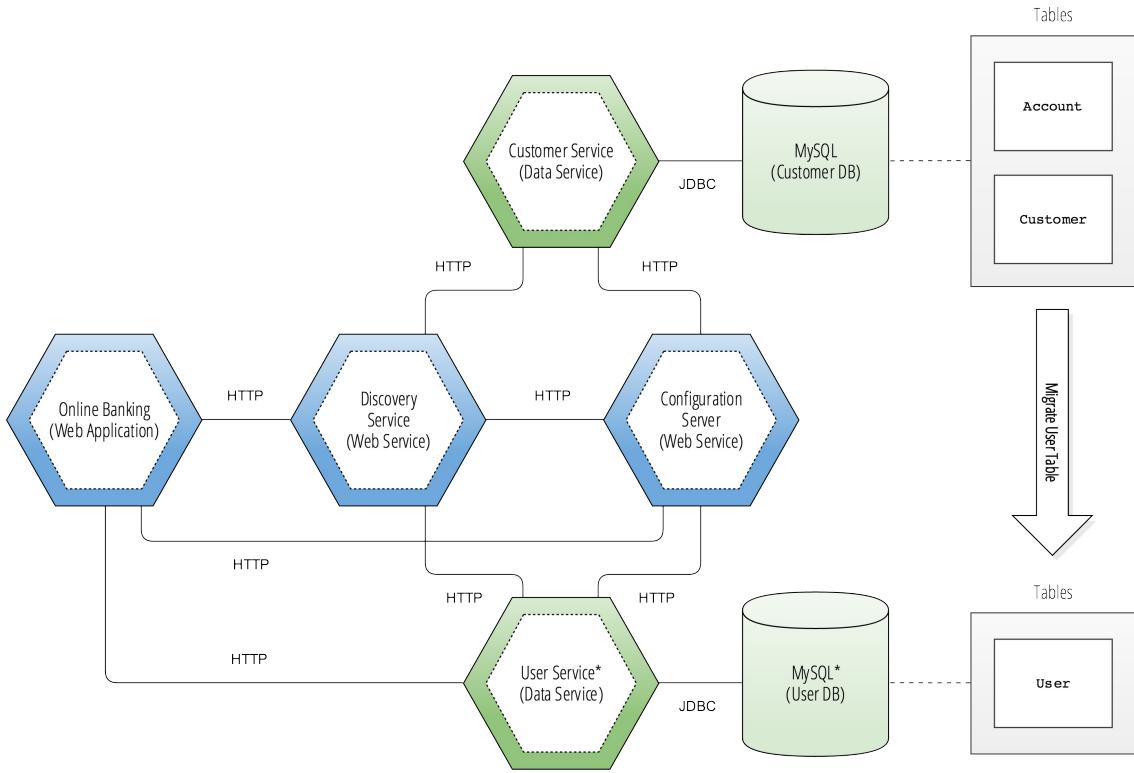


Figure 1-12. Decomposing the User table into a new microservice

In [Figure 1-12](#) we can see the updated architecture, which illustrates a new service, called the *User Service*, which has decomposed a part of the *Customer Service* into a separate application. When choosing which functionality to decompose into a new microservice, it's often a process of concerted analysis in order to make the right decision. We see here that the *User* table was migrated to a new *User DB* and out of the *Customer DB*. We can assume that features requiring changes to functionality that depend on the *User* table make it a good candidate to move into a new microservice.

Now let's go through the motions of using Spring Initializr to create a new project template for our new *User Service* from [Figure 1-12](#).

Generating Spring Boot applications

Let's take a deeper look at what dependencies from the Spring Boot starter projects we'll need to create the *User Service* from [Figure 1-12](#).

Table 1-1. Spring Boot Starters for the User Service

| Spring Project | Starter Projects |
|----------------------------|---------------------------------|
| Spring Data JPA | JPA |
| Spring Data REST | REST Repositories |
| Spring Security | Security |
| Spring Cloud Netflix | Eureka Discovery, Config Client |
| Spring Framework (MVC) Web | |
| H2 Embedded SQL DB | H2 |
| MySQL JDBC Driver | MySQL |

In [Table 1-1](#) we can see the starter projects that we will need for our new *User Service* application. We'll use Spring Initializr to generate this project.

The Spring team has an instance of the Spring Initializr hosted on Pivotal Web Services, and the website can be found at <http://start.spring.io>.

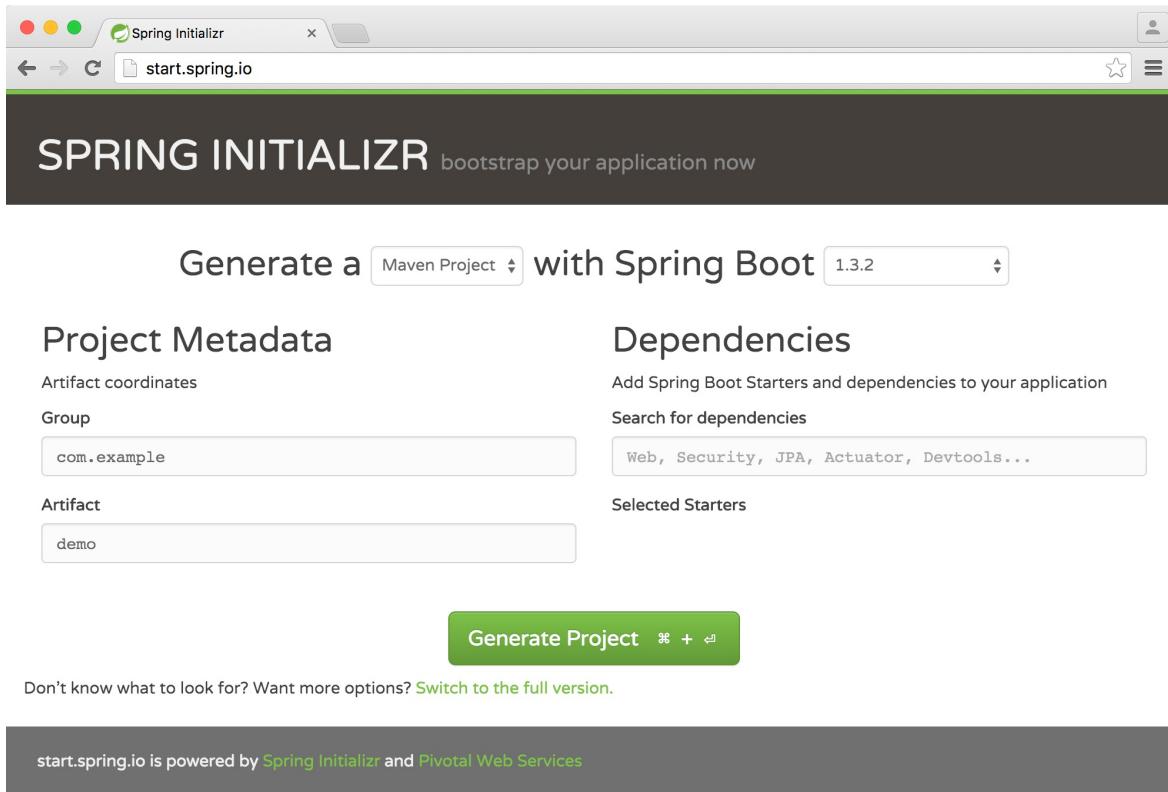


Figure 1-13. The Spring Initializr website

In [Figure 1-13](#) we can see the web interface of Spring Initializr. We can use this website to generate a working Spring Boot application configured with the starter projects that we would like to select. There are a wide variety of starter projects for Spring Boot, but we can use the *Dependencies* section to search for the starters we need from [Table 1-1](#). In the text box, we'll type each of the starter project names and add them to the list of our dependencies, as shown in [Figure 1-14](#).

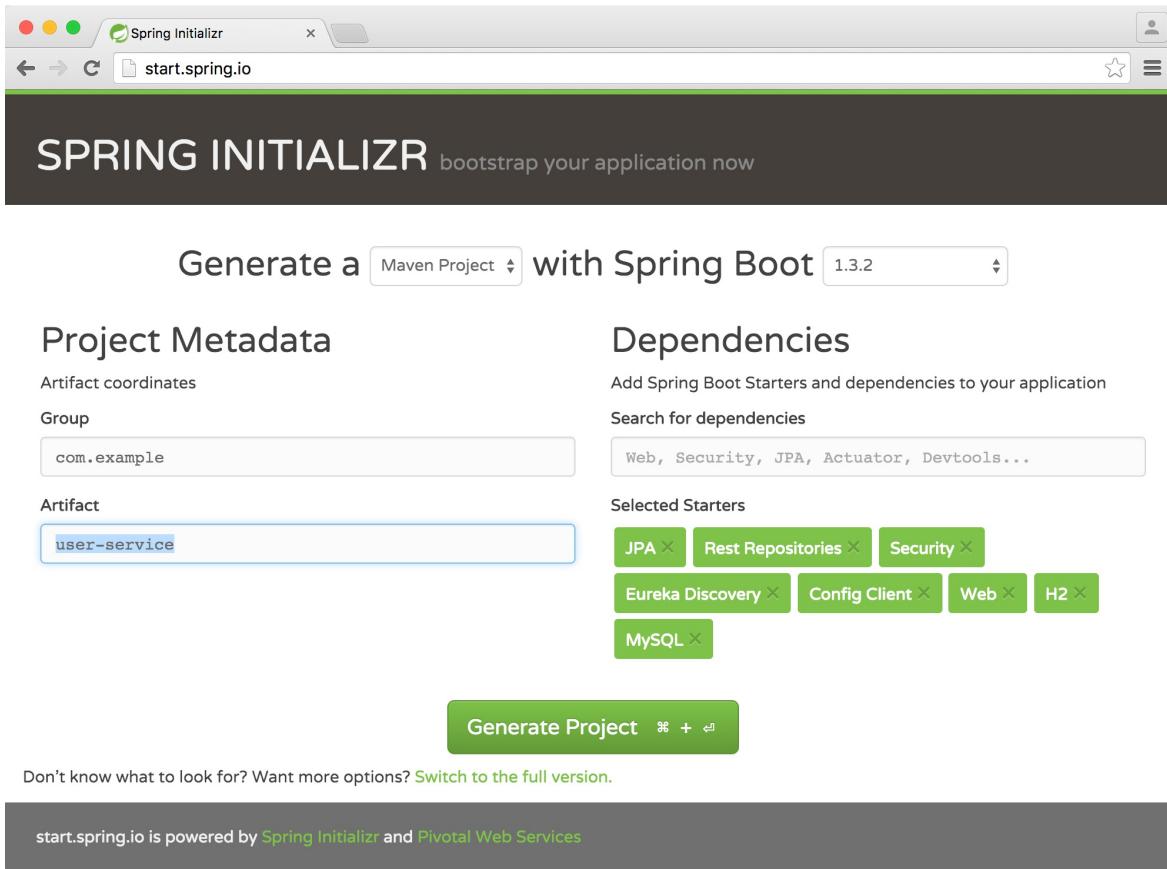


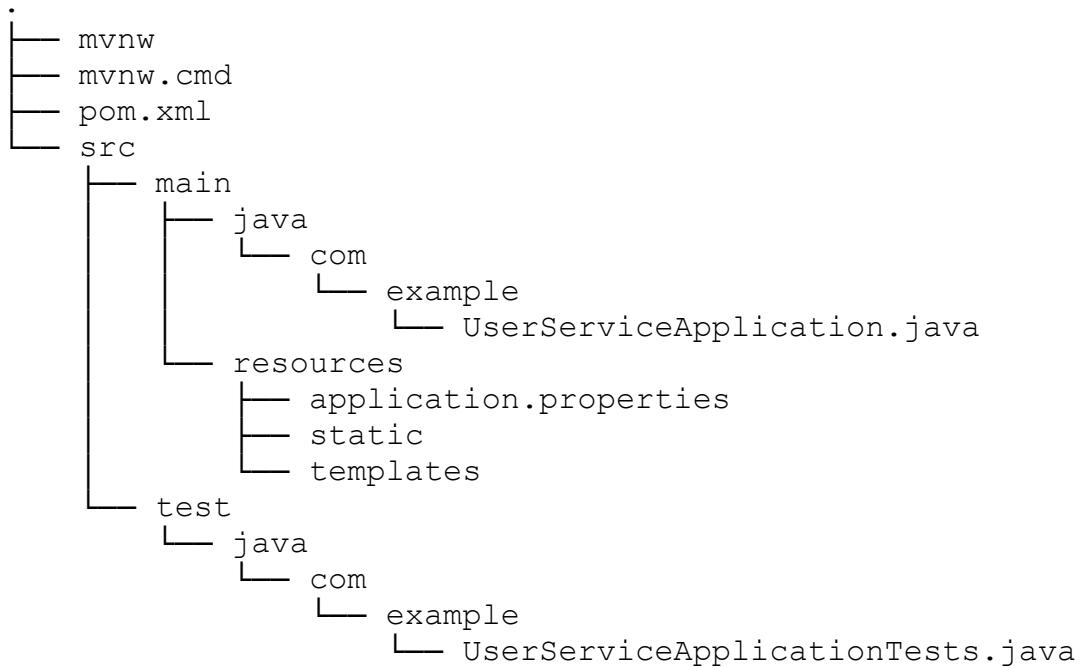
Figure 1-14. Selecting starter dependencies

We'll also be sure to select the type of project we would like, which can be either Maven or Gradle. We'll choose Maven for this application. Also, we'll need to choose the version of Spring Boot we would like to target, in this case we'll choose Spring Boot 1.3.2 release. Finally, we'll rename the artifact from *demo* to *user-service*.

Now we are ready to generate the project. We can go ahead and click the **Generate Project** button and our project will be downloaded as a compressed file named *user-service.zip*. After downloading the project, uncompress and extract the contents to a location on your machine.

Now let's take a look at the project structure of the Spring Boot application that was generated.

Example 1-2. Contents of the generated Spring Boot application



In [Example 1-2](#) we can see the directory structure of our generated *User Service* application. The Spring Initializr will provide a wrapper, either Maven or Gradle, as a part of the contents of the generated project. You can use this wrapper to build and run the project. For Maven, we can start up the application using the Maven wrapped command `./mvnw <command>` from the command line. The following command will run a clean installation of the Maven project, downloading and caching the dependencies from the `pom.xml` in the process.

```
$ ./mvnw clean install
```

To run the Spring Boot application from the command line, we can use the provided Spring Boot Maven plugin, which is referenced in the `pom.xml`. To use this plugin, we can run the following command.

```
$ ./mvnw spring-boot:run
```

After running the command, the Spring Boot application will start up and can be accessed as a web application from <http://localhost:8080>.

Now let's take a look at the `pom.xml` to see how our application dependencies have been automatically configured using Spring Initializr.

Example 1-3. The `pom.xml` dependencies of the User Service

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-rest</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- We can target an embedded H2 SQL DB for tests -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

In [Example 1-3](#) we can see each of the dependencies that were added as Spring Boot starter projects by the Spring Initializr. Each of these starter projects will help us go forward to customize the application as we refactor the functionality related to user management of the Customer Service into our new User Service.

If we take a closer look at these dependencies, we can see that there are two database libraries which are not starter projects: `h2` and `mysql-connector-java`. Later in the book, in [Chapter 6](#), we'll explore how to use two different data sources to run our unit tests against an embedded SQL database, while still using an external MySQL JDBC connection when running the application from a different profile.

Now that we've explored how to generate Spring Boot applications using starter projects with Spring Initializr, let's explore a useful resource from Spring that will help us understand how to assemble functionality from the starter projects in our application. For this we can use a set of sample projects and tutorials that are a part of the *Spring Guides* project.

The Spring Guides

When getting started with Spring Boot and the many projects in the Spring ecosystem, it can be very useful to see working samples and tutorials to help you understand best practices and how to put the tools to use for your own use cases. For this purpose, the Spring community has a comprehensive set of contributed guides. The *Spring Guides* are a collection of sample applications that are focused on demonstrating a specific feature of a Spring project. These guides are mostly contributed from the engineers on the Spring team, but are also contributed by active members of the open source community who are focused on Spring.

Each Spring guide takes the same approach to providing a comprehensive yet consumable guide that you should be able to get through within 15-30 minutes. These guides are one of the most useful resources when getting started with Spring Boot.

To get started with the *Spring Guides*, head over to <https://spring.io/guides>.

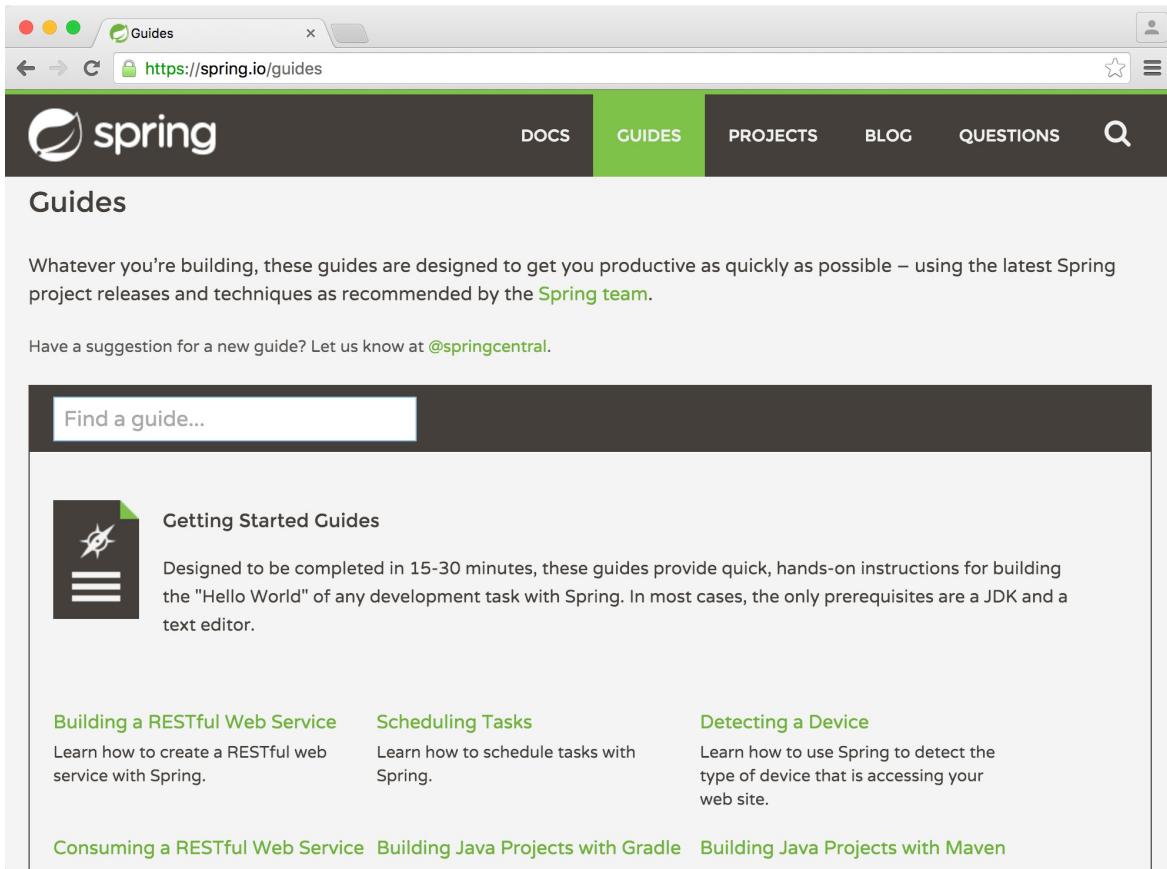


Figure 1-15. The Spring Guides website

As we can see from [Figure 1-15](#), the *Spring Guides* website provides a collection of maintained examples that target a specific use case. Let's explore guides related to Spring Boot.

The screenshot shows a search interface with a dark header bar containing the text "spring boot". Below the header, there is a section titled "Getting Started Guides" featuring a small icon of a rocket ship launching from a stack of three horizontal bars. The main content area displays several guide cards:

- Building a RESTful Web Service with Spring Boot Actuator**
Learn how to create a RESTful Web service with Spring Boot Actuator.
- Converting a Spring Boot JAR Application to a WAR**
Learn how to convert your Spring Boot JAR-based application to a WAR file.
- Building an Application with Spring Boot**
Learn how to build an application with minimal configuration.
- Application development with Spring Boot + JS**
See how to rapidly create rich apps with Spring Boot CLI + Javascript
- Spring Boot with Docker**
Learn how to create a Docker container from a Spring Boot application with Maven or Gradle

Figure 1-16. Exploring the Spring Guides

In [Figure 1-16](#) I've entered the search term *spring boot*, which will narrow down the list of guides to those that focus on Spring Boot.

As a part of each *Spring Guide*, we'll find the following familiar features.

- Getting started
- Table of contents
- What you'll build
- What you'll need
- How to complete this guide
- Walkthrough
- Summary
- Get the code

- Link to GitHub repository

Let's now choose one of the basic Spring Boot guides: [Building an Application with Spring Boot](#).

The screenshot shows a Spring guide page with the following structure:

- Header:** GETTING STARTED, build passing
- Title:** Building an Application with Spring Boot
- Text:** This guide provides a sampling of how Spring Boot helps you accelerate and facilitate application development. As you read more Spring Getting Started guides, you will see more use cases for Spring Boot. It is meant to give you a quick taste of Spring Boot. If you want to create your own Spring Boot-based project, visit [Spring Initializr](#), fill in your project details, pick your options, and you can download either a Maven build file, or a bundled up project as a zip file.
- Section:** What you'll build
- Text:** You'll build a simple web application with Spring Boot and add some useful services to it.
- Section:** What you'll need
- List:**
 - About 15 minutes
 - A favorite text editor or IDE
 - JDK 1.8 or later
 - Gradle 2.3+ or Maven 3.0+
 - You can also import the code from this guide as well as view the web page directly into [Spring Tool Suite \(STS\)](#) and work your way through it from there.
- Section:** How to complete this guide
- Text:** Like most Spring Getting Started guides, you can start from scratch and complete each step, or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.
- Right sidebar:**
 - Get the Code:** HTTPS, SSH, Subversion. A link to <https://github.com/spring-guides>. A large "DOWNLOAD ZIP" button.
 - Table of contents:**
 - What you'll build
 - What you'll need
 - How to complete this guide
 - Build with Gradle
 - Build with Maven
 - Build with your IDE
 - Learn what you can do with Spring Boot

Figure 1-17. Building an Application with Spring Boot guide

In [Figure 1-17](#) we see the basic anatomy of a Spring Guide. Each guide is structured in a familiar way to help you move through the content as effectively as possible. At the top of each guide, we'll see a brief introduction of the problem that the guide solves. You'll also be presented with a set of requirements for the guide, such as the length of the guide, the tools used. To the right of the page, you'll find a section to download the sample as a compressed ZIP file. Similar to the Spring Initializr, you'll be provided with a working Spring Boot application that is the subject of the guide. You'll also find a table of contents, just under the download section, which will help you navigate the contents of the guide.

Tip

Throughout this book you'll find situations or scenarios that you may need an expanded companion guide to help you get a better understanding of the

content. It's recommended in this case that you take a look at the *Spring Guides* and find a guide that best suits your needs.

Auto-Configuration

When using Spring Boot you'll often find that dependencies that you've added from the list of starter projects are automatically configured as Spring beans. Spring Boot uses a technique called *Auto-Configuration* to bootstrap dependencies using default configuration properties. When adding a dependency to the classpath of a Spring application, the dependency may use auto-configuration by default, and will be initialized using default properties if you have not chosen to configure it manually.

Earlier in this chapter we generated a *User Service* from Spring Initializr. In the generated project we chose two database libraries, which were added to the classpath of the application.

Example 1-4. Dependency section of a `pom.xml` with two database libraries

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

The snippet from [Example 1-4](#) describes a Maven `pom.xml` that contains dependency references for the Spring Data JPA starter project as well as two database libraries: `h2` and `mysql-connector-java`. We have not specified a specific configuration for a `DataSource`, so Spring Boot will auto-configure a `javax.sql.DataSource` based on the embedded H2 database. It does this because `spring-boot-starter-data-jpa` includes a transitive dependency

for `spring-jdbc` which in turn activates certain conditions in Spring Boot's `org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`. This auto-configuration contains nested auto-configuration, `JdbcTemplateConfiguration`.

Example 1-5. `JdbcTemplateConfiguration` class

```
@Configuration
@ConditionalOnMissingBean(DataSourceAutoConfiguration.class)
protected static class JdbcTemplateConfiguration {

    @Autowired(required = false)
    private DataSource dataSource;

    @Bean
    @ConditionalOnMissingBean(JdbcOperations.class) ②
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(this.dataSource);
    }

    @Bean
    @ConditionalOnMissingBean(NamedParameterJdbcOperations.class)
    public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {
        return new NamedParameterJdbcTemplate(this.dataSource);
    }
}
```

①

The `@Conditional` annotation scans for a `DataSourceAvailableCondition` class

②

The `@ConditionalOnMissingBean` annotation initializes the `JdbcTemplate` bean if absent

In [Example 1-5](#) we see the contents of the `JdbcTemplateConfiguration` class, which will be used to automatically configure `JdbcTemplate` using the configured `DataSource` bean. As a part of the parent `DataSourceAutoConfiguration` class, the `DataSource` bean is configured in much the same way. Depending on whether or not a dependency on the

classpath is configurable as an embedded database, which in this case h2 is supported, the DataSource bean will be initialized to connect to the embedded H2 database.

Auto-configuration in Spring Boot will scan the classpath and make a best guess at what the intention is by the developer for automatically configuring dependencies. For example, I've specified two database connectors, one for MySQL and one for H2. The h2 library will take precedence over the mysql-connector-java library in the case that the configuration properties for JPA does not specify a database. If we change the configuration property in the application.yml to use the MySQL driver and specify a JDBC connection URL, the embedded H2 database will not be configured.

Example 1-6. The application.yml can override the targeted datasource driver from H2 to MySQL

```
spring:  
  datasource:  
    driverClassName: com.mysql.jdbc.Driver ❶  
    url: jdbc:mysql://localhost/test ❷
```

❶

Overrides the DataSource auto-configuration to use a non-embedded JDBC connection

❷

We must specify a JDBC URL to override the auto-configured H2 connection

In [Example 1-6](#) we are choosing to override the default auto-configuration properties that will be configured for the embedded H2 database, because it takes precedence over the MySQL dependency for auto-configuration.

If we only configured the value of the

`spring.datasource.driverClassName` configuration property, we would see an exception after starting the Spring Boot application. The exception would display an error message due to a failed connection to an embedded H2

database, in this case it would look like [Example 1-7](#).

Example 1-7. Exception connecting to an embedded H2 database with a MySQL driver

```
java.sql.SQLException: Driver:com.mysql.jdbc.Driver@27cc341d retu
    null for URL:jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOS
```

Spring Boot Configuration

Now that we've looked at some of the basics of building and running Spring Boot applications, let's now explore how to configure Spring Boot for cloud-native applications. As we talked about earlier on in [Chapter 2](#), twelve-factor applications use a specific pattern for developing applications that are designed to be easily deployed to a cloud platform. As a part of this practice is to separate environment configurations from your source code and to instead store them in the environment. When we deploy our Spring Boot applications to different environments, they should be deployed as a single identical artifact in each environment. When the Spring Boot application starts up in the environment, it should be able to source its configurations from that environment.

Chapter 2. The Cloud Native Application

The patterns for how we develop software, both in teams and as individuals, are always evolving. The open source software movement has provided the software industry with somewhat of a Cambrian explosion of tools, frameworks, platforms, and operating systems—all with an increasing focus on flexibility and automation. A majority of today’s most popular open source tools focus on features that give software teams the ability to continuously deliver software faster than ever before possible.

In the span of two decades, starting in the early 90s, an online bookstore headquartered in Seattle, called Amazon.com, grew into the world’s largest online retailer. Known today simply as *Amazon*, the company now sells far more than just books. In 2015, Amazon surpassed Walmart as the most valuable retailer in the United States. The most interesting part of Amazon’s story of unparalleled growth can be summarized in one simple question: How did a website that started out as a simple online bookstore transform into one of the largest retailers in the world—doing so without ever opening a single retail location?

It’s not hard to see how the world of commerce has been shifted and reshaped around digital connectivity, catalyzed by ever increasing access to the internet from every corner of the globe. As personal computers became smaller, morphing into the ubiquitous smart phone and tablets we use today, we’ve experienced an exponential increase in accessibility to distribution channels that are transforming the way the world does commerce.

Amazon’s CTO, Werner Vogels, oversaw the technical evolution of Amazon from a hugely successful online bookstore into one of the world’s most valuable technology companies and product retailers. In June 2006, Vogels was interviewed in a piece for the computer magazine *ACM Queue* on the rapid evolution of the technology choices that powered the growth of the online retailer. In the interview Vogels talks about the core driver behind the

company's continued growth.

A large part of Amazon.com's technology evolution has been driven to enable this continuing growth, **to be ultra-scalable while maintaining availability and performance.**

Werner Vogels, ACM Queue, Volume 4 Issue 4, A Conversation with Werner Vogels

Vogels goes on to state that in order for Amazon to achieve ultra-scalability it needed to move towards a different pattern of software architecture. Vogels mentions in the interview that Amazon.com started as a monolithic application. Over time, as more and more teams operated on the same application, the boundaries of ownership of the codebase began to blur. "There was no isolation and, as a result, no clear ownership." said Vogels.

Vogels went on to pinpoint that shared resources, such as databases, were making it difficult to scale-out the overall business. The greater the number of shared resources, whether it be application servers or databases, the less control teams would have when delivering features into production.

You build it, you run it.

Werner Vogels, CTO, Amazon

Vogels touched on a common theme that mostly all cloud native applications share, the idea of ownership of what you are building. He goes on to say that "the traditional model is that you take your software to the wall that separates development and operations, and throw it over and then forget about it. Not at Amazon. *You build it, you run it.*"

In what has been one of the most reused quotes by prominent keynote speakers at some of the world's premier software conferences, the words "*you build it, you run it*" would later become a slogan of a popular movement we know today simply as *DevOps*.

Many of the practices that Vogels spoke about in 2006 were seeds for popular software movements that are thriving today. Practices such as *DevOps* and

microservices can be tied back to the ideas that Vogels introduced over a decade ago. While ideas like these were being developed at large internet companies similar to Amazon, the tooling around these ideas would take years to develop and mature into a service offering.

In 2006 Amazon launched a new product named Amazon Web Services (AWS). The idea behind AWS was to provide a platform, the same platform that Amazon used internally, and release it as a service to the public. Amazon was keen to see the opportunity to commodotize the ideas and tooling behind the Amazon.com platform. Many of the ideas that Vogels introduced were already built into the Amazon.com platform. By releasing the platform as a service to the public, Amazon would enter into a new market called the *public cloud*.

The ideas behind the public cloud were sound. Virtual resources could be provisioned on-demand without needing to worry about the underlying infrastructure. One could simply rent a virtual machine to house their applications without needing to purchase or manage the infrastructure. This approach was a low-risk self-service option that would help to grow the appeal of the public cloud, with AWS leading the way in terms of adoption.

It would take years before AWS would mature into a set of services and patterns for building and running applications that are designed to be operated on a public cloud. While many developers flocked to these services for building new applications, many companies with existing applications still had concerns with migrations. Existing applications were not designed for portability. Also, many applications were still dependent on legacy workloads that were not compatible with the public cloud.

In order for most large companies to take advantage of the public cloud, they would need to make changes in the way they developed their applications.

Platforms

Platform is an overused word today.

When we talk about platforms in computing, we are generally talking about a set of capabilities that help us to either build or run applications. Platforms are best summarized by the nature in which they impose constraints on how developers build applications.

For example, when we build Java applications, we are building applications *on the* Java platform. The Java platform, in turn, is a set of components that are essential for staging and running your Java applications. The platform takes care of translating Java source code into Java bytecode, and provides a runtime environment, the JVM, for executing that bytecode.

As is the case with Java, platforms may be written on top of other platforms. The JVM is considered to be *cross-platform*, designed to run on separate operating platforms.

Let's consider another example. You are a lead developer on a new software project and have been asked to provide a sizing estimate on the effort necessary to build a new application. You begin to carefully break down a list of business requirements into a technical design. In the process, you will need to make many technology choices. Now let's assume that many of these technology choices have already been made, decided upon by a team of software architects at your company.

Let's assume that the architecture team regularly meets to maintain and revise a list of approved technologies for building applications. Now because of this, a set of prescribed standards will act to constrain the technology choices you are able to make during your sizing estimate. These constraints will greatly reduce the time spent evaluating options at each decision point. This makes it far simpler to size the effort needed to implement just the business logic of the application. Platforms in the same way are able to impose constraints in order to prevent undifferentiated heavy lifting required to build

and operate applications.

Platforms are able to automate the tasks that are not essential to supporting the business requirements of an application. In turn, this makes development teams more agile in the way they are able to support only the features that help to differentiate value for the business.

Building Platforms

When we build platforms, we are creating a tool that automates a set of repeatable practices. Practices are formulated from a set of constraints that translate valuable ideas into a plan. These constraints take the form of opinions on how valuable ideas can be executed into a repeatable practice, automated by a platform.

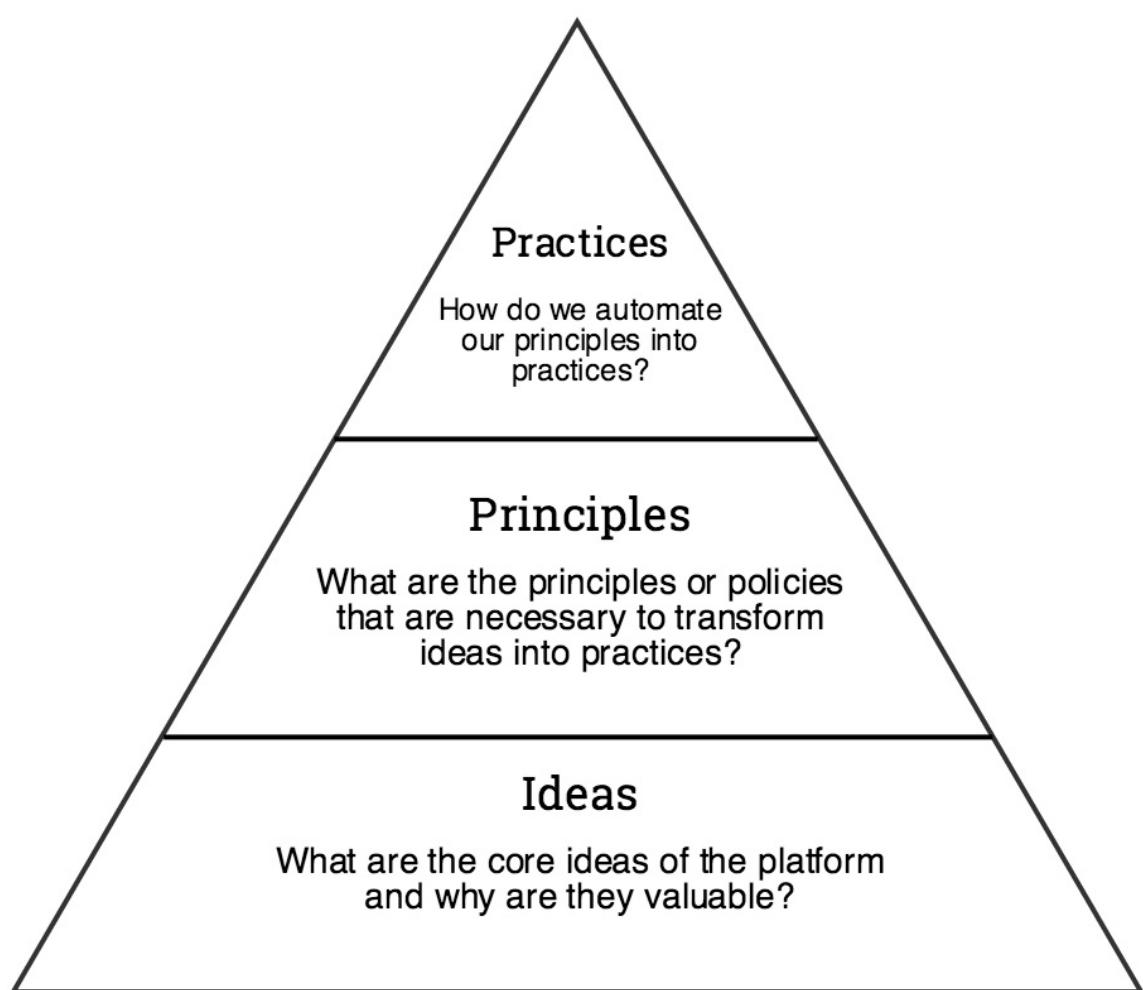


Figure 2-1. The hierarchy of concerns for building platforms

- Ideas

- What are our core ideas of the platform and why are they valuable?
- **Constraints**
 - What are the constraints necessary to transform our ideas into practices?
- **Practices**
 - How do we automate constraints into a set of repeatable practices?

At the core of every platform are simple ideas, that when realized, increase differentiated business value through the use of an automated tool.

Let's take for example the Amazon.com platform. Werner Vogels stated that by increasing isolation between software components, teams would have more control over features they delivered into production.

Idea:

- *By increasing isolation between software components, we are able to deliver parts of the system both rapidly and independently.*

By using this idea as the platform's foundation, we are able to fashion it into a set of constraints. Constraints take the form of an opinion about how a core ideal will create value when automated into a practice. The following statements are opinionated constraints about how isolation of components can be increased.

Constraints:

- *Software components are to be built as independently deployable services.*
- *All business logic in a service is encapsulated with the data it operates on.*
- *There is no direct access to a database from outside of a service.*

- *Services are to publish a web interface that allows access to its business logic from other services.*

With these constraints, we have taken an opinionated view on how isolation of software components will be increased in practice. The promises of these constraints, when automated into practices, will provide teams with more control over how features are delivered to production. The next step is to describe how these constraints can be captured into a set of repeatable practices.

Practices that are derived from these constraints should be stated as a collection of promises. By stating practices as promises we maintain an expectation with the users of the platform on how they will build and operate their applications.

Practices:

- *A self-service interface is provided to teams that allows for the provisioning of infrastructure required to operate applications.*
- *Applications are packaged as a bundled artifact and deployed to an environment using the self-service interface.*
- *Databases are provided to applications in the form of a service, and are to be provisioned using the self-service interface.*
- *An application is provided with credentials to a database as environment variables, only after declaring an explicit relationship to the database as a service binding.*
- *Each application is provided with a service registry that is used as a manifest of where to locate external service dependencies.*

Each of the practices listed above takes on the form of a promise to the user. In this way, the intent of the ideas at the core of the platform are realized as constraints imposed on applications.

Cloud native applications are built on a set of constraints that help to reduce

the time spent performing undifferentiated heavy lifting.

When AWS was first released to the public, Amazon did not force its users to adhere to the same constraints that they used internally for Amazon.com. Staying true to the name, *Amazon Web Services*, AWS is not itself a cloud *platform*, but rather it is a collection of independent infrastructure services that can be composed into automated tooling resembling a platform of promises. Years after the first release of AWS, Amazon would begin to offer a collection of managed platform services, with use cases ranging from IoT (Internet of Things) to machine learning.

If every company needs to build their own platform from scratch, the amount of time delivering value in applications is delayed until the platform is fully assembled.

Companies who were early adopters of AWS would have needed to assemble together some form of automation resembling a platform. Each company would have had to bake-in a set of promises that captured the core ideas of how to develop and deliver software into production.

More recently, the software industry has converged on the idea that there are a basic set of common promises that every cloud platform should make. These promises will be explored throughout this book using the open source *Platform-as-a-Service* (PaaS), named Cloud Foundry. The core idea behind Cloud Foundry is to provide a platform that encapsulates a set of common promises for building and operating highly available and fault tolerant applications. Cloud Foundry makes these promises while still providing portability between multiple different cloud infrastructure providers.

The subject of much of this book is how to build cloud native Java applications. We'll focus largely on tools and frameworks that help to reduce undifferentiated heavy lifting, by taking advantage of the benefits and promises of a cloud native platform.

The Patterns

New patterns for how we develop software are enabling us to think more about the behavior of our applications in production. Both developers and operators, together, are placing more emphasis on understanding how their applications will behave in production, with fewer assurances of how complexity will unravel in the event of a failure.

As was the case with Amazon.com, software architectures are beginning to move away from large monolithic applications. Architectures are now focused on achieving ultra-scalability without sacrificing performance and availability. By breaking apart components of a monolith, engineering organizations are taking efforts to decentralize change management, providing teams with more control over how features make their way to production. By increasing isolation between components, software teams are starting to enter into the world of distributed systems development, with a focus of building smaller more singularly focused services with independent release cycles.

Cloud native applications take advantage of a set of patterns that make teams more agile in the way they deliver features to production. As applications become more distributed, a result of increasing isolation necessary to provide more control to the teams that own applications, the chance of failure in the way application components communicate becomes an important concern. As software applications turn into complex distributed systems, operational failures become an inevitable result.

Cloud native application architectures provide the benefit of ultra-scalability while still maintaining guarantees about overall availability and performance of applications. While companies like Amazon reaped the benefits of ultra-scalability in the cloud, widely available tooling for building cloud-native applications had yet to surface. The tooling and platform would eventually surface as a collection of open source projects maintained by an early pioneer of the public cloud, a company named Netflix.

Netflix's Story

Today, Netflix is one of the world's largest on-demand streaming media services, operating their online services in the cloud. Netflix was founded in 1997 in Scotts Valley, California by Reed Hastings and Marc Randolph. Originally, Netflix provided an online DVD rental service that would allow customers to pay a flat-fee subscription each month for unlimited movie rentals without late fees. Customers would be shipped DVDs by mail after selecting from a list of movie titles and placing them into a queue using the Netflix website.

In 2008, Netflix had experienced a major database corruption that prevented the company from shipping any DVDs to its customers. At the time, Netflix was just starting to deploy its streaming video services to customers. The streaming team at Netflix realized that a similar kind of outage in streaming would be devastating to the future of its business. Netflix made a critical decision as a result of the database corruption, that they would move to a different way of developing and operating their software, one that ensured that their services would always be available to their customers.

As a part of Netflix's decision to prevent failures in their online services, they decided that they must move away from vertically scaled infrastructure and single points of failure. The realization stemmed from a result of the database corruption, which was a result of using a vertically scaled relational database. Netflix would eventually migrate their customer data to a distributed NoSQL database, an open source database project named Apache Cassandra. This was the beginning of the move to become a "cloud native" company, a decision to run all of their software applications as highly distributed and resilient services in the cloud. They settled on increasing the robustness of their online services by adding redundancy to their applications and databases in a scale out infrastructure model.

As a part of Netflix's decision to move to the cloud, they would need to migrate their large application deployments to highly reliable distributed systems. They faced a major challenge. The teams at Netflix would have to

re-architect their applications while moving away from an on-premise data center to a public cloud. In 2009, Netflix would begin its move to Amazon Web Services (AWS), and they focused on three main goals: scalability, performance, and availability.

By the start of 2009, the subscriptions to Netflix's streaming services had increased by nearly 100 times. Yuri Izrailevsky, VP Cloud Platform at Netflix, gave a presentation in 2013 at the AWS reinvent conference. "We would not be able to scale our services using an on-premise solution," said Izrailevsky.

Furthermore, Izrailevsky stated that the benefits of scalability in the cloud became more evident when looking at its rapid global expansion. "In order to give our European customers a better low-latency experience, we launched a second cloud region in Ireland. Spinning up a new data center in a different territory would take many months and millions of dollars. It would be a huge investment." said Izrailevsky.

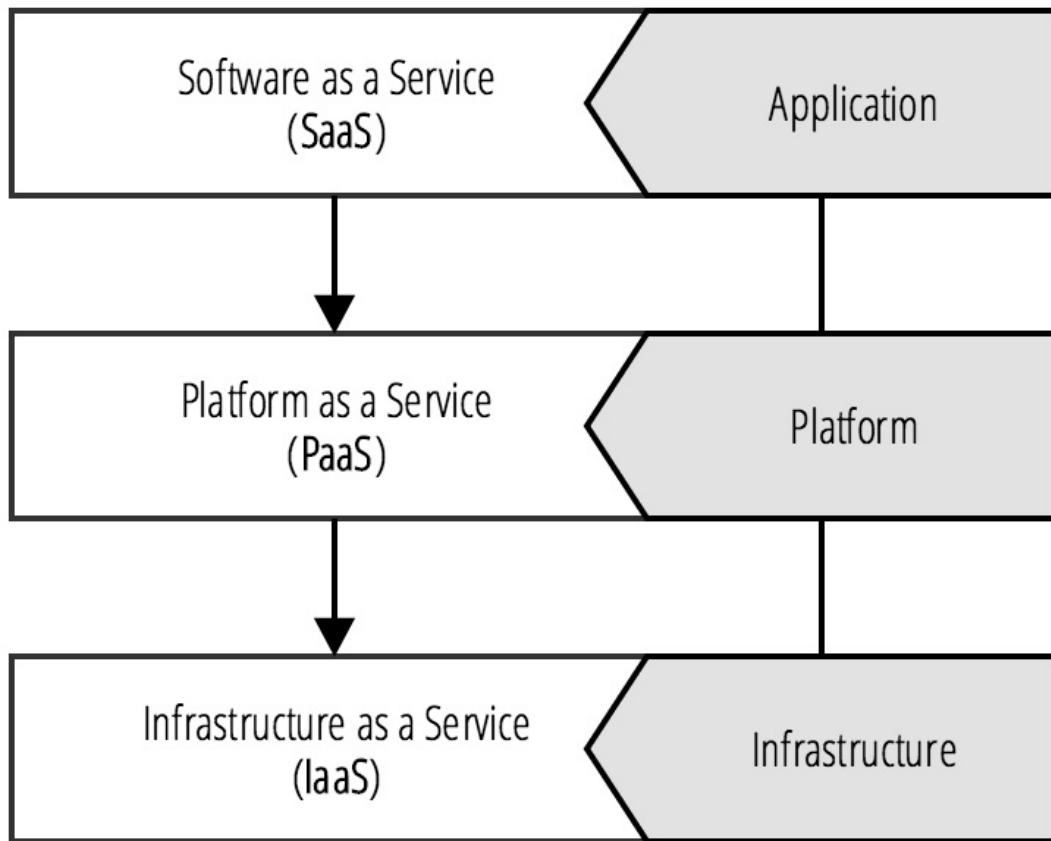
As Netflix began its move to hosting their applications on Amazon Web Services, employees of the company would chronicle their learnings on Netflix's company blog. Many of Netflix's employees were advocating a move to a new kind of architecture that focused on horizontal scalability at all layers of the software stack.

John Ciancutti, who was then the Vice President of Personalization Technologies at Netflix, said on the company's blog in late 2010 that, "cloud environments are ideal for horizontally scaling architectures. We don't have to guess months ahead what our hardware, storage, and networking needs are going to be. We can programmatically access more of these resources from shared pools within Amazon Web Services almost instantly."

What Ciancutti meant by being able to "programmatically access" resources, was that developers and operators could programmatically access certain management APIs that are exposed by Amazon Web Services in order to give customers a controller for provisioning their virtualized computing infrastructure. The interface for these APIs are in the form of RESTful web services, and they give developers a way to build applications that manage and provision virtual infrastructure for their applications.

Note

Providing management services to control virtualized computing infrastructure is one of the primary concepts of cloud computing, called *Infrastructure as a Service*, commonly referred to as IaaS.



Ciancutti admitted in the same blog post that Netflix was not very good at predicting customer growth or device engagement. This is a central theme behind cloud native companies. Cloud native is a mindset that admits to not being able to reliably predict when and where capacity will be needed.

In Yuri Izrailevsky's presentation at the 2013 AWS reinvent conference, he said that "in the cloud, we can spin up capacity in just a few days as traffic eventually increases. We could go with a tiny footprint in the beginning and gradually spin it up as our traffic increases."

Izrailevsky goes on to say “As we become a global company, we have the benefit of relying on multiple Amazon Web Services regions throughout the world to give our customers a great interactive experience no matter where they are.”

The economies of scale that benefited Amazon Web Services’s international expansion also benefited Netflix. With AWS expanding availability zones to regions outside of the United States, Netflix expanded its services globally using only the management APIs provided by AWS.

Izrailevsky quoted a general argument of cloud adoption by enterprise IT, “Sure, the cloud is great, but it’s too expensive for us.” His response to this argument is that “as a result of Netflix’s move to the cloud, the cost of operations has decreased by 87%. We’re paying 1/8th of what we used to pay in the data center.”

Izrailevsky explained further why the cloud provided such large cost savings to Netflix. “It’s really helpful to be able to grow without worrying about capacity buffers. We can scale to demand as we grow.”

Splitting the Monolith

There are two cited major benefits by Netflix of moving to a distributed systems architecture in the cloud from a monolith: agility and reliability.

Netflix's architecture before going cloud native comprised of a single monolithic JVM application. While there were multiple advantages of having one large application deployment, the major drawback was that development teams were slowed down due to needing to coordinate their changes.

When building and operating software, increased centralization decreases the risk of a failure at an increased cost of needing to coordinate. Coordination takes time. The more centralized a software architecture is, the more time it will take to coordinate changes to any one piece of it.

Monoliths also tend not to be very reliable. When components share resources on the same virtual machine, a failure in one component can spread to others, causing downtime for users. The risk of making a breaking change in a monolith increases with the amount of effort by teams needing to coordinate their changes. The more changes that occur during a single release cycle also increase the risk of a breaking change that will cause downtime. By splitting up a monolith into smaller more singularly focused services, deployments can be made with smaller batch sizes on a team's independent release cycle.

Netflix not only needed to transform the way they build and operate their software, they needed to transform the culture of their organization. Netflix moved to a new operational model, called DevOps. In this new operational model each team would become a product group, moving away from the traditional project group structure. In a product group, teams were composed vertically, embedding operations and product management into each team. Product teams would have everything they needed to build and operate their software.

Netflix OSS

As Netflix transitioned to become a cloud native company, they also started to participate actively in open source. In late 2010, Kevin McEntee, then the VP of Systems & Ecommerce Engineering at Netflix, announced in a blog post about the company's future role in open source.

McEntee stated that “the great thing about a good open source project that solves a shared challenge is that it develops its own momentum and it is sustained for a long time by a virtuous cycle of continuous improvement.”

In the years that followed this announcement, Netflix open sourced over 50 of their internal projects, each of which would become a part of the *Netflix OSS* brand.

Key employees at Netflix would later clarify on the company's aspirations to open source many of their internal tools. In July 2012, Ruslan Meshenberg, Netflix's Director of Cloud Platform Engineering, published a post on the company's technology blog. The blog post, titled *Open Source at Netflix*, explained why Netflix was taking such a bold move to open source so much of its internal tooling.

Meshenberg wrote in the blog post, on the reasoning behind its open source aspirations, that “Netflix was an early cloud adopter, moving all of our streaming services to run on top of AWS infrastructure. We paid the pioneer tax – by encountering and working through many issues, corner cases and limitations.”

The cultural motivations at Netflix to contribute back to the open source community and technology ecosystem are seen to be strongly tied to the principles behind the microeconomics concept known as *Economies of Scale*. Meshenberg then continues, stating that “We've captured the patterns that work in our platform components and automation tools. We benefit from the scale effects of other AWS users adopting similar patterns, and will continue working with the community to develop the ecosystem.”

In the advent of what has been referred to as the *era of the cloud*, we have seen that its pioneers are not technology companies such as IBM or Microsoft, but rather they are companies that were born on the back of the internet. Netflix and Amazon are both businesses who started in the late 90s as dot-com companies. Both companies started out by offering online services that aimed to compete with their *brick and mortar* counterparts.

Both Netflix and Amazon would in time surpass the valuation of their *brick and mortar* counterparts. As Amazon had entered itself into the cloud computing market, it did so by turning its collective experience and internal tooling into a set of services. Netflix would then do the same on the back of the services of Amazon. Along the way, Netflix open sourced both its experiences and tooling, transforming themselves into a cloud native company built on virtualized infrastructure services provided by AWS by Amazon. This is how the economies of scale are powering forward a revolution in the cloud computing industry.

In early 2015, on reports of Netflix's first quarterly earnings, the company was reported to be valued at \$32.9 billion. As a result of this new valuation for Netflix, the company's value had surpassed the value of the CBS network for the first time.

The Twelve Factors

The twelve-factor methodology is a popular set of application development principles compiled by the creators of the Heroku cloud platform. The *Twelve Factor App* is a website that was originally created by Adam Wiggins, a co-founder of Heroku, as a manifesto that describes *Software-as-a-Service* applications that are designed to take advantage of the common practices of modern cloud platforms.

On the website, which is located at <http://12factor.net>, the methodology starts out by describing a set of core foundational ideas for building applications.

Example 2-1. Core ideas of the 12-factor application

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a clean contract with the underlying operating system, offering **maximum portability** between execution environments;
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
- And can **scale up** without significant changes to tooling, architecture, or development practices.

Earlier on in the chapter we talked about the promises that platforms make to its users who are building applications. In [Example 2-1](#) we have a set of ideas that explicitly state the value proposition of building applications that follow the twelve-factor methodology. These ideas break down further into a set of constraints—the twelve individual factors that distill these core ideas into a collection of opinions for how applications should be built.

Example 2-2. The practices of a twelve-factor application

- **Codebase** — One codebase tracked in revision control, many deploys
- **Dependencies** — Explicitly declare and isolate dependencies
- **Config** — Store config in the environment
- **Backing services** — Treat backing services as attached resources
- **Build, release, run** — Strictly separate build and run stages
- **Processes** — Execute the app as one or more stateless processes
- **Port binding** — Export services via port binding
- **Concurrency** — Scale out via the process model
- **Disposability** — Maximize robustness with fast startup and graceful shutdown
- **Dev/prod parity** — Keep development, staging, and production as similar as possible
- **Logs** — Treat logs as event streams
- **Admin processes** — Run admin/management tasks as one-off processes

The twelve factors, each listed in [Example 2-2](#), describe constraints that help to build applications that take advantage of the ideas in [Example 2-1](#). The twelve factors are a basic set of constraints that can be used to build cloud native applications. Since the factors cover a wide range of concerns that are common practices in all modern cloud platforms, building 12-factor apps are a common starting point in cloud native application development.

Outside of the 12-factor website—which covers each of the twelve factors in detail—there are full books that have been written that expand even greater details on each constraint. The twelve-factor methodology is now used in some application frameworks to help developers comply with some, or even

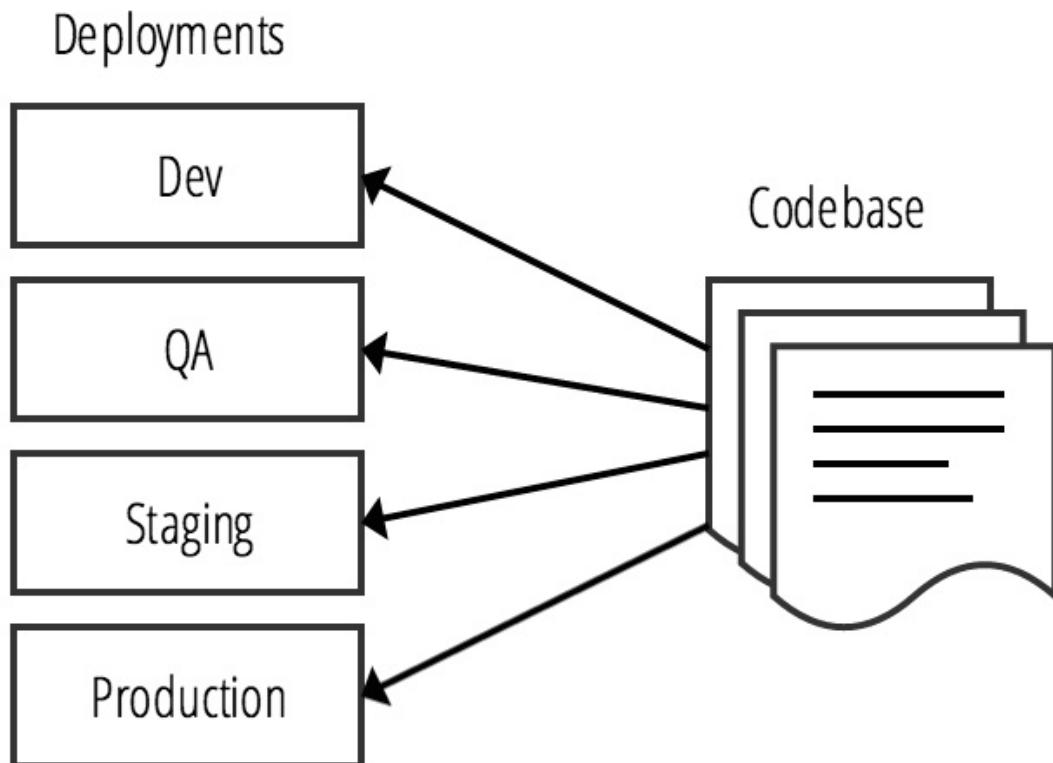
all, of the twelve factors out-of-the-box.

We'll be using the twelve-factor methodology throughout this book to describe how certain features of Spring projects were implemented to satisfy this style of application development. For this reason, it's important that we summarize each of the factors here.

Codebase

One codebase tracked in revision control, many deploys

Source code repositories for an application should contain a single application with a manifest to its application dependencies.



Dependencies

Explicitly declare and isolate dependencies

Application dependencies should be explicitly declared and any and all dependencies should be available from an artifact repository that can be downloaded using a dependency manager, such as Apache Maven.

Twelve-factor applications never rely on the existence of implicit system-wide packages required as a dependency to run the application. All dependencies of an application are declared explicitly in a manifest file that cleanly declares the detail of each reference.

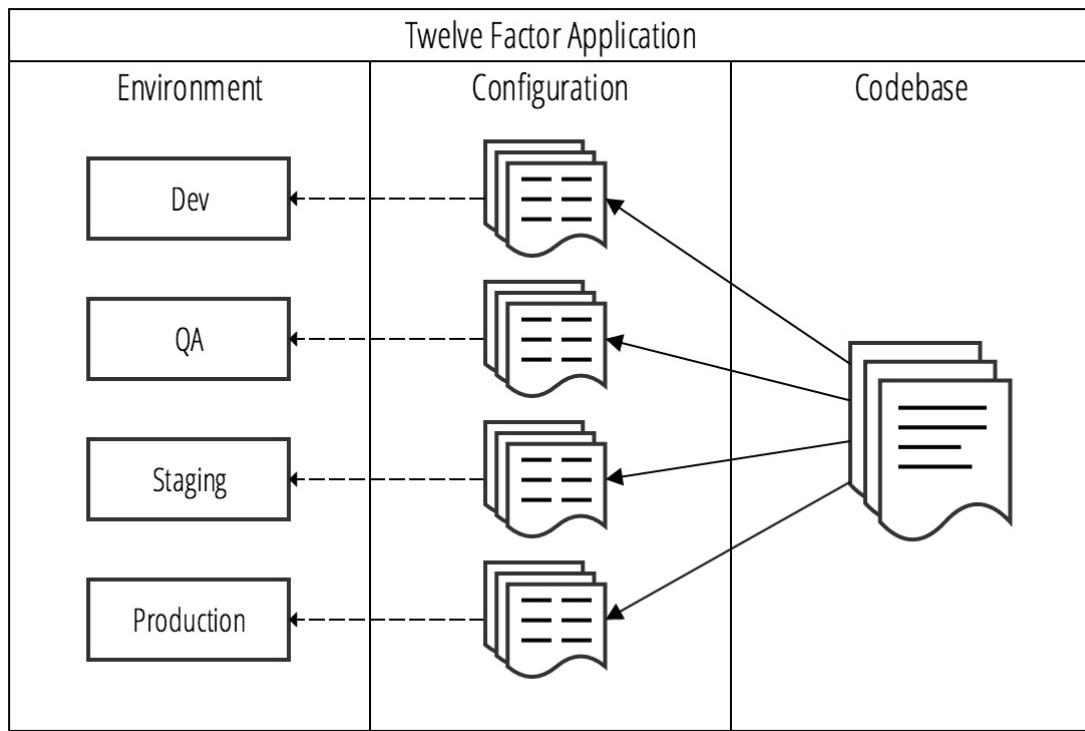
Config

Store config in the environment

Application code should be strictly separated from configuration. The configuration of the application should be driven by the environment.

Application settings such as connection strings, credentials, or host names of dependent web services, should be stored as environment variables, making them easy to change without deploying configuration files.

Any divergence in your application from environment to environment is considered an environment configuration, and should be stored in the environment and not with the application.



Backing services

Treat backing services as attached resources

A backing service is any service that the twelve-factor application consumes as a part of its normal operation. Examples of backing services are databases, API-driven RESTful web services, SMTP server, or FTP server.

Backing services are considered to be *resources* of the application. These *resources* are attached to the application for the duration of operation. A deployment of a twelve-factor application should be able to swap out an embedded SQL database in a testing environment with an external MySQL database hosted in a staging environment without making changes to the application's code.

Build, release, run

Strictly separate build and run stages

The twelve-factor application strictly separates the build, release, and run stages.

- *Build stage* — The build stage takes the source code for an application and either compiles or bundles it into a package. The package that is created is referred to as a *build*.
- *Release stage* — The release stage takes a build and combines it with its config. The release that is created for the deploy is then ready to be operated in an execution environment. Each release should have a unique identifier, either using semantic versioning or a timestamp. Each release should be added to a directory that can be used by the release management tool to rollback to a previous release.
- *Run stage* — The run stage, commonly referred to as the *runtime*, runs the application in the execution environment for a selected release.

By separating each of these stages into separate processes, it becomes impossible to change an application's code at runtime. The only way to change the application's code is to initiate the build stage to create a new release, or to initiate a rollback to deploy a previous release.

Processes

Execute the app as one or more stateless processes

Twelve-factor applications are created to be stateless in a *share-nothing architecture*. The only persistence that an application may depend on is through a backing service. Examples of a backing service that provide persistence include a database or an object store. All resources to the application are attached as a backing service at runtime. A litmus test for testing whether or not an application is stateless is that the application's execution environment can be torn down and recreated without any loss of data.

Twelve-factor applications do not store state on a local file system in the execution environment.

Port bindings

Export services via port binding

Twelve-factor applications are completely self-contained, which means that they do not require a webserver to be injected into the execution environment at runtime in order to create a web-facing service. Each application will expose access to itself over an HTTP port that is bound to the application in the execution environment. During deployment, a routing layer will handle incoming requests from a public hostname by routing to the application's execution environment and the bound HTTP port.

Note

Josh Long, one of the co-authors of this book, is attributed with popularizing the phrase "*Make JAR not WAR*" in the Java community. Josh uses this phrase to explain how newer Spring applications are able to embed a Java application server, such as Tomcat, in a build's JAR file.

Concurrency

Scale out via the process model

Applications should be able to scale out processes or threads for parallel execution of work in an on-demand basis. JVM applications are able to handle in-process concurrency automatically using multiple threads.

Applications should distribute work concurrently depending on the type of work that is used. Most application frameworks for the JVM today have this built in. Some scenarios that require data processing jobs that are executed as long-running tasks should utilize executors that are able to asynchronously dispatch concurrent work to an available pool of threads.

The twelve-factor application must also be able to scale out horizontally and handle requests load-balanced to multiple identical running instances of an application. By ensuring applications are designed to be stateless, it becomes

possible to handle heavier workloads by scaling applications horizontally across multiple nodes.

Disposability

Maximize robustness with fast startup and graceful shutdown

The processes of a twelve-factor application are designed to be disposable. An application can be stopped at any time during process execution and gracefully handle the disposal of processes.

Processes of an application should minimize startup time as much as possible. Applications should start within seconds and begin to process incoming requests. Short startups reduce the time it takes to scale out application instances to respond to increased load.

If an application's processes take too long to start, there may be reduced availability during a high-volume traffic spike that is capable of overloading all available healthy application instances. By decreasing the startup time of applications to just seconds, newly scheduled instances are able to more quickly respond to unpredicted spikes in traffic without decreasing availability or performance.

Dev/prod parity

Keep development, staging, and production as similar as possible

The twelve-factor application should prevent divergence between development and production environments. There are three types of gaps to be mindful of.

- Time gap — Developers should expect development changes to be quickly deployed into production
- Personnel gap — Developers who make a code change are closely involved with its deployment into production, and closely monitor the behavior of an application after making changes

- Tools gap — Each environment should mirror technology and framework choices in order to limit unexpected behavior due to small inconsistencies

Logs

Treat logs as event streams

Twelve-factor apps write logs as an ordered event stream to `stout`. Applications should not attempt to manage the storage of their own log files. The collection and archival of log output for an application should instead be handled by the *execution environment*.

Admin processes

Run admin/management tasks as one-off processes

It sometimes becomes the case that developers of an application need to run one-off administrative tasks. These kinds of tasks could include database migrations or running one-time scripts that have been checked into the application's source code repository. These kinds of tasks are considered to be admin processes. Admin processes should be run in the execution environment of an application, with scripts checked into the repository to maintain consistency between environments.

Chapter 3. 12-Factor Application Style Configuration

The Confusing Conflation of “Configuration”

Let's establish some vocabulary. When we talk about *configuration* in Spring, we've *usually* talked about the inputs into the Spring framework's various [ApplicationContext](#) implementations that help the container understand how you want beans wired together. This might be an XML file to be fed into a [ClassPathXmlApplicationContext](#), or Java classes annotated a certain way to be fed into an [AnnotationConfigApplicationContext](#). Indeed, when we talk about the latter, we refer to it as *Java configuration*.

In this chapter, however, we're going to look at configuration as it is defined in [12-Factor app style configuration page](#). Such configuration avoids constants embedded in the code. The page provides a great litmus test for whether configuration has been done correctly: could the codebase of an application be open-sourced at any moment without exposing and compromising important credentials? This sort of configuration refers only to the values that change from one environment to another, not - for example - to Spring bean wiring or Ruby route configuration.

Support in Spring framework

Spring has supported Twelve-Factor-style configuration since the [`PropertyPlaceholderConfigurer`](#) class was introduced. Once an instance is defined, it replaces literals in the XML configuration with values that it resolved in a `.properties` file. Spring's offered the [`PropertyPlaceholderConfigurer`](#) since 2003. Spring 2.5 introduced XML namespace support and with it XML namespace support for property placeholder resolution. This lets us substitute bean definition literal values in the XML configuration for values assigned to keys in a (external) property file (in this case `simple.properties` which may be on the class path or external to the application).

Twelve-Factor-style configuration aims to eliminate the fragility of having *magic strings* - values like database locators and credentials, ports, etc. - hard-coded in the compiled application. If configuration is externalized, then it can be replaced without requiring a rebuild of the code.

The PropertyPlaceholderConfigurer

Let's look at an example using the `PropertyPlaceholderConfigurer`, Spring XML bean definitions, and an externalized `.properties` file. We simply want to print out the value in the property file, which looks like this:

```
configuration.projectName = Spring Framework
```

This is a Spring `ClassPathXmlApplicationContext` so we use the Spring context XML namespace and point it to our `some.properties` file. Then, in the bean definitions, use literals of the form `${configuration.projectName}` and Spring will replace them at runtime with the values from our property file.

```
<context:property-placeholder location="classpath:some.properties">  
<bean class="classic.Application">  
    <property name="configurationProjectName" value="${configurat</bean>
```

❶

A `classpath:` location refers to a file in the current compiled code unit (`.jar`, `.war`, etc.). Spring supports many alternatives, including `file:` and `url:`, that would let the file live external to the compiled unit.

Finally, here's a Java class to pull it all together:

```
package classic;  
  
import org.springframework.context.support.ClassPathXmlApplicatio  
  
public class Application {  
  
    public static void main(String[] args) {  
        new ClassPathXmlApplicationContext("classic.xml")  
    }  
  
    public void setConfigurationProjectName(String pn) {  
        System.out.println("the configuration project nam
```

```
    }  
}
```

The first examples used Spring's XML bean configuration format. Spring 3.0 and 3.1 improved things considerably for developers using Java configuration. These releases saw the introduction of the `@Value` annotation and the `Environment` abstraction.

The Environment Abstraction and @Value

The [Environment](#) abstraction provides a bit of runtime indirection between the running application and the environment in which it is running and lets the application ask questions (“what’s the current platform’s `line.separator`?”) about the environment. The `Environment` acts as a map of keys and values. You can configure where those values are read from by contributing a `PropertySource`. By default Spring loads up system environment keys and values, like `line.separator`. You can tell Spring to load up configuration keys from a file, specifically, using the `@PropertySource` annotation.

The `@Value` annotation provides a way to inject values into fields. These values can be computed using the Spring Expression Language or using property placeholder syntax, assuming one registers a [PropertySourcesPlaceholderConfigurer](#).

```
package env;

import javax.annotation.PostConstruct;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;

❶
@Configuration
@PropertySource("some.properties")
public class Application {

    public static void main(String[] args) throws Throwable {
        new AnnotationConfigApplicationContext(Application.class);
    }
}
```

```

❷
@Bean
static PropertySourcesPlaceholderConfigurer pspc() {
    return new PropertySourcesPlaceholderConfigurer()
}

@Value("${configuration.projectName}")
private String configurationProjectNameField;

❸
@Value("${configuration.projectName}")
void set projectName(String projectName) {
    System.out.println("set projectName: " + projectName)
}

❹
@Autowired
void setEnvironment(Environment env) {
    System.out.println("setEnvironment: "
        + env.getProperty("configuration."
    )
}

@PostConstruct
void afterPropertiesSet() throws Throwable {
    System.out.println("configurationProjectNameField"
        + this.configurationProjectNameFi
}

❺
@Bean
InitializingBean both(Environment env,
    @Value("${configuration.projectName}") St
return () -> {
    System.out.println("@Bean with both depen
        + projectName);
    System.out.println("@Bean with both depen
        + env.getProperty("config
    );
}
}

```

❻

the `@PropertySource` annotation is a shortcut, like `propertyPlaceholder`, that configures a `PropertySource` from a `.properties` file.

②

you need to register the `PropertySourcesPlaceholderConfigurer` as a static bean because it is a `BeanFactoryPostProcessor` and must be invoked earlier in the Spring bean initialization lifecycle. This nuance is invisible when you're using Spring's XML bean configuration format.

③

you can decorate fields with the `@Value` annotation..

④

..or you can decorate fields with the `@Value` annotation.

⑤

`@Value` annotations can be declared on Spring Java configuration `@Bean` provider method arguments, as well.

This example loads up the values from a file, `simple.properties`, and then has one value, `configuration.projectName`, injected using the `@Value` annotation and then read again from Spring's `Environment` abstraction in various ways. To be able to inject the values with the `@Value` annotation, we need to register a [`PropertySourcesPlaceholderConfigurer`](#).

Profiles

The `Environment` also brings the idea of [*profiles*](#). It lets you ascribe labels (profiles) to groupings of beans. Use profiles to describe beans and bean graphs that change from one environment to another. You can activate one or more profiles at a time. Beans that do not have a profile assigned to them are always activated. Beans that have the profile `default` are activated only when there are no other profiles active. You can specify the `profile` attribute in bean definitions in XML or alternatively tag classes configuration classes, individual beans, or `@Bean`-provider methods with `@Profile`.

Profiles let you describe sets of beans that need to be created differently in one environment versus another. You might, for example, use an embedded H2 `javax.sql.DataSource` in your local `dev` profile, but then switch to a `javax.sql.DataSource` for PostgreSQL that's resolved through a JNDI lookup or by reading the properties from an environment variable in [Cloud Foundry](#) when the `prod` profile is active. In both cases, your code works: you get a `javax.sql.DataSource`, but the decision about *which* specialized instance is used is decided by the active profile or profiles.

```
package profiles;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.util.StringUtils;

@Configuration
public class Application {

    @Bean
    static PropertySourcesPlaceholderConfigurer pspc() {
        return new PropertySourcesPlaceholderConfigurer()
    }

    @Configuration
    @Profile("prod")
    ①
```

```

@PropertySource("some-prod.properties")
static class ProdConfiguration {

    @Bean
    InitializingBean init() {
        return () -> System.out.println("prod Ini
    }
}

@Configuration
@Profile({"default", "dev"})
❷
@PropertySource("some.properties")
static class DefaultConfiguration {

    @Bean
    InitializingBean init() {
        return () -> System.out.println("default
    }
}

❸
@Bean
InitializingBean which(Environment e,
    @Value("${configuration.projectName}") St
return () -> {
    System.out.println("activeProfiles: '"
        + StringUtils.arrayToComm
        .getActiv
    System.out.println("configuration.project
);
}

public static void main(String[] args) {
    AnnotationConfigApplicationContext ac = new Annot
    ac.getEnvironment().setActiveProfiles("dev"); ❹
    ac.register(Application.class);
    ac.refresh();
}
}

```

❶ this configuration class and all the `@Bean` definitions therein will only be evaluated if the `prod` profile is active.

②

this configuration class and all the `@Bean` definitions therein will only be evaluated if the `dev` profile *or* `no` profile - including `dev` - is active.

③

this `InitializingBean` simply records the currently active profile and injects the value that was ultimately contributed by the property file.

④

it's easy to programmatically activate a profile (or profiles).

Spring responds to a few other methods for activating profiles using the token `spring_profiles_active` or `spring.profiles.active`. You can set the profile using an environment variable (e.g.: `SPRING_PROFILES_ACTIVE`), a JVM property (`-Dspring.profiles.active=..`), a Servlet application initialization parameter, or programmatically.

Bootiful Configuration

[Spring Boot](#) improves things considerably. Spring Boot will automatically load properties in a hierarchy of well-known places by default. The command-line arguments override property values contributed from JNDI, which override properties contributed from `System.getProperties()`, etc.

- Command line arguments
- JNDI attributes from `java:comp/env`
- `System.getProperties()` properties`
- OS environment variables
- External property files on filesystem - `(config/)?application.(yml.properties)`
- Internal property files in archive `(config/)?application.(yml.properties)`
- `@PropertySource annotation on configuration classes`
- Default properties from `SpringApplication.getDefaultProperties()`

If a profile is active, it will also automatically reads in the configuration files based on the profile name, like `src/main/resources/application-foo.properties` where `foo` is the current profile.

If the [Snake YML library](#) is on the classpath, then it will also automatically load YML files following basically the same convention. Yeah, read that part again. YML is so good, and so worth a go! Here's an example YML file:

```
configuration:  
  projectName : Spring Boot
```

Spring Boot also makes it much simpler to get the right result in common

cases. It makes `-D` arguments to the `java` process and environment variables available as properties. It even normalizes them, so an environment variable `$CONFIGURATION_PROJECTNAME` or a `-D` argument of the form `-Dconfiguration.projectname` both become accessible with the key `configuration.projectName` in the same way that the `spring_profiles_active` token was earlier.

Configuration values are strings, and if you have enough configuration values it can be unwieldy trying to make sure those keys don't themselves become magic strings in the code. Spring Boot introduces a

`@ConfigurationProperties` component type. Annotate a POJO with `@ConfigurationProperties` and specify a prefix, and Spring will attempt to map all properties that start with that prefix to the POJO's properties. In the example below the value for `configuration.projectName` will be mapped to an instance of the POJO that all code can then inject and dereference to read the (type-safe) values. In this way, you only have the mapping from a key in one place.

```
package boot;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.stereotype.Component;

@SpringBootApplication
@EnableConfigurationProperties
❶
public class Application {

    @Autowired
    void setConfigurationProjectProperties(ConfigurationProperties cp) {
        System.out.println("configurationProjectProperties = " +
                           cp.getProjectName());
    }

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

```
②
@Component
@ConfigurationProperties("configuration")
class ConfigurationProjectProperties {

    private String projectName; ③

    public String getProjectName() {
        return projectName;
    }

    public void setProjectName(String projectName) {
        this.projectName = projectName;
    }
}
```

①

the `@EnableConfigurationProperties` annotation tells Spring to map properties to POJOs annotated with `@ConfigurationProperties`.

②

`@ConfigurationProperties` tells Spring that this bean is to be used as the root for all properties starting with `configuration.`, with subsequent tokens mapped to properties on the object.

③

the `projectName` field would ultimately have the value assigned to the property key `configuration.projectName`.

Spring Boot uses the `@ConfigurationProps` mechanism heavily to let users override bits of the system. You can see what property keys can be used to change things, for example, by adding the

`org.springframework.boot:spring-boot-starter-actuator` dependency to a Spring Boot-based web application and then visiting

`http://127.0.0.1:8080/configprops`. This will give you a list of supported configuration properties based on the types present on the classpath at runtime. As you add more Spring Boot types, you'll see more properties. This endpoint will *also* reflect the properties exported by your `@ConfigurationProperties`-annotated POJO.

Centralized, Journaled Configuration with the Spring Cloud Configuration Server

So far so good, but there are gaps in the approach so far:

- changes to an application's configuration require restarts
- there is no traceability: how do we determine what changes were introduced into production and, if necessary, roll back?
- configuration is de-centralized and it's not immediately apparent where to go to change what.
- sometimes configuration values should be encrypted and decrypted for security. There is no out-of-the-box support for this.

[Spring Cloud](#), which builds upon Spring Boot and integrates various tools and libraries for working with microservices, including [the Netflix OSS stack](#), offers a [configuration server](#) and a client for that configuration server. This support, taken together, address these last three concerns.

Let's look at a simple example. First, we'll setup a configuration server. The configuration server is something to be shared among a set of applications or microservices based on Spring Cloud. You have to get it running, somewhere, once. Then, all other services need only know where to find the configuration service. The configuration service acts as a sort of proxy for configuration keys and values that it reads from a Git repository online or on a disk.

Tip

Add the Spring Cloud Config server dependency:

```
org.springframework.cloud:spring-cloud-config-server
```

CODE TO COME ①

①

`@EnableConfigServer` installs a configuration service.

Here's the configuration for the configuration service:

CODE TO COME ① ②

①

This is a normal Spring Boot-ism that configures on which port the embedded web server (in this Apache Tomcat) is to use.

②

Points to the working Git repository, either local or over the network (like [GitHub](#)), that the Spring Cloud Config server is to use.

This tells the Spring Cloud configuration service to look for configuration files for individual client services in the Git repository on my GitHub account. The URI could, of course, just as easily have been a Git repository on my local file system. The value used for the URI could also have been a property reference, of the form, `${SOME_URI}`, that references - perhaps - an environment variable called `SOME_URI`.

Run the application and you'll be able to verify that your configuration service is working by pointing your browser at

`http://localhost:8888/SERVICE/master` where SERVICE is the ID taken from your client service's `bootstrap.yml`. Spring Cloud-based services look for a file called `src/main/resources/bootstrap.(properties,yml)` that it expects to find to - you guessed it! - bootstrap the service. One of the things it expects to find in the `bootstrap.yml` file is the ID of the service specified as a property, `spring.application.name`.



Figure 3-1. The output of the Spring Coud Config Server confirming that it *sees* the configuration in our Git repository

If you manage things correctly, then the only configuration that lives with any of your services should be the configuration that tells the configuration service where to find the Git repository and the configuration that tells the other client services where to find the configuration service, both of which usually live in a file called `bootstrap.yml` in Spring Cloud-based services. This file (or `bootstrap.properties`) gets loaded than other property files (including `application.yml` or `application.properties`). It makes sense: this file tells Spring where it's to find the rest of the application's configuration. Here's our configuration client's `bootstrap.yml`.

```
spring:
  application:
    name: configuration-client
  cloud:
    config:
      uri: ${vcap.services.configuration-service.credentials.uri:}
```

When a Spring Cloud microservice runs, it'll see that its `spring.application.name` is `config-client`. It will contact the configuration service (which we've told Spring Cloud is running at `http://localhost:8080`, though this too could've been an environment variable) and ask it for any configuration. The configuration service returns back JSON that contains all the configuration values in the `application.properties` file as well as any service-specific configuration in `config-client.(yml, properties)`. It will *also* load any configuration for a

given service *and* a specific profile, e.g., config-client-dev.properties.

This all just happens automatically and you can interact with properties exposed via the configuration server like any other configuration property.

Example 3-1.

```
package demo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class Application {

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

❶
@RestController
@RefreshScope
class ProjectNameRestController {

    ❷
    @Value("${configuration.projectName}")
    private String projectName;

    @GetMapping("/project-name")
    String projectName() {
        return this.projectName;
    }
}
```

❶

TO COME

❷

TO COME

Security

Define `spring.cloud.config.server.git.username` and `spring.cloud.config.server.git.password` properties for the Spring Cloud Config Server to talk to secured Git repositories.

You can protect the Spring Cloud Configuration Server itself with HTTP BASIC authentication. The easiest is to just include `org.springframework.boot:spring-boot-starter-security` and then define a `security.user.name` and a `security.user.password` property.

The Spring Cloud Config Clients can encode the user and password in the `spring.cloud.config.uri` value, e.g.: `https://user:secret@host.com`.

Refreshable Configuration

Centralized configuration is a powerful thing, but changes to configuration aren't immediately visible to the beans that depend on it. Spring Cloud's *refresh* scope offers a solution.

The `ProjectNameRestController` is annotated with `@RefreshScope`, a Spring Cloud scope that lets any bean recreate itself (and re-read configuration values from the configuration service) in-place. In this case, the `ProjectNameRestController` will be recreated - its lifecycle callbacks honored and `@Value` and `@Autowired` injects re-established - whenever a *refresh* is triggered.

Fundamentlaly, all *refresh*-scoped beans will refresh themselves when they receive a `Spring ApplicationContext`-event of the type `RefreshScopeRefreshedEvent`. There are various ways to trigger the refresh.

You can trigger the refresh by sending an empty `POST` request to `http://127.0.0.1:8080/refresh`, which is a Spring Boot Actuator endpoint that is exposed automatically. Here's how to do that using `curl`:

```
curl -d{} http://127.0.0.1:8080/refresh`
```

Alternatively, you can use the auto-exposed Spring Boot Actuator JMX refresh endpoint.

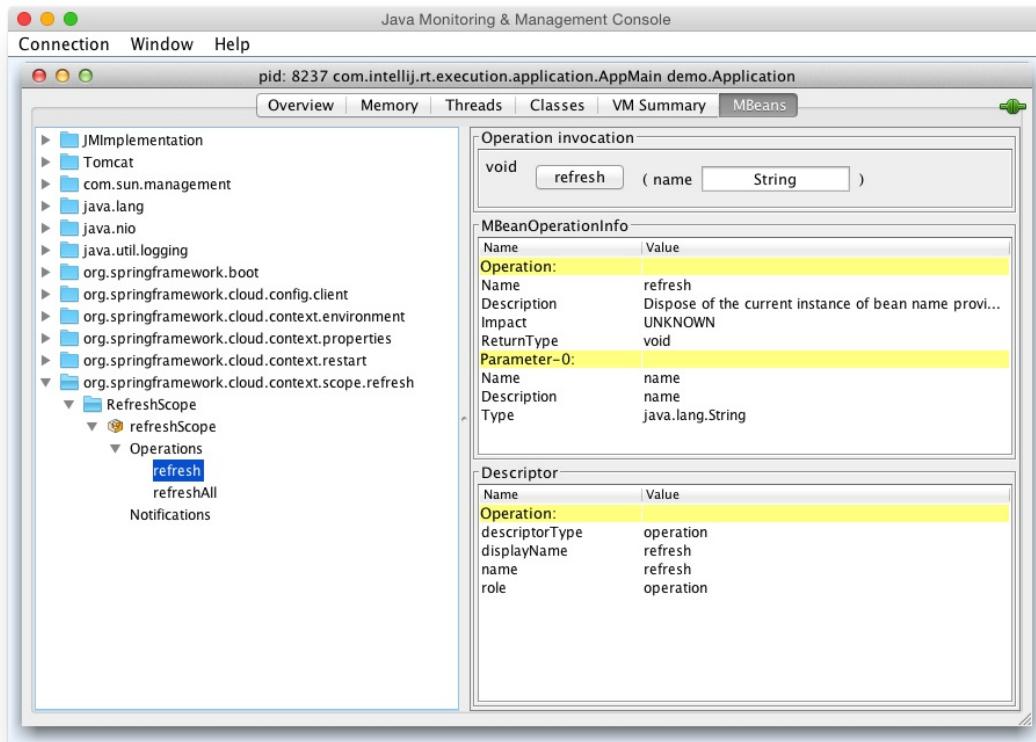


Figure 3-2. Using jconsole to activate the `refresh` (or `refreshAll`) Actuator endpoints

To see all this in action, make changes to the configuration file in Git, and at the very least `git commit` them. Then invoke the REST endpoint or the JMX endpoint on the node you want to see the configuration changed in, *not* the configuration server.

Both of those Spring Boot Actuator endpoints work on an `ApplicationContext`-by-`ApplicationContext` basis. The Spring Cloud Bus, on the other hand, provides a simple way to refresh multiple `ApplicationContext`'s (e.g.: many nodes) in one go.

The [Spring Cloud Bus](#) links all services through a RabbitMQ powered-bus. This is particularly powerful. You can tell one (or thousands!) of microservices to refresh themselves by sending a single message to a message bus. This prevents downtime and is *much* more friendly than having to systematically restart individual services or nodes.

Tip

Add the Spring Cloud Event Bus dependency `org.springframework.cloud:spring-cloud-starter-bus-amqp`.

By default Spring Boot's auto-configuration for RabbitMQ will attempt to connect to a local RabbitMQ instance. You can configure the specific host and port in the usual way in your application's `application.yml`. These specifics tend to apply to multiple services, so you might consider putting them in the `application.yml` in your configuration server's Git repository. This way, *all* services that connect to the configuration service will *also* talk to the right RabbitMQ instance.

```
spring:  
  rabbitmq:  
    host: 127.0.0.1  
    port: 5672  
    username: user  
    password: secret
```

These configuration values create a Spring AMQP `ConnectionFactory` bean that will be used to listen for event bus messages. If you have multiple `ConnectionFactory` instances in the Spring application context, you need to *qualify* which instance is to be used with the `@BusConnectionFactory` annotation. Qualify any other instance to be used for regular, non-bus business processing with the usual Spring qualifier annotation, `@Primary`.

The Spring Cloud Event Bus exposes a *different* Actuator endpoint, `/bus/refresh`, that will publish a message to the connected RabbitMQ broker that will trigger *all* connected nodes to refresh themselves. You can send the following message to *any* node with the `spring-cloud-starter-bus-amqp` auto-configuration, and it'll trigger a refresh in *all* the connected nodes.

```
curl -d{} http://127.0.0.1:8080/bus/refresh`
```

Next Steps

We've covered a *lot* here! Armed with all of this, it should be easy to package one artifact and then move that artifact from one environment to another without changes to the artifact itself. If you're going to start an application today, I'd recommend starting on Spring Boot and Spring Cloud, especially now that we've looked at all the good stuff it brings you by default. Don't forget [to check out the code](#) behind all of these examples.

Chapter 4. Testing

As applications become increasingly distributed, the strategies for how we effectively write tests changes considerably.

As most readers of this book are already aware, the practice of *integration testing* focuses on writing and executing tests against a group of software modules that depend on one another. Integration testing is a standard practice in software development where developers who are working on separate modules or components are able to automate a set of test cases which exist to ensure that the expected functionality of integration remains true—*especially* when changes are made to code that affects an integration.

It can often be the case that integration testing requires executing tests in a shared integration environment. In this scenario, applications may be subject to concurrently sharing external resources, such as a database or an application server.

Cloud native applications are designed to take advantage of the ephemeral nature of a cloud environment. When it comes to testing these applications, we should focus on designing integration tests so that they can be executed in an ephemeral environment that is decoupled from other applications. There are many cases where cloud native applications will depend on backing services. Where necessary, these dependencies should be mocked so that integration tests can be executed in a decoupled build and test environment. There will be common scenarios where mocking an external dependency becomes untenable. In such cases, any external backing service should be provisioned on-demand and torn down when testing has completed.

In this chapter we will focus on two primary integration testing topics. The first topic will be about how to design and create basic integration tests for Spring Boot applications. Here we will explore the tools and features of Spring that are critical to how we create meaningful integration tests that are decoupled from external dependencies.

The second topic we will explore will be how to perform end-to-end integration testing in a microservice architecture. Here we will focus on the various forms of testing that execute over a collection of different services in a fully formed test environment that resembles production.

Testing in Spring Boot

Testing in Spring Boot applications will break down into two separate styles of testing: unit tests and integration tests. There is a very simple difference between these two kinds of tests. An integration test is any test that requires access to the Spring context—more specifically, the `ApplicationContext`—during test execution. Unit testing on the other hand can be written in such a way that a Spring context is not required. When designing a testing strategy for your Spring Boot applications, it's important to understand when a test requires an `ApplicationContext` and when it does not.

The first step to enabling testing features in a Spring Boot application is to add the provided starter project dependency to your `pom.xml`.

Example 4-1. Adding the Spring Boot starter test dependency

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

After adding the test dependency, you'll be able to start creating tests in your project. In [Example 4-2](#) we see a simple integration test inside the class named `ApplicationTests`. The goal of the test named `contextLoads` is to assert that the `ApplicationContext` was successfully loaded and injected into the field named `applicationContext`.

Example 4-2. Basic integration test loads the `ApplicationContext`

```
package com.example;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {

    @Autowired
    protected ApplicationContext applicationContext;

    @Test
    public void contextLoads() {
        Assert.notNull(applicationContext);
    }
}

```

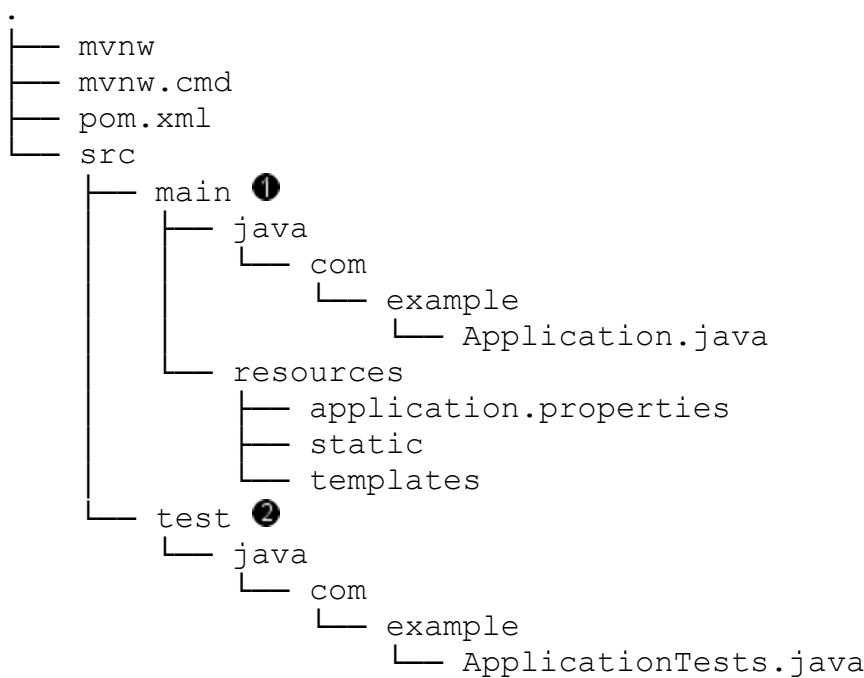
We also see that there are two annotations on the `ApplicationTests` class named `@RunWith` and `@SpringBootTest`. The `@RunWith` annotation is one that is specific to the JUnit framework. The purpose of the annotation is to use JUnit to load the enclosed `SpringRunner.class`, which allows us to run JUnit together with the *Spring TestContext Framework*—a member of the Spring Framework library that provides generic testing support for Spring applications. The `SpringRunner` class was introduced in Spring Boot 1.4 and serves as an alias that shortens and simplifies the name of the Spring implementation of the JUnit runner class, which is named `SpringJUnit4ClassRunner`.

The second annotation on the `ApplicationTests` class is `@SpringBootTest`. The `@SpringBootTest` annotation indicates that this class is a Spring Boot test class, and provides support to scan for `ContextConfiguration` that tells the test class how to load the `ApplicationContext`. By default, if no `ContextConfiguration` classes are specified as a parameter to the `@SpringBootTest` annotation, the default behavior is that the `ApplicationContext` will be loaded from scanning for a `@SpringBootConfiguration` annotation in the package root.

As was explained earlier in the [Chapter 1](#) chapter, the `@SpringBootApplication` annotation is a stereotype definition that combines multiple lower-level annotations in a Spring Boot application. One of the lower-level nested annotations of `@SpringBootApplication` is the `@SpringBootConfiguration` annotation. This is how a `@SpringBootTest` class will find the application class that loads the correct `ApplicationContext`.

In [Example 4-3](#) we see the folder structure of a basic Spring Boot application's source code. Notice that there are two sources roots for the application, both with identical package structures. For the test methods inside the `ApplicationTests.java` file to be able to successfully scan and find the Spring Boot application class contained in `Application.java`—being annotated with `@SpringBootApplication`—the package structure should be identical in each of the sources roots.

Example 4-3. The basic structure of a new Spring Boot project



❶

The sources root containing the application class

❷

The test sources root containing the application's test classes

The reason for this is because any test class annotated with `@SpringBootTest` will load an `ApplicationContext` by scanning for the `@SpringBootApplication` annotation that is on the `Application.java` class found in `src/main/java/com/example`. The source code for the

Application.java class is found in [Example 4-4](#).

Example 4-4. The annotated Spring Boot application class in the com.example package

```
package com.example; ①

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

@SpringBootApplication ②
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

①

The @SpringBootTest class must be in the same package as the @SpringBootApplication class

②

The @SpringBootApplication annotation contains the nested @SpringBootConfiguration annotation

Now that we have the basics covered on how the Spring Boot test starter dependency can be used to create an auto-configured JUnit test class, we can begin to design and create integration tests that target the interactions between various components that are loaded in an ApplicationContext.

Integration Testing

As was mentioned earlier, there are two styles of tests you'll want to create for your Spring Boot application: *unit tests* and *integration tests*. The main difference between the two styles of tests is one will require a Spring context—to test the integration between different components—and the other—*our unit tests*—will test individual components without any dependency on the Spring libraries. In this section we're going to focus strictly on writing integration tests in a Spring Boot application.

Writing integration tests for Spring Boot applications can at times be challenging due to the nature of auto-configuration classes. Auto-configuration is what makes a Spring Boot application tick. The benefit we get from auto-configuration is that Spring components no longer have to be manually configured in order to start being used. The main challenge that stems from this is that it can at times be difficult to understand how auto-configuration will differ in a testing context in a build environment and at runtime in a production environment.

One of the important guidelines from the 12-factor application methodology states that we should make sure to minimize the divergence between development and production. For most application frameworks other than Spring Boot, it would likely be important to follow this advice with strict reverence when building a cloud-native application. For a cloud-native Spring Boot application, the rules with regard to dev/prod parity can be bent somewhat, since auto-configuration allows us to switch external dependencies for mocked dependencies. Indeed, there is certainly a place for testing the end-to-end behavior of an application in an environment that exactly mirrors the backing services used in production—but more on that later.

Test Slices

Before Spring Boot 1.4 was released, integration tests would mostly be written to be executed after loading the full Spring context of your application. In many ways this was a cumbersome side effect of auto-configuration, since not all integration tests will need access to a full `ApplicationContext`.

For example, what if I wanted to only test serialization and marshaling of Java objects to and from JSON? In this case there is little need to load auto-configured classes for *Web MVC* or *Spring Data JPA*. Do I really need a servlet container to be loaded if I'm just testing JSON serialization? The answer is most definitely *no*. Because of the loosely coupled modularity of the different starter projects in Spring Boot, we'll more often than not have integration tests that will not need access to the full Spring context of the application.

The good news is that as of the Spring Boot 1.4 release there were major testing improvements that began to allow Spring Boot developers to start thinking about testing only the “slices” of an application’s auto-configuration classes. Through the addition of multiple different test annotations, a Spring Boot test class can now target a specific slice of the Spring context that is relevant only to the needs of the integration tests contained in the single test class. This step forward for testing in Spring Boot creates a simplified experience for designing and creating integration tests that isolate separate starter project dependencies.

Test slices also provide the benefit of being able to cleanly swap out certain starter projects—for example, switching from *Spring Data JPA* to *Spring Data MongoDB*, without affecting integration tests unrelated to Spring Data. In addition to test slices, Spring Boot 1.4 includes more first-class support for writing integration test classes that allow developers to declaratively mock collaborating components in the Spring context using annotations.

Mocking in Tests

Mocking is usually talked about in the context of unit testing. In the simplest sense, *mock objects* allow us to isolate parts of the system under test by replacing collaborating components of a module with a simulated object that targets testing behavior in a controlled way. For example, if one service module of my application requires access to an external backing service—perhaps taking the form of a call to another microservice—to test the module in isolation requires that we are able to mock the expected behavior of the backing service.

Mocks in Spring Boot often becomes a nuanced topic between team members when talking about the differences between an integration test and a unit test. Mocks can be equally useful when used in integration tests as they are in unit tests for a Spring Boot application. When writing integration tests in Spring Boot, it becomes possible to mock only selected components in the `ApplicationContext` of a test class. This helpful feature allows developers to test integrations between collaborating components while still being to mock objects at the boundary of the application. The difference between a unit test and an integration test *still* being whether or not a Spring context is used at all. This distinction is important to remember and will aid communication between team members who are writing both unit tests and integration tests.

As of Spring Boot 1.4, it is now possible to use the `@MockBean` annotation in a test class to define a Mockito mock for a bean inside the `ApplicationContext`. In [Example 4-5](#) we see the `AccountServiceTests` class that is annotated with a `@RunWith(SpringRunner.class)` annotation—indicating that no `ApplicationContext` will be loaded during test execution. The component that is being tested in this class is the `AccountService` bean, which relies on an integration with an external microservice that is accessed through the `UserService` bean.

Example 4-5. Mocks the `UserService` and `AccountRepository` components using a `@MockBean` annotation

```

@RunWith(SpringRunner.class)
public class AccountServiceTests {

    @MockBean ❶
    private UserService userService;

    @MockBean
    private AccountRepository accountRepository;

    private AccountService accountService;

    @Before
    public void before() {
        accountService = new AccountService(accountRepository, us
    }

    @Test
    public void getUserAccountsReturnsSingleAccount() throws Exce
        given(this.accountRepository.findAccountsByUsername("user"
            .willReturn(Collections.singletonList(new Account
                new AccountNumber("123456789")))); ❸

        given(this.userService.getAuthenticatedUser())
            .willReturn(new User(0L, "user", "John", "Doe"));

        List<Account> actual = accountService.getUserAccounts();

        assertThat(actual).size().isEqualTo(1);
        assertThat(actual.get(0).getUsername()).isEqualTo("user")
        assertThat(actual.get(0).getAccountNumber())
            .isEqualTo(new AccountNumber("123456789"));
    }
}

```

❶

Creates a Mockito mock for the `UserService` component

❷

Creates a new instance of the `AccountService` with mocked components as parameters

❸

Stubs the repository method call to `findAccountsByUsername(String`

username) with an Account list

④

Stubs the method call to `getAuthenticatedUser()` with a new instance of `User`

⑤

Calls the `getUserAccounts()` method of the `AccountService` while using the defined mocks

In this example we are mocking the behavior of collaborating components that will later need to be integration tested. In this unit test, we are able to test the functionality of the `AccountService` without making remote HTTP calls to a backing service inside the `UserService` bean. The same applies for the `AccountRepository` component. Normally this repository component would be defined as an auto-configured bean inside the `ApplicationContext`, which provides data management for an `Account` entity that is mapped to a table in a relational database. To reduce the complexity of testing just the functionality of the `AccountService`, we can create a mock for the `AccountRepository` component and describe the expected behavior of its test interaction. Now let's take a look at the contents of the `AccountService`, to understand better what it is doing.

In [Example 4-6](#) we find the definition of the `AccountService` component that we are writing tests for.

Example 4-6. The definition of the `AccountService` bean that depends on collaborating components

```
@Service
public class AccountService {

    private AccountRepository accountRepository;
    private UserService userService; ①

    public AccountService(AccountRepository accountRepository, Us
        this.accountRepository = accountRepository;
        this.userService = userService;
    }
}
```

```

public List<Account> getUserAccounts() {
    List<Account> accounts = null;
    User user = userService.getAuthenticatedUser(); ③

    if (user != null)
        accounts = accountRepository
            .findAccountsByUsername(user.getUsername());

    return accounts;
}
}

```

①

The field for a `UserService` bean that will be injected through the constructor

②

Constructor-based injection will inject beans from the `ApplicationContext` for each parameter

③

The `getAuthenticatedUser()` method makes a remote HTTP call to a user microservice

Notice how the `AccountService` definition defines two collaborating components as fields—one for `UserService` and one for `AccountRepository`. Also notice how the class does not use an `@Autowired` annotation. Since the `@Service` component only defines a single constructor with beans that we are expecting to be defined in the `ApplicationContext`, Spring will automatically inject an instance of each dependency through the constructor of the `AccountService`.

The approach that is used here—one that we will use frequently throughout this book—is called *constructor-based injection*. In a contrasting approach, we can use the `@Autowired` annotation directly on a field declaration to inject a reference for a bean that sits inside the `ApplicationContext`. This alternate approach is referred to as *field-based injection*. The important rule here is to *always* use *constructor-based injection* instead of *field-based injection* for

components that you intend to use mocks for in your unit or integration tests. By doing this, you'll be able to cleanly mock dependencies of a component that are outside of the `ApplicationContext`.

It's also important to mention that for the `AccountService`, a reference to the `AccountRepository` component will be loaded from the `ApplicationContext` if it is available. To override this behavior, we can create a new instance of `AccountService` and directly initialize the class with a mock object for the `AccountRepository` bean—overriding the `ApplicationContext` definition for the repository bean. Now our test class much more closely resembles a unit test—which successfully isolates the functionality of the `AccountService` component from its collaborating dependencies.

Let's now take a look at the final piece of the puzzle for this example by inspecting the definition of the `UserService` component. In [Example 4-7](#) we find the definition of the `UserService` class that contains a single method that will make a remote HTTP call to a backing service's REST API in order to retrieve the authenticated `User`.

Example 4-7. The `UserService` bean makes remote HTTP calls to an external microservice

```
@Service
public class UserService {

    private RestTemplate restTemplate;

    public UserService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public User getAuthenticatedUser() {
        return restTemplate.getForObject("http://user-service/uaa")
    }
}
```

①

Makes a remote HTTP call to the specified URL and returns an instance of `User`

In this fairly simple `UserService` component, we are using a `RestTemplate` to make a `GET` request to a remote dependency over HTTP. By creating a mock and stub for the `getAuthenticatedUser` method for the `UserService` defined in the `AccountServiceTests` class, we further isolate the system under test by removing any dependency on an external service—since the remote resource may not be available in a build/test environment.

As you can see, being able to use `@MockBean` becomes a powerful tool for writing tests in your Spring Boot application. While this example did not require a Spring context, there are many scenarios where creating mock objects for only a few beans inside the `ApplicationContext` is tremendously useful.

Consider integration testing for a *Spring MVC* controller class that relies on multiple collaborating components that must integrate with remote services using HTTP—a *common case for integration testing in microservices*. Since we'll need a web environment for testing the *Spring MVC* controller, an `ApplicationContext` will be required. In this scenario we only want to create mocks for the services that make calls to remote applications. By mocking objects at the boundary of the web application, we can isolate test components from their remote web service dependencies. This allows us to integration test the communication between modules of our application without going outside the boundaries of the JVM.

For integration tests that require a web environment, the `@SpringBootTest` annotation provides additional configurable parameters that refine how tests will run in a servlet environment.

Testing Annotations

As was mentioned earlier in the chapter, Spring Boot now provides multiple test annotations that help you target tests on specific slices of auto-configured classes.

@SpringBootTest

Earlier in this chapter we saw an example of using `@SpringBootTest` to execute an integration test on a Spring Boot `ApplicationContext`. In a majority of cases, a Spring Boot application will need access to a servlet container—even if only to expose its health metrics from an HTTP endpoint. In times of old, if a servlet container was required, we would need to compile a Spring MVC application as a WAR artifact and deploy it to a running application server. Times have changed. Today the most popular deployment model—*by and far*—is to embed a servlet container in a Spring Boot application’s compiled JAR artifact. By doing this we gain flexibility to deploy applications to ephemeral container environments that are equipped with nothing more than a version of the JDK required to run the application.

The `@SpringBootTest` annotation should be used when you want to write integration tests on the fully auto-configured `ApplicationContext` of a Spring Boot application. This annotation will also allow you to configure the servlet environment for your test context. Using the `webEnvironment` annotation, we can describe exactly how Spring Boot should configure the embedded servlet container that your application intends to use at runtime.

Table 4-1. `@SpringBootTest`’s `webEnvironment` attribute

| Option | Description |
|--------|-------------------------------------------------------------------------------------|
| MOCK | Loads a <code>WebApplicationContext</code> and provides a mock servlet environment. |

`DEFINED_PORT` Loads an `EmbeddedWebApplicationContext` and provides a real servlet environment on a defined port.

`RANDOM_PORT` Loads an `EmbeddedWebApplicationContext` and provides a real servlet environment on a random port.

`NONE` Loads an `ApplicationContext` using `SpringApplication` but does not provide any servlet environment (mock or otherwise).

In [Table 4-1](#) we see the different options for using a servlet environment in a test class annotated with `@SpringBootTest`. The importance of understanding when and when not to use a real servlet environment for your tests comes down to the time it takes to build your application.

In a majority of cases, a real servlet environment may be overkill for your integration tests. For each Spring Boot test class, the servlet container will need to be reloaded to run the tests contained within each class. While the time it takes to start a single servlet container will vary depending on the build/test environment, the total time it takes to execute each integration test may be unnecessarily long. In a world of continuous delivery and microservices, build time begins to become a constraint.

If you're sharing a build environment with other teams, there may only be so many workers available at any one time. If your application takes an inordinate amount of time to run integration tests, you'll be hogging precious resources that could hold up other teams needing to deploy code to production. The `webEnvironment` attribute of `@SpringBootTest` allows you to configure when and when not to use a real servlet environment for your integration tests.

In addition to `@SpringBootTest`, Spring Boot provides multiple testing annotations that targets a specific slice of your application.

@JsonTest

The `@JsonTest` annotation allows you to target auto-configured classes for the purpose of testing JSON serialization and deserialization. In [Example 4-8](#) we see the `UserTests` class that is annotated with `@JsonTest`. The methods of this class intend to apply unit testing for how the `User` object is serialized and deserialized.

Example 4-8. Using `@JsonTest` to test JSON serialization/deserialization

```
@RunWith(SpringRunner.class)
@JsonTest
public class UserTests {

    private User user;

    @Autowired
    private JacksonTester<User> json; ❶

    @Before
    public void setUp() throws Exception { ❷
        User user = new User("user", "Jack", "Frost", "jfrost@example.com");
        user.setId(0L);
        user.setCreatedAt(12345L);
        user.setLastModified(12346L);

        this.user = user;
    }

    // ...
}
```

❶

AssertJ based JSON tester backed by Jackson

❷

Sets up the `User` object to be used in the tests

The `UserTests` class has just two test methods. The first test method is `serializeJson`, which takes the `User` object that was initialized in the `setUp`

method and attempts to serialize it to JSON. In [Example 4-9](#) we can see the body of the `serializeJson` test method.

Example 4-9. Tests JSON serialization of the `User` object

```
@Test  
public void serializeJson() throws Exception {  
    assertThat(this.json.write(user)).isEqualTo("user.json"); ①  
    assertThat(this.json.write(user)).isEqualToJson("user.json");  
    assertThat(this.json.write(user)).hasJsonPathStringValue("@.u  
        assertJsonPropertyEquals("@.username", "user"); ②  
}
```

①

Write the `User` object to JSON and compare to `user.json` file

②

Asserts that the actual JSON result matches for an expected property value

Notice how the `serializeJson` test method references a `user.json` resource as a parameter to the `isEqualTo` method. Here we can specify a JSON file as a class path resource in order to test that the Jackson JSON writer generates the expected result.

Example 4-10. Structure of the test sources root containing the `UserTests` class

```
├── ./src/test  
|  
|   └── java  
|       └── demo  
|           └── user  
|               ├── UserControllerTest.java  
|               ├── UserRepositoryTest.java  
|               └── UserTests.java  
|  
└── resources  
    ├── data-h2.sql  
    └── demo  
        └── user
```

```
└── user.json *
```

In [Example 4-10](#) we find the directory structure of the test sources root containing the `UserTests` class. Notice that in the

`./src/test/resources/demo/user` directory we find the `user.json` file that we are referencing in the body of the `serializeJson` test method from [Example 4-9](#). The contents of this file are found in [Example 4-11](#).

Example 4-11. The contents of the `user.json` file in the test resources

```
{
    "username": "user",
    "firstName": "Jack",
    "lastName": "Frost",
    "email": "jfrost@example.com",
    "createdAt": 12345,
    "lastModified": 12346,
    "id": 0
}
```

There are multiple options for asserting whether or not the actual result of the JSON serialization matches the expected result of the test method. The ability to map a JSON file on the class path as a test resource goes a long way in cleaning up the body of test methods—especially when the expected JSON result has many properties. Alternatively, the JSON string can be constructed in the body of the test method.

Example 4-12. Tests JSON deserialization of the `User` object

```
@Test
public void deserializeJson() throws Exception {
    String content = "{\"username\": \"user\", \"firstName\": \"J
                     \"lastName\": \"Frost\", \"email\": \"jfrost@exampl
assertThat(this.json.parse(content))
    .isEqualTo(new User("user", "Jack", "Frost", "jfrost@
assertThat(this.json.parseObject(content).getUsername()).isEq
}
```

In [Example 4-12](#) we find the second test method of the `UserTests` class—named `deserializeJson`. This method does the inverse of the

`serializeJson` test method, testing whether or not the expected JSON string is properly serialized into the specified `User` object.

@WebMvcTest

To test individual Spring MVC controllers in a Spring Boot application, the `@WebMvcTest` annotation is provided. This annotation will only auto-configure the necessary Spring MVC infrastructure needed to test interactions with controller methods.

Example 4-13. Using `@WebMvcTest` to test a Spring MVC controller

```
@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerTest {

    @Autowired
    private MockMvc mvc; ❶

    @MockBean
    private AccountService accountService; ❷

    @Test
    public void getUserAccountsShouldReturnAccounts() throws Exce
        String content = "[{\\"username\\": \"user\", \\"accountNumb

        given(this.accountService.getUserAccounts())
            .willReturn(Collections.singletonList(new Account

        this.mvc.perform(get("/v1/accounts")).accept(MediaType.APP
            .andExpect(status().isOk()).andExpect(content().j
    }
}
```

❶

Mock MVC client for performing HTTP requests to Spring MVC controllers

❷

Mocks the `AccountService` component that collaborates with the

AccountController

③

Define the expected behavior of retrieving user accounts from the accountService bean

④

Finally, use the `MockMvc` client to assert for an expected HTTP result from the `AccountController`

In [Example 4-13](#) we see a test class annotated with `@WebMvcTest` that targets HTTP endpoints from MVC methods defined in the `AccountController` class. Here we are able to mock collaborating components of the system under test – with regards to the `AccountController` class. This allows us to limit our test surface to only the `AccountController` module and the MVC method under test. In the case that the mocked `AccountService` has collaborating components – `@MockBean` can be used to expand the test surface to test only the component integrations between `AccountController` and `AccountService`.

@DataJpaTest

As of Spring Data 1.4 a new testing annotation is provided named `@DataJpaTest`. This annotation is useful for Spring Boot applications that use the Spring Data JPA project. While there are other Spring Data projects, embedded in-memory database support is provided for Spring Data JPA for the purpose of testing. Later on in [Chapter 6](#) we will explore how to configure different Spring profiles for integration testing and runtime – switching between using MySQL and H2, an embedded in-memory relational database.

Example 4-14. Using `@DataJpa` to test a Spring Data JPA repository

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class AccountRepositoryTest {
```

```

private static final AccountNumber ACCOUNT_NUMBER = new AccountNumber("1234567890");

@Autowired
private AccountRepository accountRepository; ①

@Autowired
private TestEntityManager entityManager; ②

@Test
public void findUserAccountsShouldReturnAccounts() throws Exception {
    this.entityManager.persist(new Account("jack", ACCOUNT_NUMBER));
    List<Account> account = this.accountRepository.findAccounts();
    assertEquals(account.size(), 1);
    Account actual = account.get(0);
    assertEquals(actual.getAccountNumber(), ACCOUNT_NUMBER);
    assertEquals(actual.getUsername(), "jack");
}
}

```

①

Inject the `AccountRepository` from the `ApplicationContext`

②

Inject the `TestEntityManager` for managing persistence without using a repository

③

Persists a new `Account` entity to the in-memory database configured for the context

In [Example 4-14](#) we see a test class named `AccountRepositoryTest` that is annotated with `@DataJpaTest`. By using this annotation, only the necessary auto-configuration classes will be registered that are required for executing tests on Spring Data JPA repositories. In the test method, we see that the `entityManager` field is used to persist a new instance of the `Account` entity that will represent the actual result that is expected to be returned from the `accountRepository`. The `TestEntityManager` is a useful component used in the scope of JPA repository tests that allows you to interact with the underlying datastore to persist objects without needing to do so using a repository.

Tip

It's also possible to use the `@DataJpaTest` annotation on test classes that use `JdbcTemplate` – in addition to Spring Data Repositories.

`@RestClientTest`

The `@RestClientTest` annotation provides support to test rest clients – in the form of Spring's `RestTemplate`.

Example 4-15. Using `@RestClientTest` to mock `RestTemplate` responses

```
@RunWith(SpringRunner.class)
@RestClientTest({UserService.class})
@AutoConfigureWebClient(registerRestTemplate = true) ❶
public class UserServiceTests {

    @Autowired
    private UserService userService;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void getAuthenticatedUserShouldReturnUser() {
        // Mock the expected HTTP response from the user microservice
        this.server.expect(requestTo("http://user-service/uaa/v1/")
            .andRespond(withSuccess(getClassPathResource("user.json"))
                .withMediaType(APPLICATION_JSON))); ❷

        User user = userService.getAuthenticatedUser();

        assertThat(user.getUsername()).isEqualTo("user");
    }
}
```

❶

Register a `RestTemplate` for the context of the auto-configured test slice

❷

Mock the behavior of a `RestTemplate` request to the specified URL

In [Example 4-15](#) we see a unit test annotated with `@RestClientTest`. Here we specify that we will be targeting the `UserService` class and that we would like a `RestTemplate` to be registered as a part of the auto-configured test slice. In the test method `getAuthenticatedUserShouldReturnUser()` we use the `MockRestServiceServer` field – named `server` – to mock the expected behavior of the intended HTTP request made by a `RestTemplate` inside the `UserService`. Notice how the expected JSON response is loaded from a class path resource – from `user.json`.

After using `MockRestServiceServer` to mock the HTTP response from the remote service, any usage of `RestTemplate` that matches the stated expectation will instead return the contents of `user.json`. The beauty of this is that we are able to target parts of the system under test by mocking the behavior of external services. This is a nifty feature that especially helps us test microservices that must coordinate with other services using HTTP. The question then becomes, how do we create integration tests that mocks the actual behavior of a remote HTTP service – instead of using `MockRestServiceServer`? This is the subject of *Consumer-driven Contract Testing*—which allows us to download stubs published by other services for the purpose of mocking service behavior for integration testing.

End-to-end Testing

End-to-end testing is an important part of ensuring that release components of a distributed application can be changed without breaking consumers. Further – a microservice architecture is a composition of many different services – with each application likely being connected to multiple services in an ensemble of organized chaos that will require a sophisticated testing strategy. In this section, we will cover a few techniques for testing distributed cloud-native applications.

End-to-end testing focuses on validating the functionality of an application's business features. As opposed to integration testing, end-to-end testing focuses on testing features from the perspective of a user. For example, let's assume that a user of an application would like to register a new account. There is a sequence of actions that the user must perform to register a new account successfully. In most cases, the user interface of the application becomes the conduit for these activities, generating events that interact with backend APIs to accomplish the result of the workflow. If we take inventory of each event that is produced by the actions of a user who is registering a new account, we can translate the sequence of events into an end-to-end test.

There are multiple types of end-to-end tests, depending on what part of the system is under test. For backend developers who are building microservices, the end-to-end test for registering a new account will likely be a composite of API interactions between separate microservices – depending on the workflow. For end-to-end workflows that orchestrate a business feature that touches different microservices, managing state becomes an important consideration.

Testing Distributed Systems

In software, when we talk about consistency, we are usually referring to state. When developing applications in a distributed system, it becomes necessary to know exactly how state is shared and replicated across an architecture. One of my favorite explanations of state goes all the way back to the birth of computing. Alan Turing first wrote about computing state in his seminal paper entitled *On computable numbers, with an application to the Entscheidungsproblem.*¹ In the paper, Turing describes state using a thought experiment that theorizes how a machine can determine whether or not a number is computable. Turing's thought experiment introduces the concept of a *Turing Machine*—an imaginary mechanical device that is capable of computing numbers using a table of states.

The machine that Turing describes consists of an unbounded tape of squares containing symbols that span out in only two directions. The analogue that Turing uses to describe his machine is loosely based on that of a type writer. The *machine* is allowed to scroll either left or right on the tape, scanning only one symbol at a time. The machine also has a table of states, mapping a symbol to a condition which instructs the machine what to do next.

Let's look at an example. Let's assume Turing's machine is in an initial state of A . The machine's first scanned symbol from the tape is 0 . The machine then looks up its instructions from the state table for its current state, which is still A . The instructions describe a condition that for the scanned symbol that is 0 to write a 1 and then to move right on the tape. Before moving to the next position, the final instruction is a state change—from A to B . The state B will now contain a different set of instructions, depending on the scanned symbol at the next position. Now the machine may again scan a 0 from its new position – and because it is in state B – it will still write a 1 , but this time the machine is instructed to move *left* on the tape instead of right.

Turing reduces to the essence what it means to program the behavior of his machine by switching between instructions that react and change based on inputs—outputting data in the process.

When we talk about state in modern applications, we are usually referring to a status field on a database record – represented as a column in a table. Let's take for example a user of a website who is registering a new account. The status of the user will transition from one state to another depending on the user's inputs. If the user has just finished registering for an account by submitting a form on a web page, the provided form fields will be processed and a record will be persisted to the database with the user registration details. One of the fields of this new user record is `status`, and can contain one of multiple values at any one time that describe the current state of a user.

Table 4-2. User records containing a status field

| <code>first_name</code> | <code>last_name</code> | <code>email</code> | <code>status</code> |
|-------------------------|------------------------|------------------------------------------------------------|---------------------|
| Bob | Dylan | bdylan@example.com | PENDING |
| Taylor | Swift | tswift@info.com | CONFIRMED |
| Tracy | Chapman | tchapman@test.com | ARCHIVED |
| Bruno | Mars | bmars@example.com | INACTIVE |

In [Table 4-2](#) we see multiple rows of a database table containing user records. Each user in the table has a different value for the `status` column. For the user named *Bob Dylan*, we see the user's status is currently `PENDING`. Now, the behavior of the application will change from the user's perspective, depending on the value of the `status` column. For the `PENDING` status—before the user can sign in to the application, they are required to confirm their listed e-mail address. After a user confirms the e-mail, the user's status will then transition to `CONFIRMED`, allowing the user to sign in and access other features of the application. If the user's status is set to `ARCHIVED` or `INACTIVE`, they may be unable to sign in—depending on the requirements of the application. This workflow is often used in applications that implement a form of user authentication.

The problem that arrives with distributed systems development—and more specifically in microservices—is that workflows like the one described above won't be isolated within the boundaries of a single application. When we break up a monolithic application architecture into many separate services, network partitions will bisect workflows that would otherwise manage data within the context of a single transaction. So what does this mean for the developer?

The main problem with distributed applications is that state must be observed globally across separate machines, using information that is exchanged over a network to communicate state. For a single monolithic application that is connected to a single database, such as a relational database like MySQL, state can be consistently replicated across separate instances of the database—making it highly available. By storing a record containing state across multiple machines we gain the benefit of being able to scale out and handle increased traffic.

The downside of distributing state is best summarized in the following question:

How do we make sure that whenever we read a replicated record from a pool of machines that it is consistent with the state of a record stored on other machines in the pool?

The problem of distributing state and ensuring consistency is one of the hardest problems in computer science. Within the boundaries of a single machine and process it can be quite simple to maintain the consistency of data – because whenever we read from memory there is a single address referencing where that data is stored. If one thread locks the reference at the address with intention to change the state of a record, other threads will observe the lock before proceeding to modify the state of the record at that reference. When there is a single source of information for the state of a record, it can be globally accessed without the state of the record becoming inconsistent. This changes when state is replicated across the boundary of a single machine—where all replicas must be updated together and read together. If one of the replicas is different than the others, the state of the record is *inconsistent*, which can cause the behavior of the application to flip-flop between different states stored on different partitions.

When testing cloud-native applications that are distributing state across separate microservices, it becomes absolutely essential to understand how to design end-to-end tests that test for data consistency. Eventual consistency is the best we can hope for when sharing state across the boundaries of separate microservices. The key concern is to design testing conditions that make sure that state is always eventually consistent—maybe not consistent at the same exact time—but *always* eventually consistent without manual intervention. Where strong consistency is required, do not share state between microservices—a simple rule of thumb for knowing when to avoid distributed transactions.

Another concern for microservice architectures is the manner in which we test integrations between different applications. For this kind of testing we may normally be required to execute end-to-end tests in an integration environment—where all applications and dependencies are bootstrapped for runtime execution—mirroring the deployment model of production. As you might imagine, the cost in time associated with bootstrapping a complete end-to-end environment for integration testing a single microservice is expensive.

It can be greatly beneficial for a developer to be able to execute integration tests with many microservice dependencies as fast as possible—creating tighter feedback loops while also limiting resource contention in a shared build environment. One popular method for testing that helps to resolve some of these pains—and is quickly becoming a standard for testing microservices—is called *Consumer-driven Contract Testing*.

Consumer-driven Contract Testing

Consumer-driven Contract Testing (CDC-T) was first introduced by Ian Robinson in 2006 as a practice of using published contracts to assert and maintain expectations between consumers and producers while preserving loose coupling between services. In the article that was originally published on Martin Fowler's website, Robinson describes service evolution in an SOA through the use of consumer-driven contracts.² In the years that followed, the practice and patterns of consumer-driven contracts introduced by Robinson were adapted to be used for testing the expectations between microservices.

The central premise of CDC-T is to allow producers and consumers in a microservice architecture to publish stubs in the form of a consumer-driven contract.

- There is no sharing of libraries between microservices at runtime.
- Services may mutually share libraries during integration testing—in the case of a cyclical dependency between microservices.
- The producer first publishes stubs by defining a consumer-driven contract in the form of an integration test.
- Consumers are able to mock a producer by downloading its versioned stubs from a repository
- Producers share stubs generated through a contract definition, but do not share types or client libraries.
- CDC-T should always strive to hide the implementation details of a producer's API.

Spring Cloud Contracts

Spring Cloud Contracts is an open source project in the Spring ecosystem that implements a variant of consumer-driven contracts for the sole purpose of integration testing distributed applications—such as microservices. Contracts is an exciting edition to the Spring family of projects, providing Spring Boot applications the ability to publish, simulate, and mock remote services using stubs that are generated from unit tests.

Example 4-16. The DSL describes expectations between a producer and consumer

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/uaa/v1/me'
        headers {
            header('Content-Type'): consumer(regex('application/*j
        }
    }
    response {
        status 200
        body([
            username      : value(producer(regex('[A-Za-z0-9]+'))
            firstName     : value(producer(regex('[A-Za-z]+'))))
            lastName      : value(producer(regex('[A-Za-z]+'))),
            createdAt     : value(producer(regex('[0-9]+'))),
            lastModified: value(producer(regex('[0-9]+'))),
            id           : value(producer(regex('[0-9]+'))))
        ])
        headers {
            header('Content-Type': value(
                producer('application/json; charset=UTF-8'),
                consumer('application/json; charset=UTF-8')))
        }
    }
}
```

Continuous Integration

- Continuous integration testing options for microservices
- Orchestrating end-to-end tests

Functional Testing

- Writing JS BDD [with Pivotal's Jasmine](#)
- Building Client Side Testing [with Phantom.js](#) and Fluentlenium

Behavior-driven Testing

- [Cucumber JVM](#)
- [Cucumber InfoQ Presentation](#)
- [SpringSource webinar](#)
- Cucumber JVM for feature tests in he given/when/then style (which would use Selenium)
- A/B testing (Asgard || Spinnaker)

¹ On computable numbers, with an application to the Entscheidungsproblem. London Mathematical Society. 1937.

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

² Consumer-Driven Contracts: A Service Evolution Pattern. Ian Robinson. 2006. <http://martinfowler.com/articles/consumerDrivenContracts.html>

Chapter 5. REST APIs

Amazon.com famously decreed that all services should be exposed as APIs, and that no state should be shared at the database tier. This is an important first step in making the move to microservices: everything is an API.

Representational State Transfer (REST) is far and away the most popular protocol of the web's millions of APIs.

REST was originally advanced by Dr. Roy Fielding as part of his doctoral dissertation in 2001. Fielding helped define the HTTP 1.1 specification and wanted to help illustrate how the web - an already proven, massively scalable, decentralized, failure resistant fabric - could be used to build services. HTTP is an *existence proof* of the REST architecture's merits.

Where previous approaches to distributed services (like CORBA, EJB, RMI, and SOAP) focused more or less on exposing an object-oriented interface and methods as a remotely accessible service (RPC), REST instead focuses on the manipulation of remote *resources* or entities. Nouns, not verbs.

REST is all about exposing the mutations of business state with the verbs (GET, PUT, POST, DELETE, etc.) and idioms (HTTP headers, status codes, etc.) that HTTP already gives us to support service-to-service communication. The best REST APIs tend to exploit more of HTTP's capabilities. REST, for all of its benefits, is a architectural constraint on HTTP, not any sort of *standard*, and so we've seen a proliferation of APIs of varying degrees of compliance with RESTful principles. To counter this, Dr. Leonard Richardson put forth his [maturity model](#) to qualify how RESTful an API is.

Leonard Richardson's Maturity Model

Level 0 - the Swamp of POX (naturally, it *would* be zero-indexed!) describes APIs that use HTTP as a transport and nothing more. SOAP-based web services, for example, use HTTP, but they *could* as easily use JMS. These are incidentally HTTP-based. Such a service uses HTTP mainly as a tunnel through one URI. SOAP and XML-RPC are examples. They usually use only one HTTP verb.

Level 1 - Resources describes APIs that use multiple URIs to distinguish related nouns.

Level 2 - HTTP verbs describes APIs that leverage transport native properties (like HTTP verbs and status codes) to enhance the service. If you do everything wrong with Spring MVC, or even JAX-RS, you'll *still* probably end up with an API that's level 2 compliant!

Level 3 - Hypermedia Controls (HATEOAS) describes APIs that require no *a priori* knowledge of a service in order to navigate it. Such a service promotes longevity through a uniform interface for interrogating a service's structure.

In this chapter, we'll look at how to use Spring to build REST services, starting at level 2. Later, we'll look at how to use hypermedia to promote uniform, self-describing services. REST APIs are meant to represent the HTTP contract between services in the system. API clients will use them, but humans must develop against them and so we will pay particular attention to developing approachable, correctly and consistently documented APIs, as well.

REST's an important part of a cloud-native application. While nothing about microservices stipulates or even requires REST, it is the most common approach to expose services.

HTTP lends itself to cloud-native applications. A modern Platform-as-a-Service (PaaS), such as Cloud Foundry, is specifically HTTP-aware, though it may not be TCP/UDP-aware. HTTP is an ideal fit for service caching because it has no client-side state. Instead, each request is self-contained. This also implies that services can be horizontally scaled easily so long as the state represented by the REST API can also scale horizontally.

Caching reduces time-to-first-byte times. HTTP is content-type agnostic and includes support for content-negotiation; one client may support only XML, another JSON or Google's Protocol Buffers. All may be served by the same services. This mechanism is extensible, too. You can squeeze a lot of performance out of individual HTTP requests with caching, GZip compression and - with HTTP 2 on the horizon - this situation looks to improve considerably in the near term.

Simple REST APIs with Spring MVC

Add `org.springframework.boot:spring-boot-starter-web` to your build to bring in Spring MVC. Spring MVC is a request/response-oriented HTTP framework. In Spring MVC, *controller* handler methods are mapped to HTTP requests and ultimately provide responses. Let's look at a REST API designed to manipulate `Customer` entities.

Example 5-1. Our first `@RestController`, `CustomerRestController`

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.method.annotation.MvcU
import org.springframework.web.servlet.support.ServletUriComponen

import java.net.URI;
import java.util.Collection;

@RestController
@RequestMapping("/v1/customers")
public class CustomerRestController {

    @Autowired
    private CustomerRepository customerRepository;

    ①
    @RequestMapping(method = RequestMethod.OPTIONS)
    ResponseEntity<?> options() {
        return ResponseEntity
            .ok()
            .allow(HttpMethod.GET, HttpMethod
                HttpMethod.OPTION
            .build();
    }
}
```

```

    @RequestMapping(method = RequestMethod.GET)
    ResponseEntity<Collection<Customer>> getCollection() {
        return ResponseEntity.ok(this.customerRepository.
    }

②
    @RequestMapping(value = "/{id}", method = RequestMethod.G
    ResponseEntity<Customer> get(@PathVariable Long id) {
        return this.customerRepository.findById(id).map(R
            .orElseThrow(() -> new CustomerNo
    }

    @RequestMapping(method = RequestMethod.POST)
    ResponseEntity<Customer> post(@RequestBody Customer c) {

        Customer customer = this.customerRepository.save(
            .getFirstName(), c.getLastName())

        URI uri = MvcUriComponentsBuilder.fromController(
            .path("/{id}").buildAndExpand(cus
        return ResponseEntity.created(uri).body(customer)
    }

④
    @RequestMapping(value = "/{id}", method = RequestMethod.D
    ResponseEntity<?> delete(@PathVariable Long id) {
        return this.customerRepository.findById(id).map(c
            customerRepository.delete(c);
            return ResponseEntity.noContent().build()
        ).orElseThrow(() -> new CustomerNotFoundException
    }

⑤
    @RequestMapping(value = "/{id}", method = RequestMethod.H
    ResponseEntity<?> head(@PathVariable Long id) {
        return this.customerRepository.findById(id)
            .map(exists -> ResponseEntity.noC
            .orElseThrow(() -> new CustomerNo
    }

⑥
    @RequestMapping(value = "/{id}", method = RequestMethod.P
    ResponseEntity<Customer> put(@PathVariable Long id, @Requ
        return this.customerRepository
            .findById(id)
            .map(existing -> {

```

```

        Customer customer = this.
            .save(new

                URI selfLink = URI.create
                    .fromCurr
                return ResponseEntity.cre
            }).orElseThrow(() -> new Customer

        }
    }

```

❶

a handler method is designated with the `@RequestMapping` annotation. A class may contain a `@RequestMapping` annotation, in which case the method-level declarations override or add to the root, class-level mappings. The handler responds to the `OPTIONS` HTTP verb, which a client will issue when it wants to know what other HTTP verbs are supported for a given resource.

❷

this handler method returns individual records whose IDs are encoded as *path variables* in the `@RequestMapping` URI syntax: `{id}`.

❸

data may be transferred from the client to the service and the body of the data passed to the service as the handler's `@RequestBody`.

❹

the `DELETE` handler returns an HTTP 204 on success, or it throws an exception which ultimately results in a 404 on failure.

❺

the `HEAD` handler is meant only to confirm the existence of the resource, so it returns with a 204 on success, or it throws an exception which ultimately results in a 404 on failure, just as in the `DELETE` handler.

❻

the `PUT` handler is charged with updating an existing record. It uses `@PathVariable` parameters to specify which record to update. If the record does not exist, an exception that ultimately yeilds a 404 is thrown.

This is a basic Spring MVC REST controller. The `@RequestMapping` annotation maps handler methods to specifications of types of HTTP requests. The `@RequestMapping` annotation lets us further discriminate by headers in the request, by content-type sent and returned, by cookies, and more. The path specified is relative to the root of the application's context-path *unless* a controller-level `@RequestMapping` annotation specifies a path, in which case the handler-level path is relative to the path in the controller.

Content Negotiation

The handler methods return objects or, sometimes, the `ResponseEntity` envelope with objects as a payload. When Spring MVC sees a return value it uses a strategy object, `HttpMessageConverter`, to convert the object into a representation suitable for the client. The client specifies what kind of representation it expects using *content-negotiation*. By default, Spring MVC looks at the incoming request's `Accept` header for a media-type, like `application/json`, `application/xml`, etc. - that a configured `HttpMessageConverter` is capable of creating, given the object and the desired media type. This same process happens in reverse for data sent from the client *to* the service and passed to the handler methods as `@RequestBody` parameters.

Content-negotiation is one of the most powerful features of HTTP: the same service may service clients that accept different protocols. We use media types to signal to the client the type of content being served. Normally, a media type *also* signals how the client is to process the content of the response. A client knows, for example, not to bother trying to extract JSON strings out of a response with a media type of `image/jpeg`.

Reading and Writing Binary Data

Thus far we've looked at JSON, but there's no reason REST resources couldn't serve media like images, or raw files. Let's look at how to read and write image data for a profile endpoint, `/customers/{id}/photo`.

Example 5-2. Reading and writing binary (image) data

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.util.FileCopyUtils;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.net.URI;
import java.util.concurrent.Callable;

import static org.springframework.web.servlet.support.ServletUriC

@RestController
@RequestMapping(value = "/customers/{id}/photo")
public class CustomerProfilePhotoRestController {

    private File root;

    @Autowired
    private CustomerRepository customerRepository;

    private Log log = LogFactory.getLog(getClass());

    @Autowired
    void setUserHome(
```

```

        @Value("${upload.dir:${user.home}}/images")
this.root = new File(uploadDir);
Assert.isTrue(
            this.root.exists() || this.root.m
String.format("The path '%s' must
                this.root.getAbso

}

❶
@RequestMapping(method = RequestMethod.GET)
 ResponseEntity<Resource> read(@PathVariable Long id) {
    return this.customerRepository
        .findById(id)
        .map(customer -> {
            File file = fileFor(custo
            Assert.isTrue(
                file.exists()
                String fo

Resource fileSystemResour
return ResponseEntity.ok(
    .contentT
    .body(fil
})).orElseThrow(() -> new Customer
}

❷
@RequestMapping(method = {RequestMethod.POST, RequestMethod.
Callable<ResponseEntity<?>> write(@PathVariable Long id,
    @RequestParam MultipartFile file) ❸
    throws Exception {
    log.info(String.format("upload-start /customers/%
        id, file.getSize()));
    return () -> this.customerRepository
        .findById(id)
        .map(customer -> {
            File fileForCustomer = fi
            try (InputStream in = fil
                OutputStr
                FileCopyUtils.cop
} catch (IOException ex)
    throw new Runtime
}

```

```

        URI location = fromCurren
                      .toUri();
        log.info(String.format(
                      "upload-f
                      location)
        return ResponseEntity.cre
    )).orElseThrow(() -> new Customer
}

private File fileFor(Customer person) {
    return new File(this.root, Long.toString(person.g
}
}

```

❶

Spring MVC automatically returns the `byte[]` data from the buffer - a file, URL resource, `InputStream`, etc. - underlying a `org.springframework.core.io.Resource` returned from a Spring MVC controller handler method.

❷

file uploads may block and monopolize the `Servlet` container's threadpool. Spring MVC *backgrounds* `Callable<T>` handler method return values to a configured `Executor` thread pool and frees up the container's thread until the response is ready.

❸

Spring MVC will automatically map multipart file upload data to a `org.springframework.web.multipart.MultipartFile` request parameter

❹

it's good form when creating a resource to return a HTTP 201 (`Created`) status code and return a `Location` header with the URI of the newly created resource.

Reading media is easy if the data can be represented with one of Spring's `org.springframework.core.io.Resource` implementations. There are many implementations provided out of the box: `FileSystemResource`,

`GridFsResource` (which is based on MongoDB's Grid File System), `ClassPathResource`, `GzipResource`, `VfsResource`, `UrlResource`, etc. Spring MVC automatically returns the `byte[]` data from the buffer underlying a `org.springframework.core.io.Resource` to the client.

Spring MVC applications run in a Servlet container. The Servlet container itself maintains a thread pool that is used to respond to incoming HTTP requests; each time an incoming HTTP request arrives a new thread is dispatched to produce a reply. It's important to not monopolize the threads in the Servlet container's thread pool. Spring MVC will move the flow of execution of controller handler method to a separate thread pool - one provided by specifying a `TaskExecutor` bean - if a controller handler method returns a `java.util.concurrent.Callable<T>`, an `org.springframework.web.context.request.async.DeferredResult`, or a `org.springframework.web.context.request.async.WebAsyncTask`.

A `Callable<T>` is simply a specialization of a `Runnable` that returns a result. Spring MVC invokes the `Callable` on the specified thread pool and, once the callable's produced a result, returns the result asynchronously as the reply to the original HTTP request.

A `WebAsyncTask` is basically the same thing; it wraps a `Callable<T>` but also provides fields to specify the `TaskExecutor` on which to run the `Callable<T>` and a timeout period after which a response will be committed. A `DeferredResult` doesn't trigger execution elsewhere; instead, Spring MVC returns the Servlet container thread to the pool and only triggers an async response when the `DeferredResult# setResult` method is called. `DeferredResult` instances may be cached and later updated in an out of band thread of execution.

The example handles the write in a separate thread; it assumes that writing may be a slower operation than reading, which could benefit from caching. The example moves the upload, the write, to a separate thread by handling it in a `Callable<T>`. This is an efficient approach from the perspective of the service, but it is potentially ineffective from the perspective of the *client*. The client will block, waiting for the reply, for as long as the `Callable<T>` is executing.

An alternative approach is to return *immediately*, returning an HTTP 202 Accepted status code and providing a `Location` header that either indicates where the newly created resource lives or at least where information about the status of its creation exists.

Using Google Protocol Buffers

Content-negotiation lets us optimize, where possible, but provide fallback support where not possible. A common example of this is when using Google Protocol Buffers. Google Protocol Buffers are an ideal choice for efficient, cross-platform communication. If the requesting client doesn't support Google Protocol Buffers, Spring can downgrade to JSON or XML thanks to content-negotiation. Smarter clients get smarter responses, but everybody can play.

Google Protocol Buffers is a highly efficient serialization format. Google Protocol Buffers messages carry no information to describe the *structure* of the message, only the data in the message. The *structure* is implied by the Google Protocol Buffer .proto schema definition.

Example 5-3. Google Protocol Buffers schema for a `Customer` message.

```
package demo;

option java_package = "demo"; ❶

option java_outer_classname = "CustomerProtos"; ❷

message Customer { ❸
    optional int64 id = 1;
    required string firstName = 2;
    required string lastName = 3;
}

message Customers { ❹
    repeated Customer customer = 1;
}
```

❶

the language-specific package or module for the resulting types

❷

Google Protocol Buffer types can be nested inner classes if you specify the `java_outer_classname`; the resulting type will be `demo.CustomerProtos.Customer`

③

the schema for a single `Customer`

④

the schema for a *collection* of `Customer`

Fields in the definition are assigned an integer offset to help the compiler understand what it is looking at in a datastream of binary data.

If a client knows about an older definition of a message, and a service replies with a more detailed message, the client will still be able to interoperate so long as the offsets are stable. This is ideal in a distributed system where clients and services may not always have the same *version* of a message. Independent deployability, the ability to deploy one service without a *flag day* deploy of others, is a very important feature of microservices. It lets teams iterate and evolve their services without constant synchronization overhead. Google Protocol Buffers supports this loose coupling.

The `protoc` compiler will generate language-specific (Java, Python, C, .NET, PHP, Ruby, and a [large number of other technologies besides](#)) bindings to read and write messages described by a `.proto` definition. Here, we'll look a simple script to generate Java, Python, and Ruby bindings for our `Customer` Google Protocol Buffer definition.

Example 5-4. a script to generate the required Java, Ruby and Python clients.

```
#!/usr/bin/env bash

SRC_DIR=`pwd`
DST_DIR=`pwd`/../src/main/

echo source:          ${SRC_DIR}
echo destination root: ${DST_DIR}

function ensure_implementations() {
```

```

gem list | grep ruby-protocol-buffers || sudo gem install rub
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
}

function gen() {
D=$1
echo $D
OUT=$DST_DIR/$D
mkdir -p $OUT
protoc -I=$SRC_DIR --$D_out=$OUT $SRC_DIR/customer.proto
}

ensure_implementations

gen java
gen python
gen ruby

```

Spring MVC ships with a custom `HttpMessageConverter` implementation that can read and write objects created using the generated Java bindings for our `.proto` definition. The `protoc` compiler will emit bindings for our Google Protobuf compiler that we can use when handling REST requests. Let's look at a REST API built to handle `application/x-protobuf`. Most of this service will seem very familiar! Indeed, the main difference between this and what you've seen before, besides the URI path, is the additional busy work imposed from having to convert from a `Customer` to a Protobuffer-specific `CustomerProtos.Customer` DTO and back.

Example 5-5. A REST service that leverages content-negotiation to serve content with the media type `application/x-protobuf`.

```

package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.MvcU

import java.net.URI;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

import static org.springframework.web.servlet.support.ServletUriC

```

```

@RestController
@RequestMapping(value = "/v1/protos/customers")
public class CustomerProtobufRestController {

    @Autowired
    private CustomerRepository customerRepository;

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    ResponseEntity<CustomerProtos.Customer> get(@PathVariable Long id) {
        return this.customerRepository.findById(id)
            .map(this::fromEntityToProtobuf).map(ResponseEntity::ok)
            .orElseThrow(() -> new CustomerNotFoundException());
    }

    @RequestMapping(method = RequestMethod.GET)
    ResponseEntity<CustomerProtos.Customers> getCollection() {
        List<Customer> all = this.customerRepository.findAll();
        CustomerProtos.Customers customers = this.fromCollectionToProtobuf(all);
        return ResponseEntity.ok(customers);
    }

    @RequestMapping(method = RequestMethod.POST)
    ResponseEntity<CustomerProtos.Customer> post(
        @RequestBody CustomerProtos.Customer c) {

        Customer customer = this.customerRepository.save(new Customer(
            c.getFirstName(), c.getLastName()));

        URI uri = MvcUriComponentsBuilder.fromController(getClass())
            .path("/{id}").buildAndExpand(customer.getId()).toUri();
        return ResponseEntity.created(uri).body(
            this.fromEntityToProtobuf(customer));
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    ResponseEntity<CustomerProtos.Customer> put(
        @PathVariable Long id,
        @RequestBody CustomerProtos.Customer c) {

        return this.customerRepository
            .findById(id)
            .map(existing -> {

                Customer customer = this.customerRepository
                    .save(new Customer(existing.getId(),
                        c.getFirstName(), c.getLastName()));

                URI selfLink = URI.create(fromCurrentRequest()

```

```

        return ResponseEntity.created(selfLink).body(
    )).orElseThrow(() -> new CustomerNotFoundException
}

private CustomerProtos.Customers fromCollectionToProtobuf(
    Collection<Customer> c) {
    return CustomerProtos.Customers
        .newBuilder()
        .addAllCustomer(
            c.stream().map(this::fromEntityToProtobuf
                .collect(Collectors.toList()))).bu
}

private CustomerProtos.Customer fromEntityToProtobuf(Customer
    return fromEntityToProtobuf(c.getId(), c.getFirstName(),
        c.getLastName());
}

private CustomerProtos.Customer fromEntityToProtobuf(Long id,
    String l
    CustomerProtos.Customer.Builder builder = CustomerProtos.
        .newBuilder();
    if (id != null && id > 0) {
        builder.setId(id);
    }
    return builder.setFirstName(f).setLastName(l).build();
}
}

```

In order for all of this to work, we need to *teach* Spring MVC about the new content-type by registering a custom `HttpMessageConverter` implementation:

Example 5-6. Register a custom `HttpMessageConverter` implementation to go to `application/x-protobuf` and back.

```

package demo;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.protobuf.ProtobufHttpMe
@Configuration

```

```

class GoogleProtocolBuffersConfiguration {

    @Bean
    ProtobufHttpMessageConverter protobufHttpMessageConverter
        return new ProtobufHttpMessageConverter();
    }

}

```

The `RestTemplate` supports the same content-negotiation mechanism as the service; to *read* `application/x-protobuf` data, specify a `ProtobufHttpMessageConverter` on the `RestTemplate`. Google Protocol Buffers are cross-language and cross-platform. Use the `protoc` client to generate bindings for as many languages as you like.

Example 5-7. A Python-language Google Protocol Buffer client; `customer_pb2` is the client generated by `protoc`

```

#!/usr/bin/env python

import urllib
import customer_pb2

if __name__ == '__main__':
    customer = customer_pb2.Customer()
    customers_read = urllib.urlopen('http://localhost:8080/customers')
    customer.ParseFromString(customers_read)
    print customer

```

Example 5-8. A Ruby-language Google Protocol Buffer client; `/customers.pb` is the client generated by `protoc`

```

#!/usr/bin/ruby

require './customer.pb'
require 'net/http'
require 'uri'

uri = URI.parse('http://localhost:8080/customers/3')
body = Net::HTTP.get(uri)
puts Demo::Customer.parse(body)

```

Error Handling

Various handlers in our `CustomerRestController` operate on existing records and - if that record is not found - throw an exception. These handlers could explicitly return a `ResponseEntity` with a 404 status, but this sort of error handling quickly becomes repetitive. Centralize error handling logic. Spring MVC supports listening for, and responding to, error conditions in Spring MVC controllers with handler methods annotated with `@ExceptionHandler`. Normally, `@ExceptionHandler` handlers live in the same controller component as the handlers that may throw exceptions. A `@ControllerAdvice` component is a special type of component that may introduce behavior (and respond to exceptions) for any number of controllers. They are a natural place to stick centralized `@ExceptionHandler` handlers.

Errors are an important part of an effective API. Errors are most useful for humans, who must ultimately resolve or at least understand the implications of the error. It's paramount that errors be as helpful as possible. HTTP status codes are many things, but they are *not* particularly approachable. A common practice is to send back an error code along with some sort of representation of the error and a human readable error message. There is no official standard on the way errors should be encoded, beyond HTTP status codes, but a *de-facto* standard is to use the `application/vnd.error` content type. Spring HATEOAS provides easy support for this content-type in `VndError` and `VndErrors`.

Let's look at a simple `@ControllerAdvice` that intercepts all exceptions thrown and properly handles them by sending back an HTTP status *and* a `VndError`.

Example 5-9. Centralized error-handling with a `@ControllerAdvice`.

```
package demo;

import org.springframework.hateoas.VndErrors;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
```

```

import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;

import java.util.Optional;

@ControllerAdvice(annotations = RestController.class)
public class CustomerControllerAdvice {

    ①
    private final MediaType vndErrorMediaType = MediaType
        .parseMediaType("application/vnd.error");

    ②
    @ExceptionHandler(CustomerNotFoundException.class)
    ResponseEntity<VndErrors> notFoundException(CustomerNotFo
        return this.error(e, HttpStatus.NOT_FOUND, e.getc
    }

    @ExceptionHandler(IllegalArgumentException.class)
    ResponseEntity<VndErrors> assertionException(Illega
        return this.error(ex, HttpStatus.NOT_FOUND, ex.ge
    }

    ③
    private <E extends Exception> ResponseEntity<VndErrors> e
        HttpStatus httpStatus, String logref) {
        String msg = Optional.of(error.getMessage()).orEl
            error.getClass().getSimpleName());
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.setContentType(this.vndErrorMediaType
        return new ResponseEntity<>(new VndErrors(logref,
            httpStatus);
    }
}

```

①

there's no inbuilt media type for application/vnd.error, so we create our own

②

we have two handlers in this class, and they both respond to possible

exceptions thrown in other components. This handler, specifically, responds to our `CustomerNotFoundException`, which in turn just extends `RuntimeException`.

③

the `error` method builds up a `ResponseEntity` containing a `VndErrors` payload and conveys the media type and status code desired. Simple *and* useful.

Hypermedia

The API as built will work just fine *if* a client knows which endpoint to call, when to call it, and with what HTTP verb to make the call. The HTTP OPTIONS verb helps with the last bit: understanding which HTTP verbs work for a given resource. The rest of it, however, is implied and - one hopes - well documented somewhere. The trouble with documentation is that few keep it up to date and fewer still bother to read it. It ultimately drifts out of sync with the living specification of the service as defined in the code.

Dr. Fielding's thesis features heavily the *hypermedia tenent*. It refers to the idea that links - `<a/>` elements in the markup - in everyday HTML resources provide information to the client - HTML browsers and their users - that ultimately lead to changes in application state. You may start your session at Amazon.com, enter a search, find a satisfying product to purchase, click to add it to the shopping cart, choose to checkout and then pay, all by way of navigations from one HTTP resource to another. Ultimately, you've changed system state. These links tell you where to go and - because they only appear when they're relevant (there's no refund link for products not yet paid for!), they imply *when* to go. These steps from one resource to another imply a protocol - a series of steps and interactions required to achieve a goal in software. This system works because we humans can parse the links, extract them from the user experience and imply a sequence of steps based on visual cues, from the site's design to understand their relevance.

Machine clients, on the other hand, aren't so clever as humans. We need to simplify things a bit. Confusingly, HTML provides another link element, `<link/>`, which supports exactly this use case. These `<link/>` elements aren't navigable for humans - you can't *click* them. But they're there and the browser client knows what to do with them. These links have two important attributes: `rel` and `href`. There are many applications for this element, but the most common application is in loading stylesheet data:

Example 5-10. loading a document's stylesheet

```
<link rel="stylesheet" href="bootstrap.css">
```

These links provide metadata about the resource they're included in. The browser looks first at the `rel` to understand what sort of resource is being linked. It optionally follows the `href` to load the linked resource. The client uses the `rel` attribute to make a determination about the relevance of the linked resource. The linked resource can live anywhere! The browser doesn't make any assumptions about their locations *a priori* - it simply follows the `href` in the `<link/>` if one is available. The client is thus *decoupled* from the location of the resource.

The client (the browser) can still use the API even if the location of the resources in the API change. The client can determine which resource to follow by examining the `rel` attribute to understand the linked resource's *relevance*. The `rel` becomes the contract: as long as that stays stable then the client will never break.

Let's revisit the protocol for checking out from an Amazon.com-like API.

Sure, the `<link/>` element is an XML element, but there's no reason the same approach - payloads accompanied by enriching links - couldn't work to our advantage in other encodings such as JSON.

There's even a *de facto* standard encoding for describing these link elements in JSON, [called HAL, or Hypertext Application Language](#), which is a specialization of JSON with a content type of `application/hal+json`. So, instead of spending time building ad-hoc structures to describe your REST resources, you can rely on HAL to implement HATEOAS.

Example 5-11. The earlier `Customer` REST resource described with HAL

```
{
  "id":1,
  "firstName":"Mark",
  "lastName":"Fisher",
  "_links": {
    "self": {"href":"http://localhost:8080/v2/customers/1"},
    "profile-photo": {"href":"http://localhost:8080/customers/1/p"}
  }
}
```

An API so described can be navigated by any compatible HAL client. Spring Boot supports a *very* convenient HAL client called the *HAL Browser*. You can add it to any Spring Boot web application by adding the `org.springframework.boot:spring-boot-starter-actuator` and `org.springframework.data:spring-data-rest-hal-browser` dependencies. It registers a Spring Boot Actuator endpoint, so you'll need the Actuator as well.

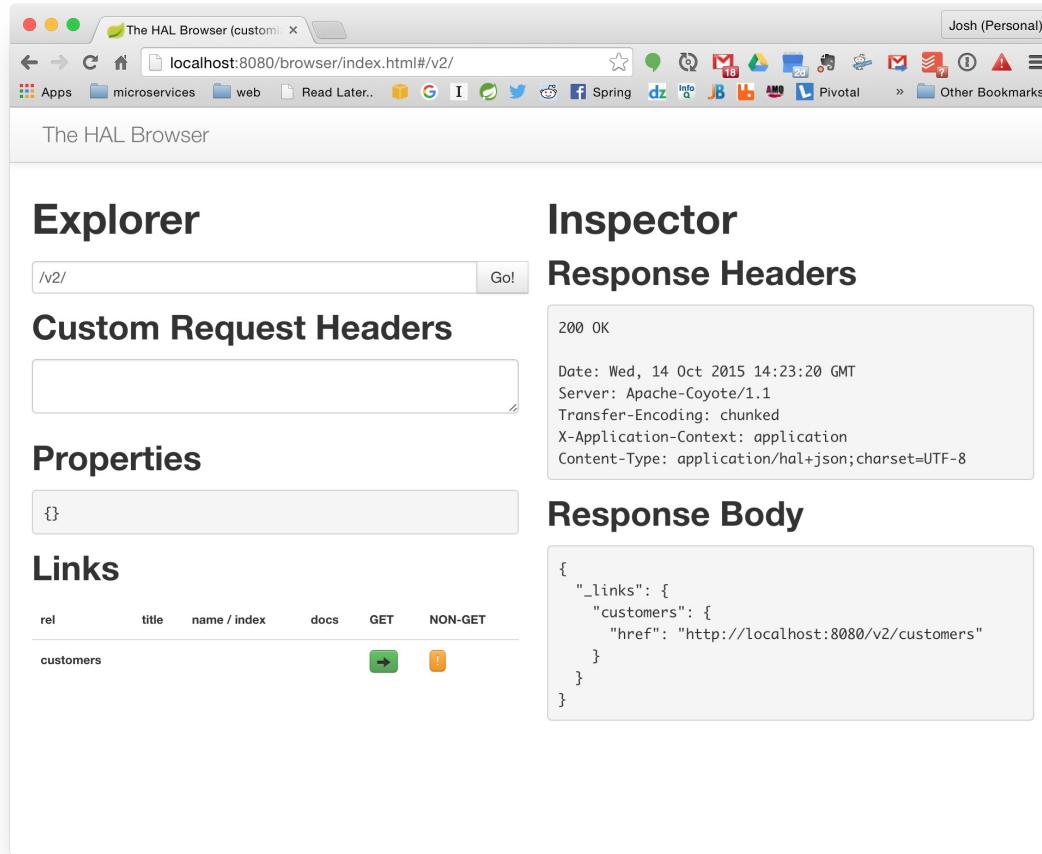


Figure 5-1. the default configured HAL browser Actuator endpoint

[Spring HATEOAS](#) sits on top of Spring MVC and provides the necessary plumbing to consume and describe resources in terms of payloads and their related links. It relies on Spring MVC fundamentals; instead of consuming and producing resources of type `T`, you consume and produce resources of type `org.springframework.hateoas.Resource<T>`, for single objects, or `org.springframework.hateoas.Resources<T>`, for collections of objects.

Resource is an envelope that contains a payload and a set of related links. You're going to convert objects of type `T` to objects of type `Resource<T>` or `Resources<T>` frequently. Instances of the `org.springframework.hateoas.ResourceAssembler` interface contract codify this conversion. Let's revisit our `CustomerRestController`, adding in hypermedia with Spring HATEOAS.

Example 5-12. A revised CustomerHypermediaRestController

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.MvcU
import org.springframework.web.servlet.support.ServletUriComponen

import java.net.URI;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping(value = "/v2", produces = "application/hal+json")
❶
public class CustomerHypermediaRestController {

    @Autowired
    private CustomerResourceAssembler customerResourceAssembl

    private final CustomerRepository customerRepository;

    @Autowired
    CustomerHypermediaRestController(CustomerRepository custo
        this.customerRepository = customerRepository;
    }

❷
@RequestMapping(method = RequestMethod.GET)
ResponseEntity<Resources<Object>> root() {
    Resources<Object> objects = new Resources<>(Colle
    URI uri = MvcUriComponentsBuilder
        .fromMethodCall(
            MvcUriComponentsB
                .build().toUri();
    Link link = new Link(uri.toString(), "customers")
    objects.add(link);
    return ResponseEntity.ok(objects);
}
}
```

④

```
@RequestMapping(value = "/customers", method = RequestMethod.GET)
ResponseEntity<Resources<Resource<Customer>>> getCollection() {
    List<Resource<Customer>> collect = this.customerRepository
        .stream().map(customerResourceAssembler::assemble)
        .collect(Collectors.<Resource<Customer>>toList());
    Resources<Resource<Customer>> resources = new Resources();
    URI self = ServletUriComponentsBuilder.fromCurrentRequest()
        .toUri();
    resources.add(new Link(self.toString(), "self"));
    return ResponseEntity.ok(resources);
}

@RequestMapping(value = "/customers", method = RequestMethod.POST)
ResponseEntity<Resource<Customer>> options() {
    return ResponseEntity
        .ok()
        .allow(HttpMethod.GET, HttpMethod.PUT, HttpMethod.DELETE)
        .build();
}

@RequestMapping(value = "/customers/{id}", method = RequestMethod.GET)
ResponseEntity<Resource<Customer>> get(@PathVariable Long id) {
    return this.customerRepository
        .findById(id)
        .map(c -> ResponseEntity.ok(this.customerResourceAssembler
            .toResource(c)))
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

@RequestMapping(value = "/customers", method = RequestMethod.POST)
ResponseEntity<Resource<Customer>> post(@RequestBody Customer customer) {
    Customer savedCustomer = this.customerRepository.save(
        customer);
    URI uri = MvcUriComponentsBuilder.fromController(CustomerController.class)
        .path("/customers/{id}").buildAndExpand(savedCustomer.getId())
        .toUri();
    return ResponseEntity.created(uri).body(
        this.customerResourceAssembler.toResource(savedCustomer));
}

@RequestMapping(value = "/customers/{id}", method = RequestMethod.DELETE)
ResponseEntity<Resource<Customer>> delete(@PathVariable Long id) {
    return this.customerRepository.findById(id).map(c -> {
        this.customerRepository.delete(c);
        return ResponseEntity.noContent().build();
    });
}
```

```

        return ResponseEntity.noContent().build()
    )).orElseThrow(() -> new CustomerNotFoundException
}

@RequestMapping(value = "/customers/{id}", method = Reqe
ResponseEntity<?> head(@PathVariable Long id) {
    return this.customerRepository.findById(id)
        .map(exists -> ResponseEntity.noC
        .orElseThrow(() -> new CustomerNo
}

@RequestMapping(value = "/customers/{id}", method = Reqe
ResponseEntity<Resource<Customer>> put(@PathVariable Long
    @RequestBody Customer c) {
    Customer customer = this.customerRepository.save(
        .getFirstName(), c.getLastName())
    Resource<Customer> customerResource = this.custom
        .toResource(customer);
    URI selfLink = URI.create(ServletUriComponentsBui
        .fromCurrentRequest().toUriString
    return ResponseEntity.created(selfLink).body(cust
}
}
}

```

①

responses are sent not as plain ol' application/json, but as a application/hal+json

②

we've injected a custom `ResourceAssembler` implementation to simplify conversion from `T` to `Resource<T>`.

③

the root endpoint simply returns a collection of `Link` elements that act as a sort of menu: it should be possible to start at / and navigate without any *a priori* knowledge.

④

the `GET` method on the `customers` collection is procedurally the same as before, but we've added a Spring HATEOAS `Link` to the resource. The `Link` is constructed using Spring MVC's handy

ServletUriComponentsBuilder.

The rest of this class is basically as before. Most of the methods in the CustomerHypermediaRestController lean on the injected CustomerResourceAssembler to do their work.

Example 5-13. the CustomerResourceAssembler provides a default set of links for a given Customer entity

```
package demo;

import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.ResourceAssembler;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.mvc.method.annotation.MvcU

import java.net.URI;

@Component
class CustomerResourceAssembler
    implements
        ResourceAssembler<Customer, Resource<Cust

    @Override
    public Resource<Customer> toResource(Customer customer) {

        Resource<Customer> customerResource = new Resourc
        URI photoUri = MvcUriComponentsBuilder
            .fromMethodCall(
                MvcUriComponentsB
                    C
                    C

        URI selfUri = MvcUriComponentsBuilder
            .fromMethodCall(
                MvcUriComponentsB
                    C
                    C

        customerResource.add(new Link(selfUri.toString()),
        customerResource.add(new Link(photoUri.toString())
        return customerResource;
    }
}
```

The `ResourceAssembler` is a natural place to conditionally include or exclude links based on entity state. In this example, for example, a check to see if there actually *is* a profile photo present might conditionally include the link for the profile photo. The links available describe the protocol of the client as it interacts with this system.

The hypermedia included in the results, along with the cues that HTTP (and the HTTP `OPTIONS` method) provide, mean that it is easy to navigate through this API without any *a priori* documentation.

HAL encoded hypermedia is but one of many popular options. 2009 saw the introduction of [Web Application Description Language](#) which is a broad-scope approach to describing HTTP services. [ALPS](#) debuted more recently and both defines possible state transitions *and* the attributes of the resources involved in those state transitions in a way that is media-type agnostic. Spring Data REST provides automatic support for ALPS.

Media Type and Schema

We've looked at using Spring HATEOAS to introduce hypermedia to a REST API. The goal is to give the client as much information as possible to be able to navigate the API without any *a priori* knowledge. HTTP already dictates the possible *verbs* - PUT, POST, GET, DELETE, etc. - and hypermedia tells us where we can go for a given resource.

HTTP provides the notion of a content-type. In theory, the content type provides enough information for any client to know what the content being served is and how to manipulate it. In practice, it's not sufficient. The content type of `image/jpg` tells us that we can decode the bytes being returned from a service by using a JPEG image reader. The content type `application/json` or `application/hal+json` tell us that the bytes being returned from a resource can be read in using a JSON reader and that it may contain HAL-encoded links. These content types do *not* tell us whether a JSON document matches a certain structure. For this, you need a concept of a schema.

XML offers [XML schema](#) to constrain the structure of an XML resource. For JSON, things are less cut and dry, but there are many de-facto alternatives like [JSON Schema](#). Schema provides a way to specify what the structure of a payload is.

API Versioning

The only thing we know for certain is that things will change; it's inevitable. Indeed, one of the biggest benefit of the cloud-native architecture is that it encourages small batch changes, microservices, that can be evolved quickly and independently of others. Change is a *very* good problem to have. We want to embrace change, but we don't want change to our APIs to emperil other parts of the system or third party clients. We must find ways to evolve our services without unduly breaking consumers.

We could simply try to not break things. If you're only ever changing implementation details, then the visible surface area of an API shouldn't break. Indeed, both clients and services should provide for some flexibility to reduce sensitivity to changes. *Consumer Driven Contracts* help protect APIs from accidental breaks. Continuous integration provides a valuable feedback loop that can be used to continuously protect from accidental API breaks by testing as many possible clients against the APIs under change.

Martin Fowler talks about the idea of a *tolerant reader* - an API client that takes care to avoid over sensitivity to payload changes. A client might expect to find a `order-id` XML element or JSON element in a payload. JSON-Path or XPath queries let clients find elements and patterns in a structure while remaining indifferent to the order or depth of the pattern in the original structure.

Postel's law, also known as the *The Robustness Principle*, says that service implementations should be conservative in what they produce, but liberal in what they accept from others. If a service only needs a subset of a given message payload to do its work it should not protest if there is extra information in the message for which it may not immediately have any use.

It's easy to be more robust so long as changes are evolutionary and additive, not breaking. If an element that was expected in V1 of a service should suddenly disappear in V2, clients would most certainly break. A service should make explicit the assumptions about what clients will be able to work

it. One very popular approach is to use *semantic versioning*. A semantic version is one of the form, MAJOR.MINOR.PATCH. The MAJOR version should change only when an API has breaking changes from its previous version. the MINOR version should change when an API has evolved but existing clients should continue to work with it. A PATCH version change signals bug fixes to existing functionality.

Semantic versioning signals to clients what version of the API is available, but even if the clients are aware of the version they may not be ready to move forward to the new version. If clients are forced to upgrade whenever a service is upgraded, it threatens one of the fundamental benefits of microservices: the ability to evolve independently. So, it will sometimes be necessary to host multiple versions of an API. One approach might be to deploy and maintain two different codebases, side-by-side, but this can quickly become a maintenance nightmare. Instead, consider hosting the versioned APIs side-by-side, in the same codebase. If possible, try to transform and forward requests to older endpoints to the newer endpoints, where possible.

Clients will need to signal to the service which version of the API it intends to talk to. There are some common approaches in REST APIs: encode the version in the URL, as a Header, or as part of the `Accept` header for the request.

Example 5-14. the `VersionedRestController` demonstrates a few approaches to versioning REST APIs

```
package demo;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class VersionedRestController {

    public static final String V1_MEDIA_TYPE_VALUE = "aplica
    public static final String V2_MEDIA_TYPE_VALUE = "aplica

    private enum ApiVersion {
```

```

        v1, v2
    }

public static class Greeting {

    private String how;
    private String version;

    public Greeting(String how, ApiVersion version) {
        this.how = how;
        this.version = version.toString();
    }

    public String getHow() {
        return how;
    }

    public String getVersion() {
        return version;
    }
}

❶ @RequestMapping(value = "/{version}/hi", method = RequestMethod.GET)
Greeting greetWithPathVariable(@PathVariable ApiVersion v)
    return greet(version, "path-variable");
}

❷ @RequestMapping(value = "/hi", method = RequestMethod.GET)
Greeting greetWithHeader(@RequestHeader("X-API-Version"))
    return this.greet(version, "header");
}

❸ @RequestMapping(value = "/hi", method = RequestMethod.GET)
Greeting greetWithContentNegotiationV1() {
    return this.greet(ApiVersion.v1, "content-negotiation");
}

❹ @RequestMapping(value = "/hi", method = RequestMethod.GET)
Greeting greetWithContentNegotiationV2() {
    return this.greet(ApiVersion.v2, "content-negotiation");
}

private Greeting greet(ApiVersion version, String how) {

```

```
        return new Greeting(how, version);
    }
}
```

❶

this approach encodes version in the URL; Spring MVC will automatically map URL parameters into instances of the corresponding `ApiVersion` enum. Test it using `curl http://localhost:8080/api/v2/hi`

❷

this approach automatically converts request headers into instances of the corresponding `ApiVersion` enum. Test it using `curl -H "X-API-Version:v1" http://localhost:8080/api/hi`

❸

this controller handler handles requests for the `application/vnd.bootiful.demo-v1+json` media-type. Test it using `curl -H "Accept:application/vnd.bootiful.demo-v1+json" http://localhost:8080/api/hi`

❹

this controller handler handles requests for the `application/vnd.bootiful.demo-v2+json` media-type. Test it using `curl -H "Accept:application/vnd.bootiful.demo-v2+json" http://localhost:8080/api/hi`

Documenting REST APIs

The agile manifesto suggests working software *over* comprehensive documentation; that is, make it work, *then* document it. It's much better to have working software that explains itself correctly than to have documentation that may lie. Code is a living, breathing thing, and - as we imagine most have at some point experienced, documentation developed in parallel with code tends to drift out of sync with the realities of the code. The documentation serves as a map, but the code is the territory and - if you find yourself lost - it's better to be familiar with the territory than the map!

Documentation is still a very natural, convenient thing, of course! Ever optimistic, we engineers have done a lot of work to collocate the documentation - information about an API - with the API itself. JavaDoc was the first large-scale example of this, though other languages have since introduced alternatives. JavaDoc is maintained by the developer as it lives with the code. The hope is that developers will always see to it that the documentation reflects the realities of the code because it lives with the code. Of course, we know this doesn't always happen! Sophisticated tools, beyond JavaDoc, can infer all manner of things about code.

Hypermedia provides many clues about a well-designed API's use. HTTP `OPTION` tells us what HTTP verbs are supported for a given resource. Hypermedia links tell us what state transitions are supported for a given resource. Schema can tell us the payloads accepted for a given resource. There's no great way of documenting expected HTTP request parameters or headers, though, and there's no great way to explain motivation. Documentation, at its best, describes motivation, not implementation. What's required, then, is a way to complete the coverage of an API's use, as automatically as possible and in such a way that the documentation is guaranteed in-sync with the realities of the code it documents, the territory. There are some options, like Swagger, that requires intrusive changes to the code itself. Swagger relies upon Strings embedded in the Java code in annotations on the API itself. It's fairly cumbersome to maintain human prose in, and it sometimes fails to adequately capture the

intent of an API because it tries to automatically map language-specific constructs to the HTTP contract implied by those constructs.

[Spring RESTDocs](#) takes a different approach; it's delivered as decorations for the Spring MVC *test* and freeform prose expressed in AsciiDoctor.

To begin, add Spring RESTDocs as a test scoped dependency on the CLASSPATH: org.springframework.restdocs : spring-restdocs-mockmvc : 1.0.0.RELEASE.

Then write your documentation using the Spring MVC Test API.

Example 5-15. ApiDocumentation.java uses the Spring MVC test API to exercise an API.

```
package demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoC
import org.springframework.boot.test.autoconfigure.web.servlet.Au
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import javax.servlet.Dispatcher;

import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.notNullValue;
import static org.springframework.restdocs.mockmvc.MockMvcRestDoc
import static org.springframework.test.web.servlet.request.MockMvcV
import static org.springframework.test.web.servlet.result.MockMvcV
import static org.springframework.test.web.servlet.result.MockMvcV
import static org.springframework.test.web.servlet.result.MockMvcV

@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class, webEnvironment = Sp
@AutoConfigureMockMvc
@AutoConfigureRestDocs(outputDir = "target/generated-snippets") ❶
public class ApiDocumentation {
```

test rules support cross-cutting concerns from the tests they're configured on. In this case, `RestDocumentation` rule will emit Asciidoctor snippets to the `target/generated-snippets` directory when the tests are run.

the `documentationConfiguration` static method simply registers the

RestDocumentation object with the MockMvc object which the tests use to exercise the API

③

the String passed to the document method, *error-example*, defines the logical identifier for the automatically generated documentation for this test. In this case, the generated documentation will *also* live under a folder named `error-example` in which we'll find `.adoc` files like `curl-request.adoc`, `http-request.adoc`, `http-response.adoc`, etc. Use these various generated snippets as includes in the creation of a larger AsciiDoctor document.

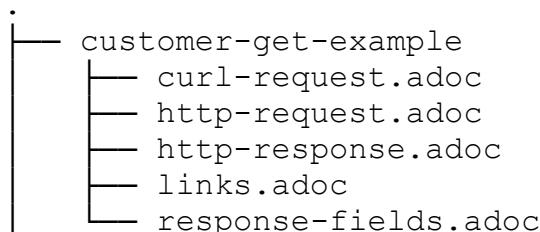
Normally, the build's test plugin finds classes ending in `Test` and runs them. We need to configure the test plugin to include and run `*Documentation` classes, as well.

Example 5-16. including `*Documentation.java` in the Maven Surefire plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>**/*Documentation.java</include>
    </includes>
  </configuration>
</plugin>
```

The bi-product of these tests will be automatically generated AsciiDoctor snippets that are easily included in a larger AsciiDoctor document.

Example 5-17. the automatically generated snippets after running the tests



```
└── customer-update-example
    ├── curl-request.adoc
    ├── http-request.adoc
    ├── http-response.adoc
    └── request-fields.adoc
└── customers-create-example
    ├── curl-request.adoc
    ├── http-request.adoc
    ├── http-response.adoc
    ├── links.adoc
    ├── request-fields.adoc
    └── response-fields.adoc
└── customers-list-example
    ├── curl-request.adoc
    ├── http-request.adoc
    ├── http-response.adoc
    └── response-fields.adoc
└── error-example
    ├── curl-request.adoc
    ├── http-request.adoc
    ├── http-response.adoc
    └── response-fields.adoc
└── index-example
    ├── curl-request.adoc
    ├── http-request.adoc
    └── http-response.adoc
```

6 directories, 26 files

The approach varies from one tool to another, and the Spring RESTDocs project will have more detailed examples for your perusal, but it's easy enough to have Maven process AsciiDoctor documents using the `org.asciidoctor:asciidoctor-maven-plugin:1.5.2` plugin.

Example 5-18. the Asciidoc Maven plugin

```
<plugin>
  <groupId>org.asciidoctor</groupId>
  <artifactId>asciidoctor-maven-plugin</artifactId>
  <version>1.5.2</version>
  <executions>
    <execution>
      <id>generate-docs</id>
      <phase>prepare-package</phase>
      <goals>
```

```
<goal>process-asciidoc</goal>
</goals>
<configuration>
    ①
    <backend>html</backend>
    <doctype>book</doctype>
    <attributes>
        <snippets>${project.build.directory}/generated-snippets</
    </attributes>
</configuration>
</execution>
</executions>
</plugin>
```

①

As configured, the application will look in the `src/main/resources` directory for any valid `.adoc` files and then convert them to `.pdf` and `.html`.

②

To simplify resolving the snippets and avoid repetition in the documentation itself, expose an attribute that resolves to the `generated-snippets` directory.

We won't reprint too much in the way of an AsciiDoctor example here, in the book - you can follow along with the book's code - but you should understand that it's easy to then simply include those snippets whenever you want with the form ````include::{snippets}/error-example/response-fields.adoc[]````.

Once you've written your documentation and used the generated snippets as appropriate, it can be very convenient to serve the documentation as part of the served resources for the application itself. Configure your build tool to copy the generated output to Spring Boot's `src/main/resources/static` directory. Here's how that looks with Maven.

Example 5-19. configuring the Maven resources plugin to include the generated documentation in Spring Boot's `static` directory so that it's available under `http://localhost:8080/docs`.

```
<plugin>
```

```
<artifactId>maven-resources-plugin</artifactId>
<executions>
  <execution>
    <id>copy-resources</id>
    <phase>prepare-package</phase>
    <goals>
      <goal>copy-resources</goal>
    </goals>
    <configuration>
      <outputDirectory>${project.build.outputDirectory}/static/do
    <resources>
      <resource>
        <directory>${project.build.directory}/generated-docs</dir
      </resource>
    </resources>
    </configuration>
  </execution>
</executions>
</plugin>
```

The Client Side

In this section, we'll look at various ways to work with a REST API, both interactively and programmatically.

REST Clients for Ad-Hoc Exploration and Interaction

Spring Boot makes short work of standing up the HAL Browser in your own services to interact with a HAL-based REST APIs. If you're using somebody else's API, or just want a general purpose client, read on!

There are many great, free tools for interactively working with REST APIs. Here are some favorites to have handy during development:

Firefox's Poster plugin - this freely available Firefox plugin is a handy little utility that lives in the bottom right corner of the browser as a little yellow icon. Click it and you'll be presented with a dialog where you can describe and execute HTTP requests. It provides easy features for handling things like file uploads, and content-negotiation.

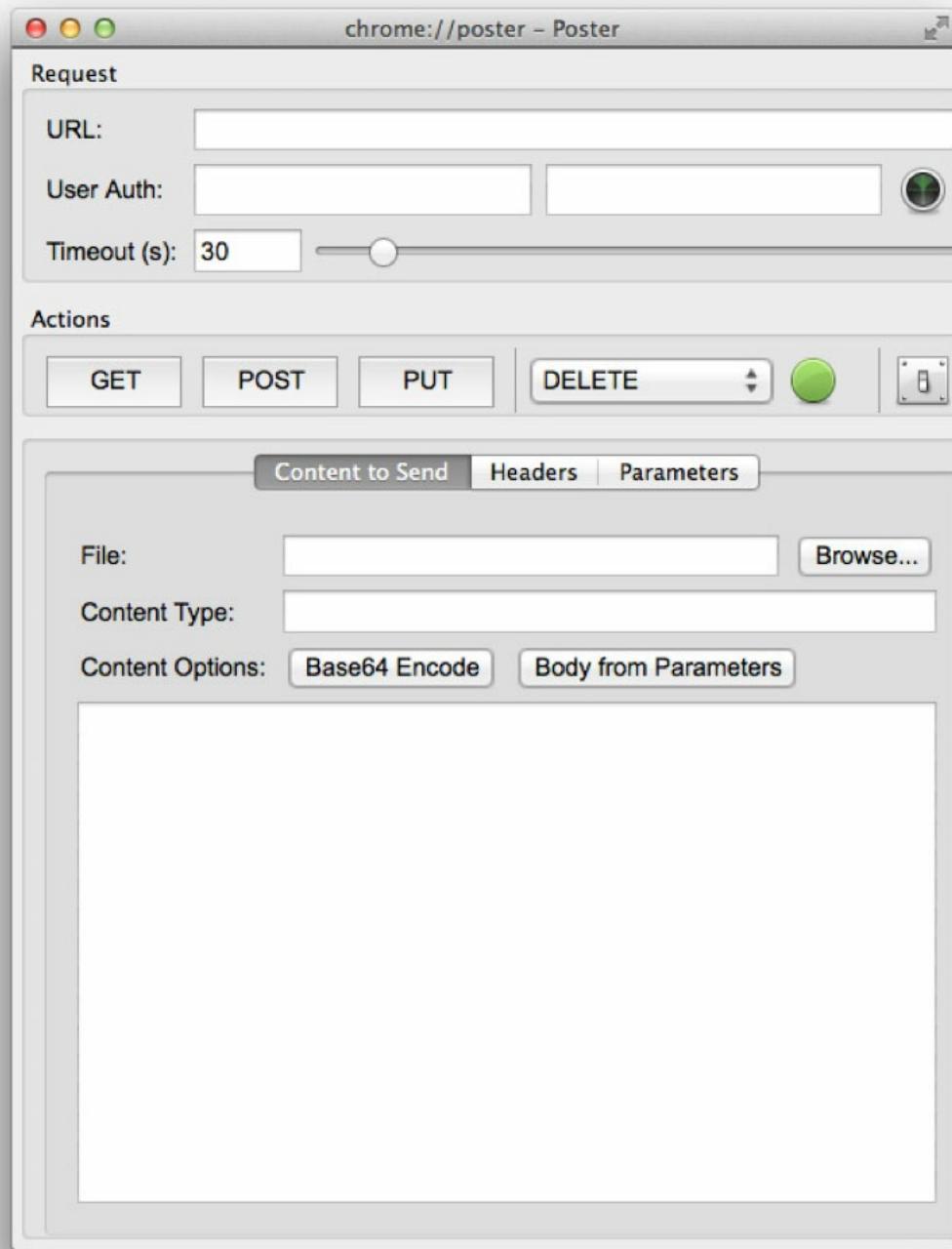
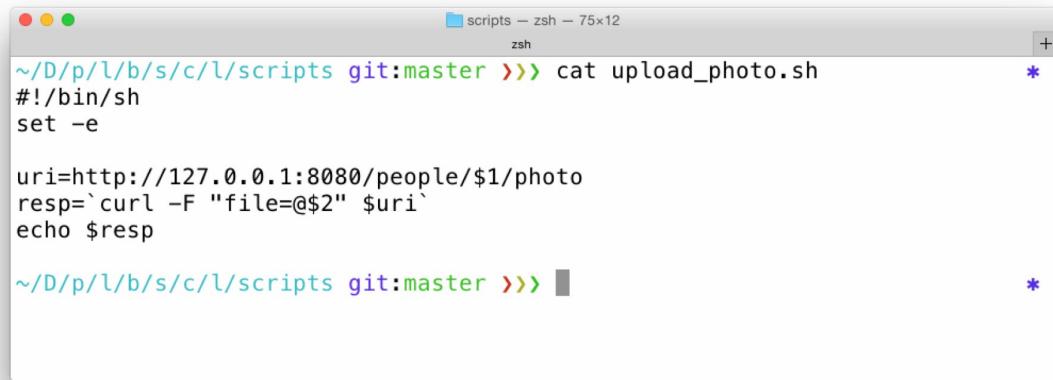


Figure 5-2. the Firefox Poster plugin provides a lot of bang for the buck

curl - the venerable `curl` command is a command line utility that supports any manner of HTTP requests with convenient options for sending HTTP

headers, custom request bodies, and more.



A screenshot of a terminal window titled "scripts - zsh - 75x12". The window contains the following text:

```
~/D/p/l/b/s/c/l/scripts git:master >>> cat upload_photo.sh
#!/bin/sh
set -e

uri=http://127.0.0.1:8080/people/$1/photo
resp=`curl -F "file=@$2" $uri`
echo $resp

~/D/p/l/b/s/c/l/scripts git:master >>> *
```

Figure 5-3. A script that uses curl

The Google Chrome Advanced HTTP Client Extension lets you describe and execute HTTP requests *and* save their configuration for later reuse.

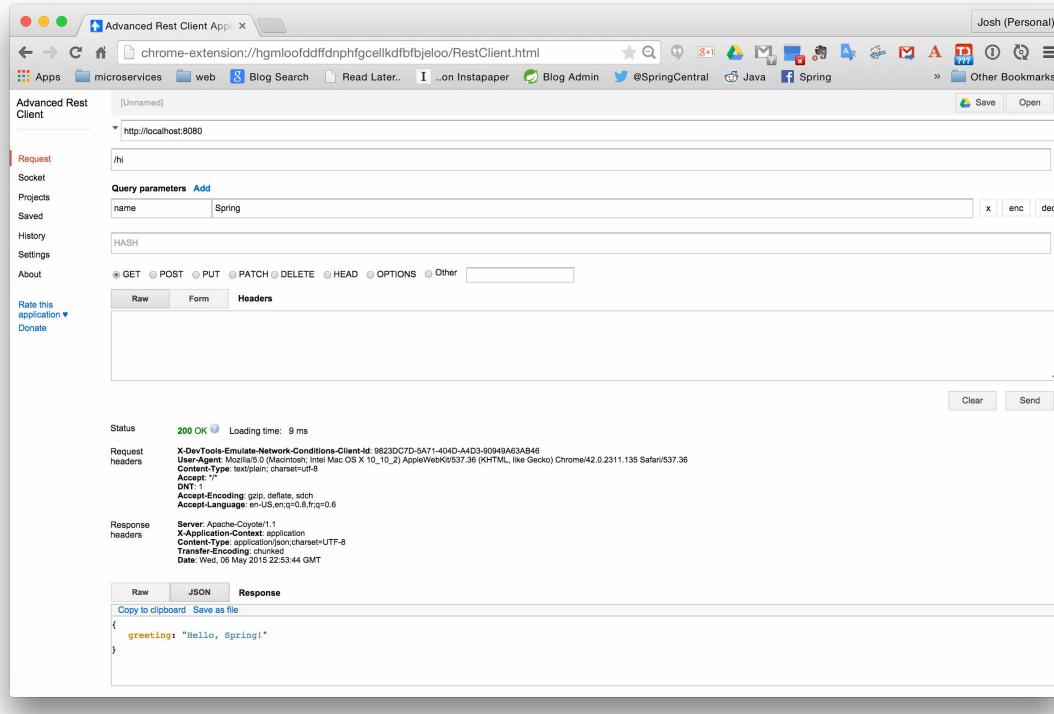


Figure 5–4. The Google Chrome Advanced HTTP Client

The Advanced HTTP Client integrates nicely with the rest of the Google Chrome developer tools. You can open up the Developer Tools (which are in every Google Chrome installation) and inspect the *Network* tab. From there, it's easy to find the request that you crafted and triggered using the Advanced HTTP Client and then have it exported as a `curl` command.

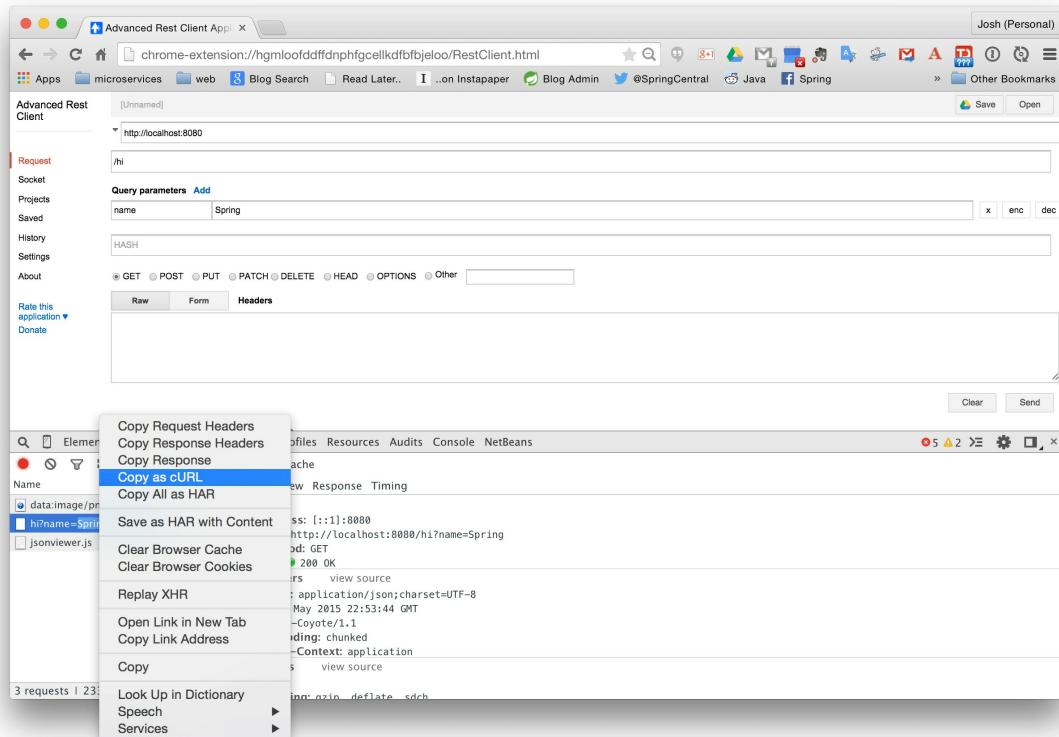


Figure 5-5. Using the Developer Tools in Google Chrome to get a `curl` invocation that replays the HTTP request - very handy!

PostMan this handy Chrome extension provides saved HTTP queries, synchronization across devices, and collections, so that you can manage collections of related HTTP requests. This is becoming my go to HTTP client.

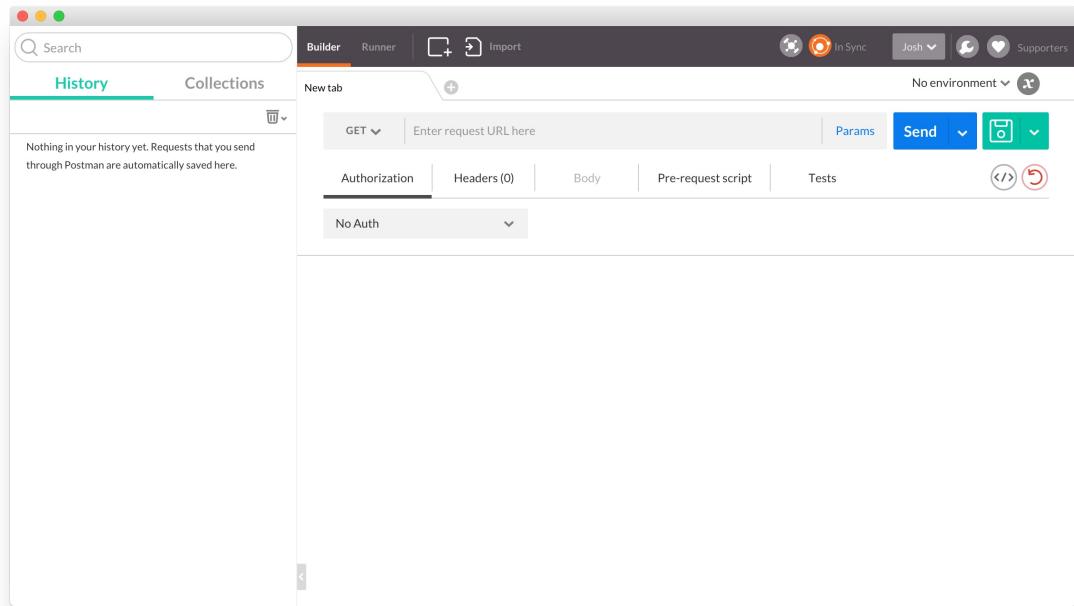


Figure 5-6. The PostMan HTTP Client

The RestTemplate

Spring's `RestTemplate` is a one-stop-shop HTTP client that supports common HTTP interactions with *template* methods; common HTTP interactions - `get`, `put`, `post`, etc. - are one-liners. It supports the same content-negotiation mechanism as Spring MVC does on the server and it supports a pluggable interceptors to centrally handle cross-cutting concerns like authentication with HTTP BASIC and OAuth, GZip-compression, and service resolution.

The `RestTemplate` supports content-negotiation using the same `HttpMessageConverter` strategy implementations as Spring MVC on the server side. It can both convert objects into a valid HTTP request body and convert HTTP responses into objects.

Let's examine an example REST API powered by Spring Data REST. Spring Data REST automatically creates a hypermedia API complete with HAL-style links for relationships based on specified Spring Data repositories. `Movie` entities have one or more `Actor` entities associated with them.

Spring Boot autoconfigures a `RestTemplate` instance for us, though it's easy to override if we want to, for instance, install an interceptor.

Example 5-20. configure a `RestTemplate`

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.web.client.RestTemplate;

@Configuration
public class RestTemplateConfiguration {
```

```

❶
@Bean
RestTemplate restTemplate() {
    Log log = LogFactory.getLog(getClass());
    ClientHttpRequestInterceptor interceptor = (HttpR
        byte[] body, ClientHttpRequestExe
        log.info(String.format("request to URI %s
            request.getURI(), request
            return execution.execute(request, body);
    };

    RestTemplate simpleRestTemplate = new RestTemplat
    simpleRestTemplate.getInterceptors().add(intercep
    return simpleRestTemplate;
}

}

```

The `RestTemplate` is the general purpose HTTP workhorse in the Spring framework.

Example 5-21. use the `RestTemplate`

```

package actors;

import com.fasterxml.jackson.databind.JsonNode;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.embedded.EmbeddedServletC
import org.springframework.context.ConfigurableApplicationConte
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

```

```

import java.util.Collections;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class RestTemplateTest {

    private Log log = LogFactory.getLog(getClass());
    private URI baseUri;
    private ConfigurableApplicationContext server;
    private RestTemplate restTemplate;
    private MovieRepository movieRepository;
    private URI moviesUri;

    @Before
    public void setUp() throws Exception {

        this.server = new SpringApplicationBuilder()
            .properties(Collections.singletonMap("server.port", "8080"))
            .sources(DemoApplication.class).r
            .run();

        int port = this.server.getEnvironment().getProper
            "local.server.port", Integer.class);

        this.restTemplate = this.server.getBean(RestTempl
        this.baseUri = URI.create("http://localhost:" + p
        this.moviesUri = URI.create(this.baseUri.toString
        this.movieRepository = this.server.getBean(MovieR
    }

    @After
    public void tearDown() throws Exception {
        if (null != this.server) {
            this.server.close();
        }
    }

    @Test
    public void testRestTemplate() throws Exception {
        ①
        ResponseEntity<Movie> postMovieResponseEntity = t
            .postForEntity(moviesUri, new Movie("The New Movie", "Action", 100));
        URI uriOfNewMovie = postMovieResponseEntity.getHe
        log.info("the new movie lives at " + uriOfNewMovie);

        ②
        JsonNode mapForMovieRecord = this.restTemplate.ge
    }
}

```

```
        uriOfNewMovie, JsonNode.class);
log.info("\t..read as a Map.class: " + mapForMovieRecord);
assertEquals(mapForMovieRecord.get("title").asText(),
postMovieResponseEntity.getBody())

③
Movie movieReference = this.restTemplate.getForObject(
    Movie.class);
assertEquals(movieReference.title,
    postMovieResponseEntity.getBody())
log.info("\t..read as a Movie.class: " + movieReference)

④
 ResponseEntity<Movie> movieResponseEntity = this.
    .getForEntity(uriOfNewMovie, Movie.class);
assertEquals(movieResponseEntity.getStatusCode(),
assertEquals(movieResponseEntity.getHeaders().get(
    MediaType.parseMediaType("application/json")),
log.info("\t..read as a ResponseEntity<Movie>: "))

⑤
ParameterizedTypeReference<Resources<Movie>> movieResourceType =
    new ParameterizedTypeReference<Resources<Movie>>() {
};
ResponseEntity<Resources<Movie>> moviesResponseEntity =
    exchange(this.moviesUri, HttpMethod.GET);
Resources<Movie> movieResources = moviesResponseEntity.
    getResources();
movieResources.forEach(this.log::info);
assertEquals(movieResources.getContent().size(),
    this.movieRepository.count());
assertTrue(movieResources.getLinks().stream()
    .filter(m -> m.getRel().equals("self"))
    .map(Link::getHref)
    .collect(Collectors.toList())
    .contains(postMovieResponseEntity.getHeaders().get(
        "Location").getHref()));

⑥
}
}
```

the `RestTemplate` provides all the convenience methods you'd expect, supporting common HTTP verbs like `POST`

you can ask the `RestTemplate` to return responses converted to an appropriate representation. Here, we're asking for a Jackson `JsonNode` return value. The `JsonNode` lets you interact with the JSON as you might an XML DOM node.

③

the conversion applies to domain objects, where possible. Here, we've asked for the JSON to be mapped (also using Jackson) to a `Movie` entity

④

the `ResponseEntity<T>` return value is a wrapper for the response payload, converted as described before, and headers and status code information from the HTTP response

⑤

the REST API uses Spring HATEOAS's `Resources<T>` object to serve collections of `Resource<T>` instances which in turn wrap payloads and provide links. In previous examples, we've signaled the return value by providing a `.class` literal, but Java provides no way to capture something like `Resources<Movie>.class` using class literals because the generic information is burned away at compile time, thanks to Java's type erasure. The only way to retain generic parameter information is to bake it into a type hierarchy through subclassing. This pattern, called the *type token* pattern, is supported with Spring's `ParameterizedTypeReference` type. `ParameterizedTypeReference` is an abstract type and so must be subclassed. Here, what seems like an instance variable, `ptr`, is actually an anonymous subclass that can, at runtime, answer the question: "what's your generic parameter?" with "`Resources<Movie>`." The `RestTemplate` can use that to bind the returned JSON values correctly.

Our hypermedia API uses HAL, and HAL emits links to relevant resources. HAL-style links are ideal because one links follows another in a sort of chain. Resolving a specific resource is a matter of following the chain, *hopping* from one link to the next until you're finally at the relevant resource. This is straightforward, but fairly boilerplate code. If we start a resource retrieval at `/`, and need to get to `/actors/search/by-movie?movie=Cars` then it's at *least* three hops:

- one to fetch the root resource `/` and find the `search` link

- another to find the search link for the finder endpoint that queries by movie title, `by-movie`
- another to actually run the search, passing in the `movie` request parameter, `Cars`.

Spring HATEOAS includes the handy `Traverson` client which accepts a link *path* and follows the links for you, yielding the final result.

Tip

the Spring Java client is inspired by [a JavaScript library of the same name](#), so this facility is available to browser clients, as well!

Example 5-22. configure a `Traverson` client using a configured `RestTemplate`

```
package actors;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.embedded.EmbeddedServletC
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.event.EventListener;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.client.Traverson;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

@Configuration
public class TraversonConfiguration {

    private int port;

    private URI baseUri;

    @EventListener
    public void embeddedPortAvailable(EmbeddedServletContain
        this.port = e.getEmbeddedServletContainer().getPo
        this.baseUri = URI.create("http://localhost:" + t
    }
}
```

```
❶
    @Bean
    @Lazy
    Traverson traverson(RestTemplate restTemplate) {
        Traverson traverson = new Traverson(this.baseUri,
        traverson.setRestOperations(restTemplate);
        return traverson;
    }
}
```

❷

configure a `Traverson` client with the base URI, the `MediaType` instances that you want the client to handle, and the `RestTemplate` instance to delegate underlying HTTP calls to.

Use the `Traverson` client by describing a series of links as a path and then following that path.

Example 5-23. use the `Traverson` client with the `RestTemplate`

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.hateoas.client.Traverson;

import java.util.Collections;
import java.util.stream.Collectors;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class TraversonTest {

    private Log log = LogFactory.getLog(getClass());
}
```

```

private ConfigurableApplicationContext server;
private Traverson traverson;

@Before
public void setUp() throws Exception {

    this.server = new SpringApplicationBuilder()
        .properties(Collections.singletonMap("server.port", "0"))
        .sources(DemoApplication.class).run();
    // this.server =
    // SpringApplication.run(DemoApplication.class);
    this.traverser = this.server.getBean(Traverser.class);
}

@After
public void tearDown() throws Exception {
    if (null != this.server) {
        this.server.close();
    }
}

@Test
public void testTraverser() throws Exception {

    String nameOfMovie = "Cars";

    ① Resources<Actor> actorResources = this.traverser
        .follow("actors", "search", "by-name")
        .withTemplateParameters(
            Collections.singletonList("name", "Cars"));
        .toObject(new ParameterizedTypeReference<List<Actor>>() {
    });

    actorResources.forEach(this.log::info);
    assertTrue(actorResources.getContent().size() > 0);
    assertEquals(
        actorResources.getContent().stream()
            .filter(actor ->
                actor.getName().equals("Cars"))
            .collect(Collectors.toList())
    );
}
}

```

① the `Traverser#follow` method makes short work of traversing a *chain* of links, substituting appropriate parts of the URI parameters template as

necessary. The `Traverson#follow` method also accepts a `ParameterizedTypeReference<T>` parameter, just as does the `RestTemplate`.

The `RestTemplate` plays a critical part of any REST-based microservice system; it is the connective tissue between services. Later, we'll look at how to use it to make calls from one service to another and handle load-balancing and service resolution through auto-configured interceptors.

Chapter 6. Managing Data

This chapter will address some of the concerns of managing data when building a scalable cloud-native application. We will review some familiar methods for modeling the data of a domain. We'll take a look at how Spring Data projects expose repositories for managing data. We will also look at a few examples of microservices that manage exclusive data access to a data source using popular Spring Data projects.

Modeling Data

Well constructed data models help us to effectively communicate the desires of a business's domain in our software applications. Domain models can be constructed in a manner that are best suited to express the salient traits of a business's domain. By-and-far one of the most successful techniques for domain modeling was first presented by Eric Evans in his seminal book *Domain-Driven Design* (Addison-Wesley).

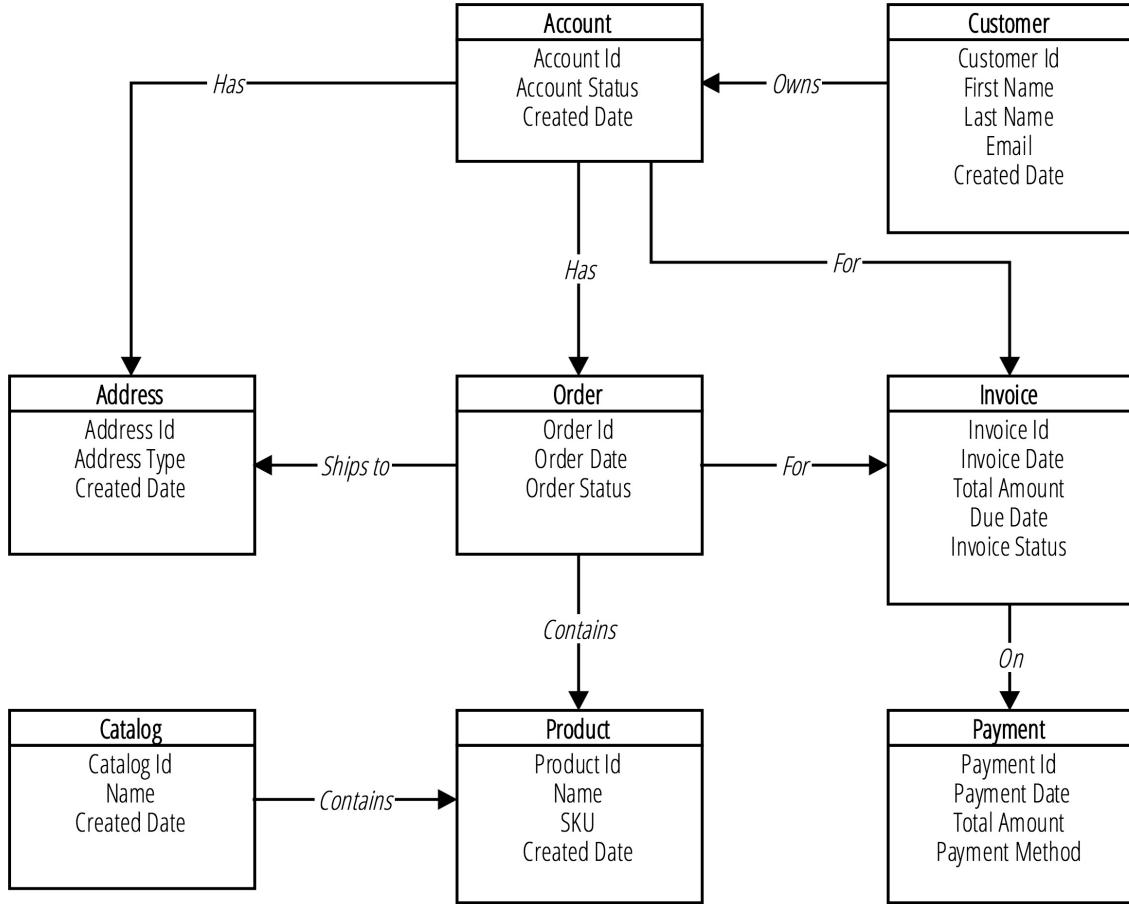


Figure 6-1. A domain model expressing the relationships between domain classes

Evans popularized the concepts of domain-driven design by bringing forward an understanding that both the domain experts in the business and the software engineers in the IT organization should be able to effectively communicate using unambiguous terms that describe objects and modules in a software application.

The problem that domain-driven design purported to solve was **complexity**. Evans sub-titled his book “*Tackling Complexity in the Heart of Software*”. What Evans meant by this was that technologists should focus on making software easier to create and maintain through focusing on untangling the complexity in the underlying model of the business. This complexity exists in the tangle of business processes and functions that companies create to serve customers. Domain models map out this complexity and provide technologists a language to write better software for the business.

Note

If an engineer names a **Customer** as a **User** inside a software module, the engineers are then forced to use ambiguous terms that may mean something entirely different to the domain experts of the business.

Consider the following example.

Example 6-1. A user story for customer account management

As a **customer**, I want to be able to create **users** that are able to manage a set of **accounts**.

This user story was written by a domain expert of the business. The story describes an account management use case and clearly distinguishes between a *customer* and a *user* but does not disambiguate whether or not a *user* can also be a *customer*.

Without a domain model that distinguishes this difference, an engineer may decide that the simplest solution in code would be to have a single domain class named `User`. Later on in a separate release the business may then clarify that a `Customer` is not the same thing as a `User`. This clarification would later turn out to be a costly design decision with stark consequences for new feature development, causing the software engineers to miss their original release date.

RDBMS

The relational database model has been the staple of transactional data storage solutions for many decades. As technology continues to evolve to cloud-native architectures, the market is starting to demand other types of data models that have the same transactional guarantees as RDBMS. Certain use cases with a low level of data complexity are a great fit for relational databases. Spring Data JPA provides extensive support for SQL-based RDBMS solutions, including embedded in-memory databases such as H2 and HSQLDB, which are a popular option for integration testing without needing to connect to a database server.

NoSQL

NoSQL databases provide a variety of data models that have specialized traits that are optimized to address the needs of an application's use cases. With microservices this becomes advantageous since we can decompose bounded contexts of a domain model to use a specialized trait of a NoSQL database. Using multiple databases in an architecture, with a microservice managing exclusive data access to a single database, we have the opportunity to build cloud-native applications that are truly polyglot.

Polyglot Persistence is a term popularized by Martin Fowler and describes an architecture that uses multiple types of database models. Applications that are centralized on a single relational database can slowly be decomposed to use microservices that manage their own NoSQL database, depending on the use case of that facet of the application. This is especially useful for greenfield projects where the business is demanding higher complexity solutions at a rapid speed of delivery. Since NoSQL databases provide trait-specific data models that are optimized for a use case, there is less of a need to bend a relational database model to solve a certain problem.

Spring Data

Spring Data is an open source project in the [Spring Framework](#) ecosystem of tools that provides a familiar abstraction for interacting with a data store while preserving the special traits of its database model. In total, and at the current time of writing this book, there are 10 Spring Data projects that cover a range of popular databases, including RDBMS and NoSQL data models.

Table 6-1. The Spring Data Projects

| JDBC | JPA |
|-----------|---------------|
| MongoDB | Neo4j |
| Redis | Elasticsearch |
| Solr | Gemfire |
| Cassandra | Couchbase |

Note

If you’re wondering where all the relational databases are in this list, that’s because Spring Data JPA covers all of the popular RDBMS vendors, including: Oracle, MySQL, PostgreSQL, H2, HSQL, Derby and more.

Structure of a Spring Data Application

When getting started with Spring Data it's helpful to understand the patterns that are used for designing a data access layer. Let's start by understanding what the basic ingredients are that are used to interact with a data store.

Domain Class

The first ingredient is an entity class that represents an object in your domain model. An entity class is a basic class that functions as a model for your domain data. These domain classes consist of a set of private fields and expose their contents using public getters and setters, depending on the design of your domain model.

Example 6-2. A basic domain class representing a User model

```
public class User {  
  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;
```

Note

Domain classes represent entities that map to data objects (*such as tables*) in an application's data store.

Repositories

The primary method for accessing data in a Spring Data application is a Repository. Repositories in Spring Data take the form of multiple types of interfaces that describe data management functions. The most basic of these interfaces is `Repository`, and is responsible for capturing two pieces of information: the domain class's type, and the ID type of the domain class.

Example 6-3. The base repository abstraction in Spring Data

```
public interface Repository<T, ID extends Serializable> {  
}
```

If we wanted to create a repository for a `User` domain class we could use the `CrudRepository` interface, which extends the `Repository` interface above, to provide basic data management operations on our `User` domain class.

Example 6-4. A Spring Data repository for a User domain class

```
public interface UserRepository extends CrudRepository<User, Long>
```

This repository definition will be instantiated at runtime as a managed `Bean` that provides store-specific data management capabilities.

Note

Repositories provide data management APIs for an application's domain classes that are mapped to native data store objects.

Organizing Java Packages for Domain Data

The style in which we organize our packages becomes a much more important consideration when building microservices. The need for more opinionated styles of package structure is important because microservices may later be decomposed into smaller services that cover a smaller area of a bounded context. We recommend that you organize your classes and packages in the way you feel most comfortable with, but we will propose here a pattern we've found suitable for building microservices using Spring Data and Spring Boot.

First we need to determine what the bounded contexts are in our domain model. Let's consider the following domain model for this exercise.

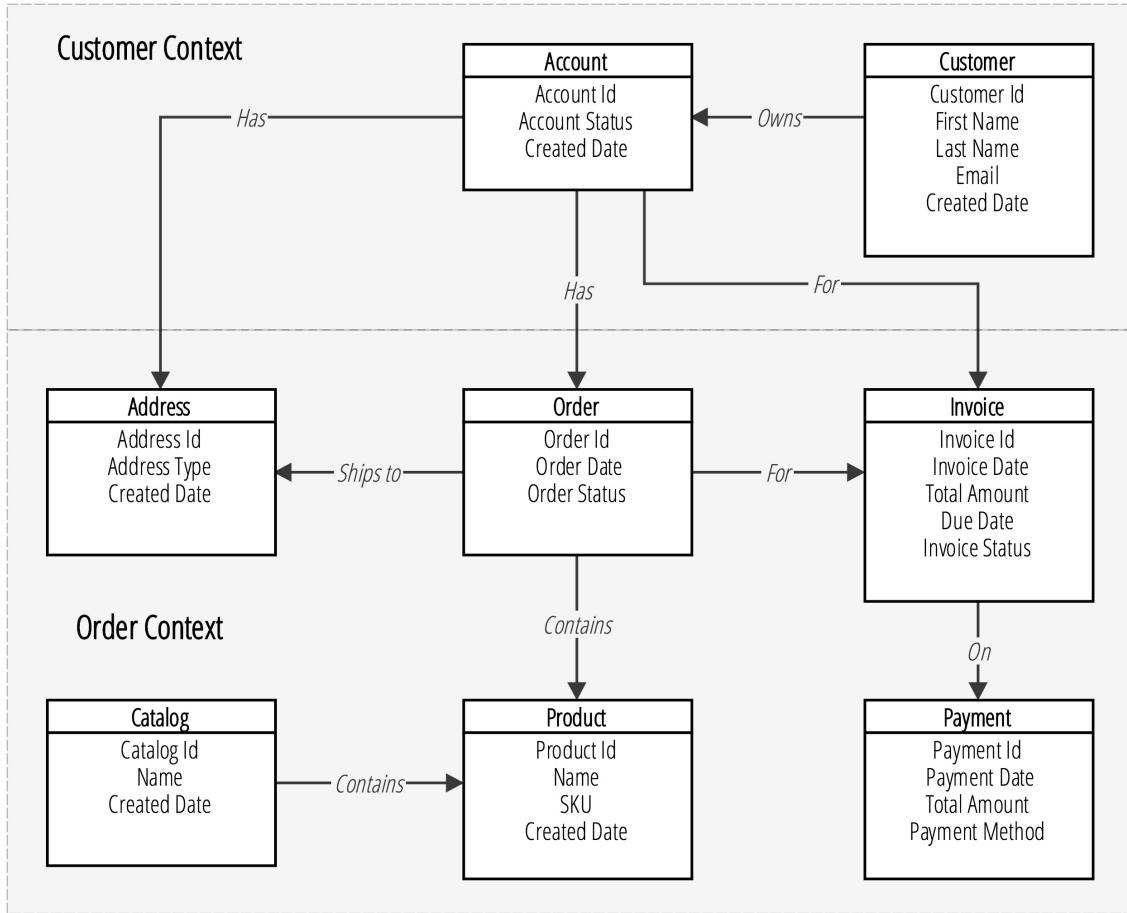


Figure 6-2. A domain model with two bounded contexts

The diagram represents a domain model that has a clear boundary between the Customer Context and the Order Context. Let's assume we would like to create two microservices to cover each of these bounded contexts. We will need to organize our packages so that later we can easily migrate domain classes and repositories for a domain concept. To best prepare ourselves for this refactoring exercise, we can group domain classes and repositories that manage a single domain class into a single package.

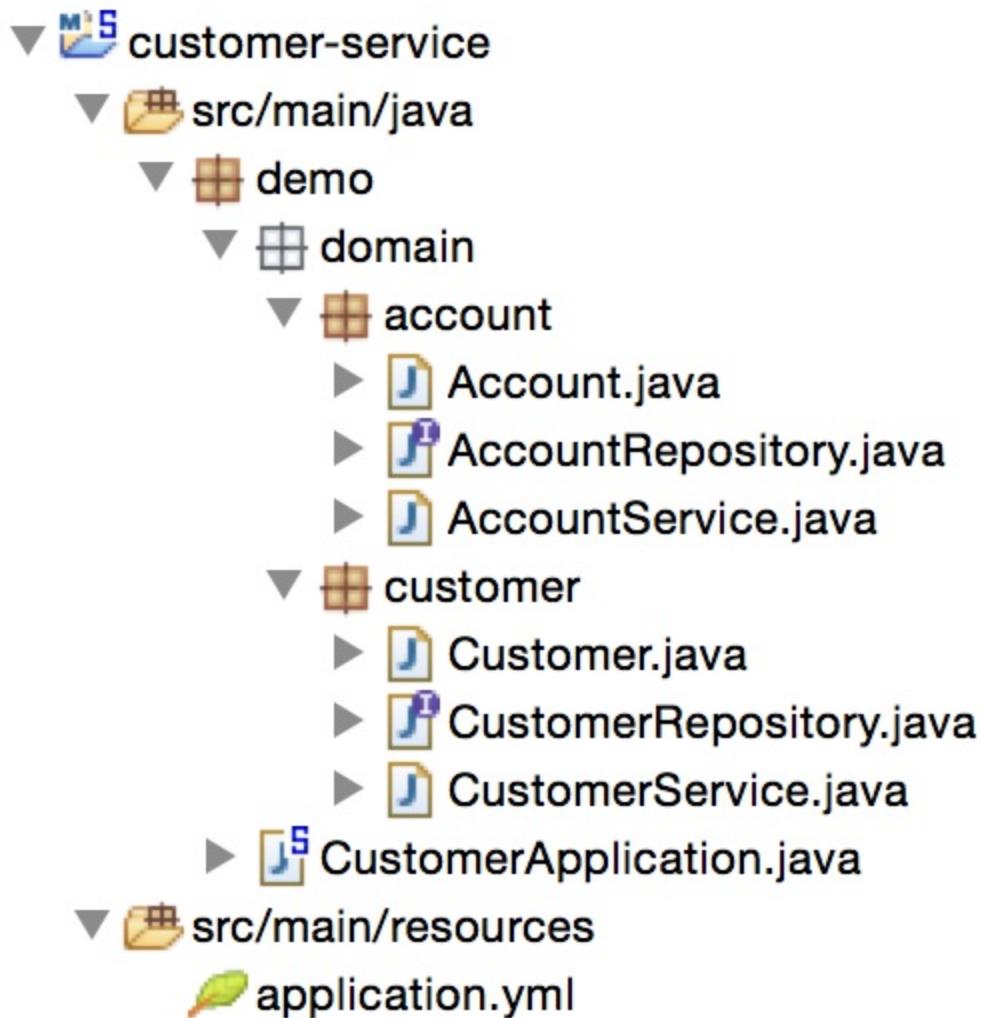


Figure 6-3. Package structure for the Customer context

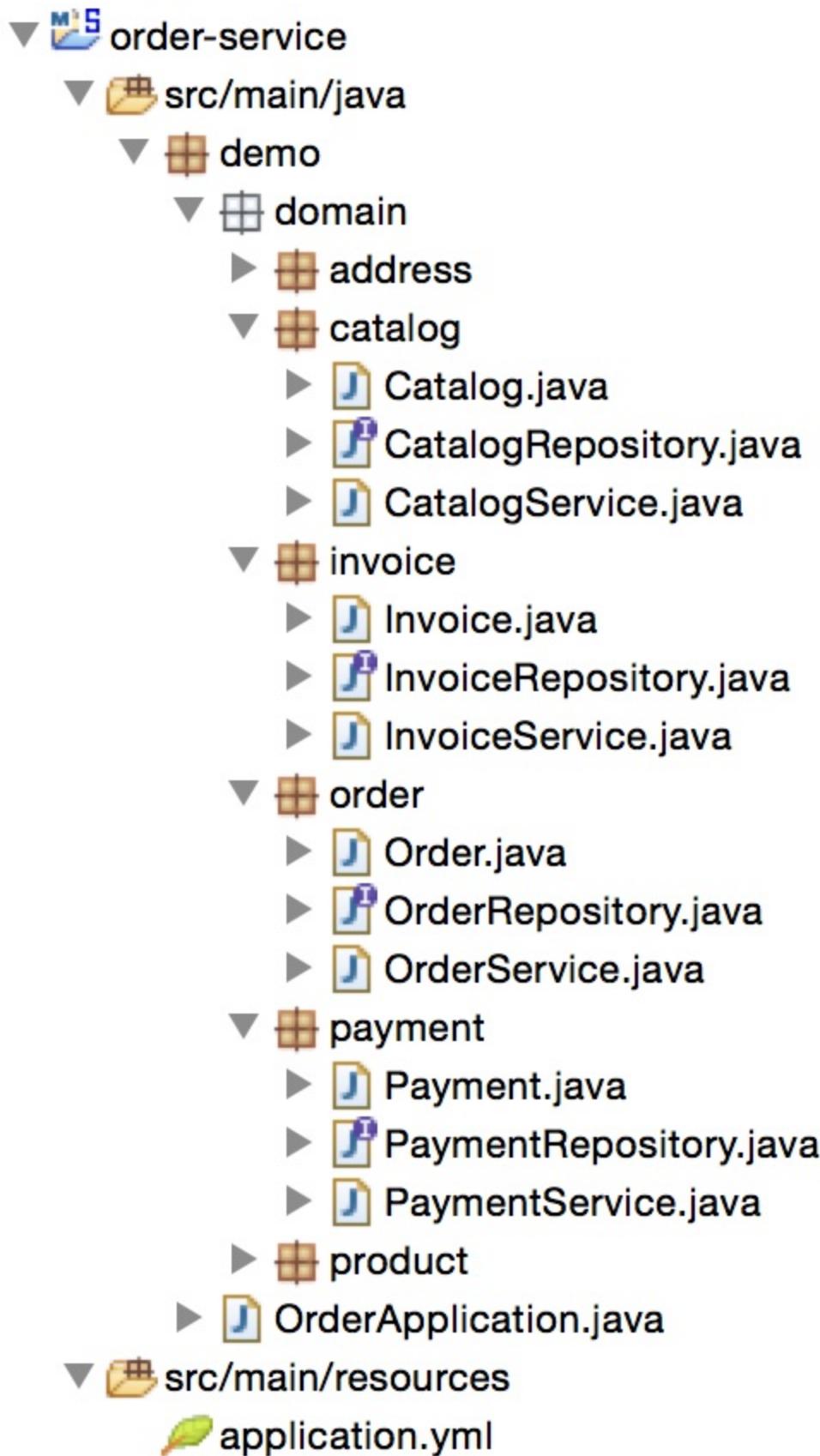


Figure 6-4. Package structure for the Order context

Supported Repositories

Spring Data projects will support multiple kinds of repositories that extend the `Repository` interface.

Table 6-2. Common Spring Data Repositories

| Repository Type | Spring Data Project | Purpose |
|-----------------------------------------|----------------------------------|----------------------------------------------------------------------------------------------------|
| <code>Repository</code> | <code>Spring Data Commons</code> | Provides the core abstraction for Spring Data repositories |
| <code>CrudRepository</code> | <code>Spring Data Commons</code> | Extends <code>Repository</code> and adds utility for basic CRUD operations |
| <code>PagingAndSortingRepository</code> | <code>Spring Data Commons</code> | Extends <code>CrudRepository</code> and adds utility for paging and sorting records |
| <code>JpaRepository</code> | <code>Spring Data JPA</code> | Extends <code>PagingAndSortingRepository</code> and adds utility for JPA and RDBMS database models |
| <code>MongoRepository</code> | <code>Spring Data MongoDB</code> | Extends <code>PagingAndSortingRepository</code> and adds utility for managing MongoDB documents |

`CouchbaseRepository`

Spring Data Couchbase Extends `CrudRepository` and adds utility for managing Couchbase documents

Each repository interface that ships with Spring Data provides an abstraction for a specific utility that may be needed for your Spring Data application. The table above describes some of the many repository interfaces in the Spring Data ecosystem of projects. Both the `CrudRepository` and `PagingAndSortingRepository` interfaces are members of the Spring Data Commons library, providing abstractions that are used by the vendor-specific Spring Data projects.

One such example is the `MongoRepository` interface, which is the primary repository abstraction used in the Spring Data MongoDB project. This repository extends the `PagingAndSortingRepository` interface, providing basic data management capabilities to interact with paged database records.

Tip

CRUD is an acronym for **CREATE**, **READ**, **UPDATE**, and **DELETE**. These 4 verbs describe the basic management capabilities for data in any kind of persistent storage technology. Not coincidentally, the HTTP protocol uses slightly different verbs to manage resources as network transactions, which are **POST**, **GET**, **PATCH**, and **DELETE**.

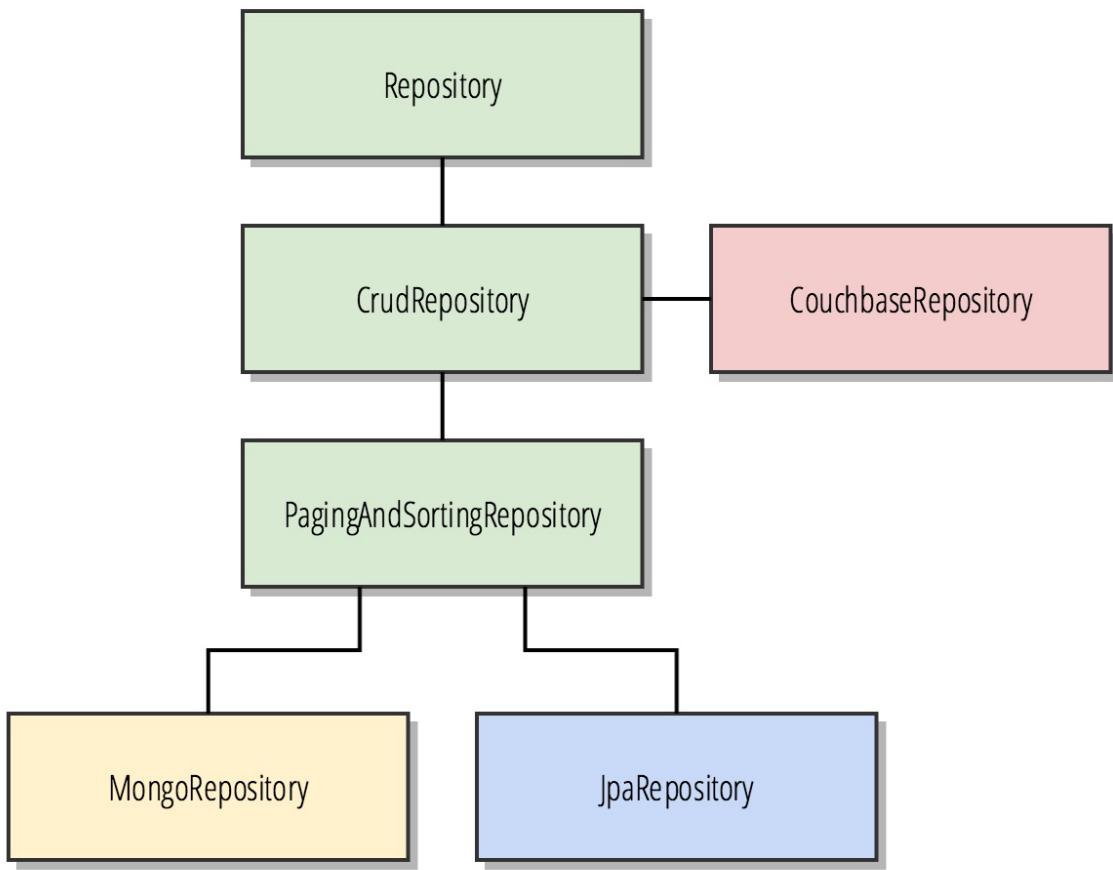


Figure 6-5. The 3 base repositories in Spring Data Commons provide basic data access to vendor-specific data stores

Auto-configuration

As we already discussed earlier on, Spring Boot uses something called *auto-configuration* to bootstrap dependencies of your application using a set of default settings.

These dependencies are declared in a Spring Boot application's `pom.xml`. The common dependencies we will need are listed below.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Bootstrapping Datasource Dependencies

One of the core principles of Spring Boot is minimal configuration. Spring Boot will automatically scan the classpath of an application at startup and bootstrap dependencies. For example, if I added `spring-boot-starter-jpa` as one of my dependencies, the data source is automatically configured by looking for a compatible database driver elsewhere in the dependencies.

In the example code snippet from the `pom.xml`, we've included a `mysql-connector-java` dependency. Now when I start the Spring Boot application, this MySQL driver will be automatically configured as our default [data source for JPA](#). We'll dive deeper into how this works in the Spring Data example projects later in this chapter.

Datasource Connections

The data source connection details are retrieved from application properties for the active profile. Those configurations are contained in `application.yml`. Below is an example application properties for a *Spring Data JPA* application that has a connection to a *MySQL* database.

```
spring:
  profiles:
    active: development
---
spring:
  profiles: development
  jpa:
    database: MYSQL
  datasource: ❶
    url: jdbc:mysql://localhost/test
    username: dbuser
    password: dbpass
```

❶

The `spring.datasource` property block is where you configure store-specific connection details.

JDBC Template

One of the classic data access mechanisms in the Spring Framework is the `JdbcTemplate`. The `JdbcTemplate` provides management functions over SQL-based relational databases that support the JDBC specification. This section is going to walkthrough how to use `JdbcTemplate` to execute SQL queries and map result sets using the the *Java Streams API* that was added in Java SE 8.

Tip

Templates are an abstraction used in Spring Data that provide management functions that are specific to the traits of a type of data store.

A `JDBCTemplate` can be Autowired as an annotation or through setter injection inside a managed Bean.

```
@Bean  
CommandLineRunner commandLineRunner(JdbcTemplate jdbcTemplate) {
```

Now we can execute SQL queries to the MySQL database that we've configured as a datasource using `application.yml` for the development profile. Let's create a `User` table and insert some records through the `CommandLineRunner` method illustrated above.

First, we create the `User` table:

Example 6-5. Create a user table with 4 columns: `id`, `first_name`, `last_name`, and `email`

```
jdbcTemplate.execute("CREATE TABLE user(id BIGINT PRIMARY KEY NOT  
                      first_name VARCHAR(255), last_name VARCHAR(
```

Next, we can use the *Java Streams API* to parse an array of `String` objects using a space symbol as a delimiter. We'll do this so that we can batch insert users into our newly created `User` table.

Example 6-6. Split each supplied string into columns using the space symbol as a delimiter

```
List<Object[]> splitUserRecords = Arrays.asList("Michael Hunger m  
"Bridget Kromhout bridget@outlook.com",  
"Kenny Bastani kbastani@gmail.com",  
"Josh Long josh@joshlong.com")  
.stream()  
.map(name -> name.split(" "))  
.collect(Collectors.toList());
```

Here we have 4 users we want to batch insert into the `User` table. Each user represents a string of text which we can use to generate an array that maps to our columns we created in the `User` table. We can use the `map` function of the *Java Streams API* to split each row into 3 columns of our `User` table:

- *first_name*
- *last_name*
- *email*

Now that we have a `List<Object[]>` that represents the 4 users we would like to insert using the `JdbcTemplate`, we can simply use `(?, ?, ?)` as a placeholder parameter in the `INSERT` statement. The placeholder will be sourced from our `splitUserRecords` object we created earlier.

Example 6-7. Use the `JdbcTemplate`'s `batchUpdate` method to insert the new user records

```
jdbcTemplate.batchUpdate("INSERT INTO user(first_name, last_name,  
splitUserRecords);
```

Now that we have inserted a few records into our `User` table, let's use `JdbcTemplate`'s `query` method to find a user by their first name. The user we will query for will be *Josh Long*. We will use the following statement to execute a SQL query lookup using the first name *Josh* as the parameter.

Example 6-8. Use the `JdbcTemplate` `query` method to search for records with the first name *Josh*

```
jdbcTemplate.query("SELECT id, first_name, last_name, email FROM
    new Object[] {"Josh"}, 
    (rs, rowNum) ->
        new User(rs.getLong("id"),
            rs.getString("first_name"),
            rs.getString("last_name"),
            rs.getString("email")))
    .forEach(user -> log.info(user.toString()));
```

Here we declare our SQL query and provide the parameter *Josh*. We then use a lambda function to initialize an instance of `User`. Finally, we use `forEach` to iterate through the results and log to the console. The console result is shown below.

```
User{id='3', firstName='Josh', lastName='Long', email='jlong@hotm
```

Spring Data Examples

Earlier we explored some of the basic concepts of Spring Data: domain classes and repositories. Now we'll dive deeper into example projects that use vendor-specific database technologies. We will cover example data services for the following Spring Data projects.

- Spring Data JPA (MySQL)
- Spring Data MongoDB
- Spring Data Neo4j
- Spring Data Redis

The example domain we will use for these set of data examples will be an online storefront. We will use a domain model that describes an enterprise resource planning (ERP) application. Each one of our services will extend itself to a particular bounded context of the domain.

The online storefront application we will use for our example projects will be for the fictitious company *Monolithic Ltd.*, which is a clothing brand that is looking to move their application development for their online store to the cloud.

Monolithic Ltd. has been having trouble enabling development teams to deploy new features in their existing online store. Inspired by their move to more modern methods of software development, the startup company has decided to rebrand their offering to *Cloud-native Clothing*.

Let's take a look at the domain model that *Cloud-native Clothing* will be using to construct their new online store.

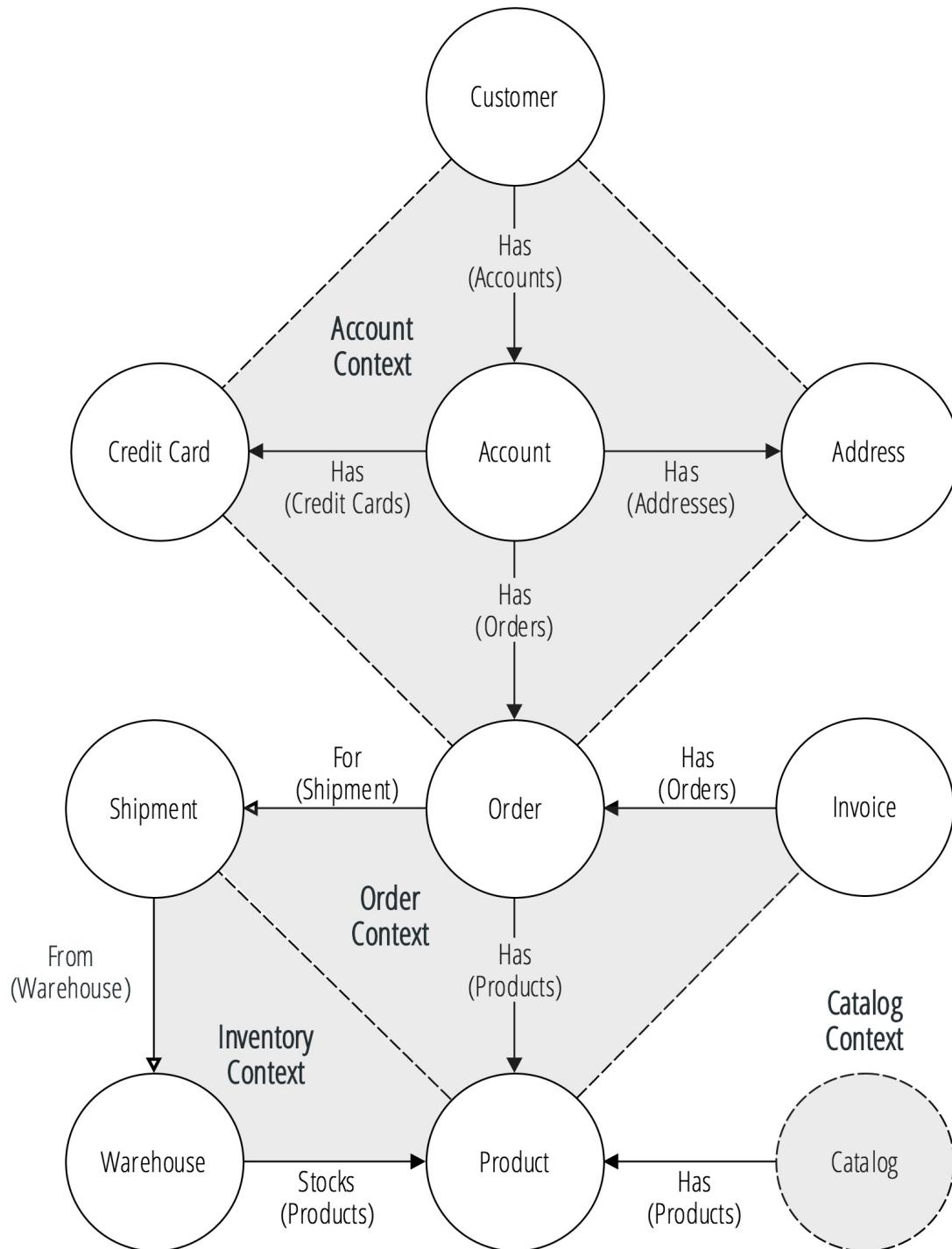


Figure 6-6. Cloud-native Clothing’s Domain Model

We can see from the *Cloud-native Clothing* domain model that we have several bounded contexts. Each bounded context in the diagram is shaded in

gray and labeled as a context. According to the principles set forth in DDD, bounded contexts are a way modularize a part of a business's domain into separate unified models. What this means is that each bounded context can have a set of unrelated concepts while making explicit the connection between shared concepts. I've decided to create a model that can both articulate the concepts of a business domain while also articulating how the application is constructed using Spring Data repositories.

Each domain class in the diagram is represented as a circle. Each circle also represents a Spring Data repository. Repositories, in the context of the diagram, are required for any domain class that will need to expose query capabilities within a bounded context, hence using this type of model we'll always consider a circle to be both a *domain class* and a *repository*. The interrelationships between repositories indicate some level of connectedness in each domain class.

Concepts that are connected by a *directed relationship* indicate an *explicit relationship* between domain classes. Concepts that are connected by dotted lines indicate inferred connectedness through a *repository join*. A dotted line would indicate that there should be a repository query that acts as a joined bridge between domain concepts. The bridge should only be valid within the context of a shared concept of a domain class that exists between two repositories (*for example, Account bridges the connection between Order and Address*).

In SQL this concept is simply referred to as a `JOIN` query between tables.

```
SELECT o.* FROM orders o
INNER JOIN account ac ON o.accountId = ac.Id
INNER JOIN address ad ON ad.accountId = ac.Id
WHERE ad.id = 0;
```

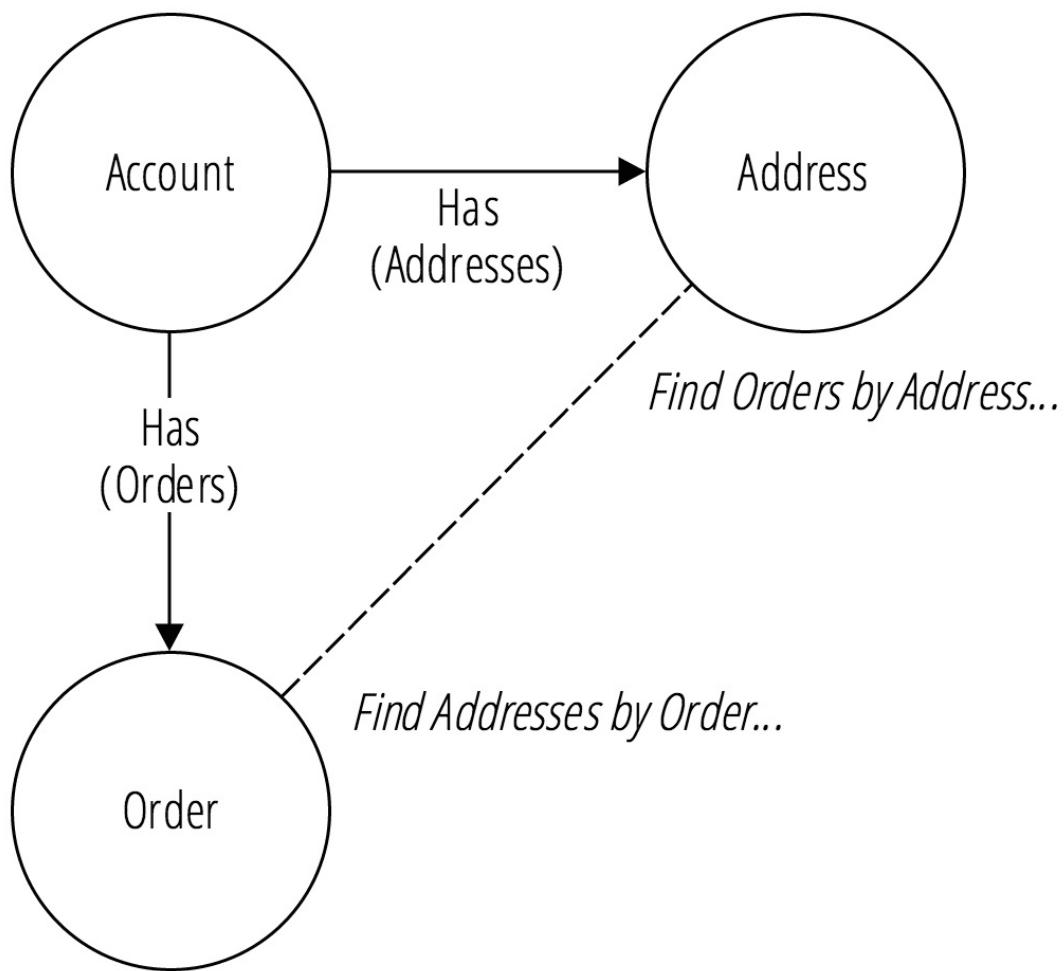


Figure 6-7. Inferred relationships can be looked up through a shared domain class

We will use these bounded contexts to create 4 separate data services that use a specific Spring Data project. Since we're building data services using Spring Boot, later in the book we will be able to easily add in Spring Data REST and Spring Cloud to create a cloud-native application with service discovery and configuration management.

Table 6-3. Microservices for our Cloud-native Clothing Application

| Service | Datasource | Spring Data Project |
|----------------|-------------------|----------------------------|
|----------------|-------------------|----------------------------|

| | | |
|-----------------|-------|-----------------|
| Account Service | MySQL | Spring Data JPA |
|-----------------|-------|-----------------|

Order Service MongoDB Spring Data MongoDB

Inventory Service Neo4j Spring Data Neo4j

Spring Data JPA

The Spring Data JPA project covers repository-based data management solutions for databases that support the SQL query language. Spring Data JPA provides vendor-specific object-relationship mapping (ORM) mostly for relational database models (RDBMS) that use SQL. It's worth noting that the Spring Data JPA project is the only Spring Data project that provides repository abstractions for RDBMS.

Note

JPA stands for `Java Persistence API`, which was first described and introduced as a part of JSR 220 of the JCP (Java Community Process). JPA provides the abstraction and implementations for vendor-specific ORM technologies, such as Hibernate and DataNucleus. This is a powerful facet of the Spring Data project that allows you to use virtually any RDBMS technology that provides a driver that supports the JPA specification.

Account Service

We will use Spring Data JPA for our Account Service, which covers the account context of *Cloud-native Clothing*'s domain model. Spring Data JPA is somewhat of a special project in the Spring Data ecosystem. Other than the *Spring Data JDBC* project, *Spring Data JPA* provides a single library that plays nicely with multiple data sources. This is advantageous for integration testing since we can use an embedded SQL database to perform integration testing.

In this section we're going to walk through the main concerns of setting up a Spring Data JPA application for the Account Service of *Cloud-native Clothing*. We'll cover:

- Spring Boot configurations

- Profiles
- Domain classes
- Repositories
- JPA auditing
- Integration testing

We're going to use the *Spring Initializr* project at <http://start.spring.io> to preconfigure our project dependencies for our `Account Service`. The project dependencies we will need for *Cloud-native Clothing's* `Account Service` are in the table below.

Table 6-4. Dependencies for Cloud-native Clothing's Account Service

| Dependency | Purpose |
|-----------------|---------------------------------------------------|
| Spring Data JPA | Spring Boot starter project for Spring Data JPA |
| MySQL Database | Database driver for MySQL |
| H2 Database | Embedded in-memory database for integration tests |

Note

Spring provides a *Spring Initializr* service at <http://start.spring.io> to help you bootstrap your Spring Boot applications. You can also follow along with the provided example projects for this chapter at <http://github.com/cloud-native-java/data>.

Let's use the following `CURL` command from the terminal to setup our boiler plate for the `Account Service`. Note that we are providing parameters for the project dependencies we listed earlier for this JPA project.

```
curl https://start.spring.io/starter.tgz -d dependencies=data-jpa  
-d type=maven-project -d baseDir=account-servi
```

This command will first make a request to the *Spring Initializr* service, then download and extract a compressed file that includes a preconfigured Spring Boot application. The project will be configured with the dependencies we need in the Maven `pom.xml`. After you run the command you can import the project to your IDE of choice.

Configuring Spring Data JPA

Navigate to the `pom.xml` file to see that the following dependencies are configured for our application.

```
<dependencies>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-data-jpa</artifactId>  
    </dependency>  
  
    <dependency>  
        <groupId>com.h2database</groupId>  
        <artifactId>h2</artifactId>  
        <scope>runtime</scope>  
    </dependency>  
    <dependency>  
        <groupId>mysql</groupId>  
        <artifactId>mysql-connector-java</artifactId>  
        <scope>runtime</scope>  
    </dependency>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-test</artifactId>  
        <scope>test</scope>  
    </dependency>  
</dependencies>
```

Configuration Profiles for Development and Test

Now that we have the skeleton for our Spring Data JPA project with MySQL and H2 database drivers on the class path, we need to configure our

application to use the two databases in the correct context. The MySQL database only runs external to our application and we will connect to it remotely while running the application. We only want to use the embedded H2 database when running our integration tests. Spring provides a way to configure your application to run with a separate set of configurations depending on what the active profile is at runtime. In the `application.yml` file you can create `development` profile and a `test` profile.

Example 6-9. Configure your application for development and test

```
spring:
  profiles:
    active: development ①
---
spring:
  profiles: development ②
  jpa:
    database: MYSQL
    generate-ddl: true
  datasource:
    url: jdbc:mysql://192.168.99.101:3306/dev
    username: root
    password: dbpass
---
spring:
  profiles: test ③
  jpa:
    database: H2
  datasource:
    url: jdbc:h2:mem:testdb
```

①

This is the active profile by default when running the application

②

This is the configuration for the *development* profile

③

This is the configuration for the *test* profile

Now that we have two separate profiles defined for development and test configurations, we need to configure our integration tests to use the **test** profile. Since we've set the active profile in the application properties to default to **development**, our integration tests will try to use the development profile and we don't want that. To solve this we need to override the active profile when in the context of our integration tests only.

The code block below describes a test class that is enabled to run with `SpringJUnit4ClassRunner`, which indicates that this class should run with the Spring *testing context*. We also are able to indicate the profile that should be active for this *testing context*, overriding the default active profile that we've set through the application properties.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = AccountApplication.class)
@ActiveProfiles(profiles = "test") ❶
public class AccountApplicationIntegrationTests extends TestCase
```

❶

The `@ActiveProfiles` annotation allows us to override the active profile for the testing context

Now when we run our integration tests for the `Account Service`, we will use the H2 embedded in-memory database. By using an embedded database we can run tests in any build environment and not worry about having to connect to an external dependency.

JPA Entity Classes

The domain classes for our `Account Service` will be created as JPA entity classes. A JPA class is an annotated domain class that will be mapped to a table in our relational database (either MySQL or H2 in this case).

In the domain model for our *Cloud-native Clothing* application we can see that there are 4 bounded contexts. In the *Account Context* we can see that there are 5 repositories: **Account**, **Address**, **Order**, **CreditCard**, and **Customer**.

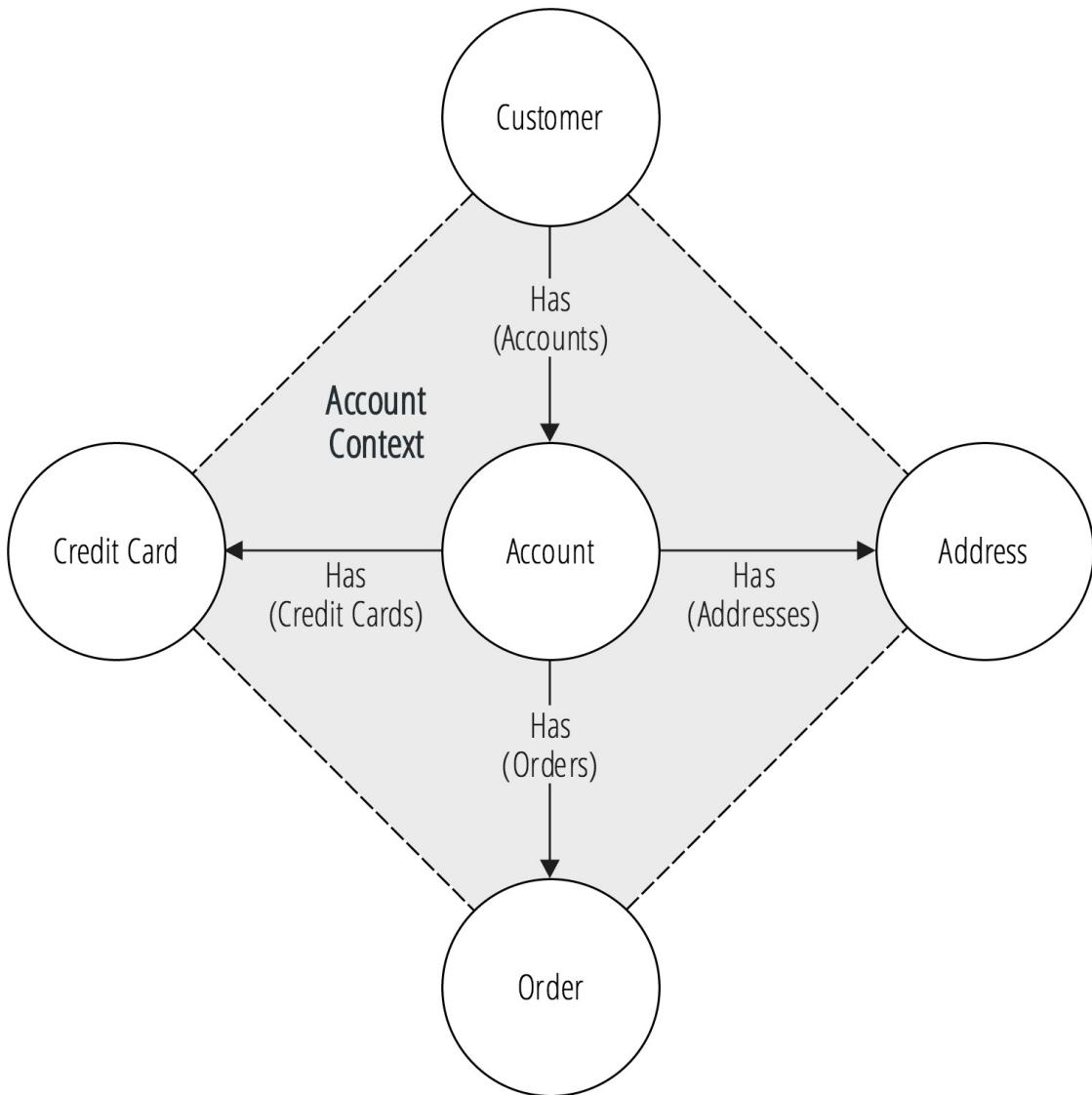


Figure 6-8. Account Context for the Account Service

The next step to create our *Account Service* is to construct our domain classes. Each Spring Data project has a slightly different process to create a domain class. For Spring Data JPA we have a few tools in our toolbox that we can use to annotate parts of our domain classes to match the domain model in the *Account Context*.

Creating Account Service's Domain Classes

Now let's create each of our domain classes for the `Account Service` as JPA entity classes using the JPA annotations described in the last table. We need to create domain classes for each of the concepts in the *Account Context*, which includes the following objects.

- Account
- Customer
- CreditCard
- Address

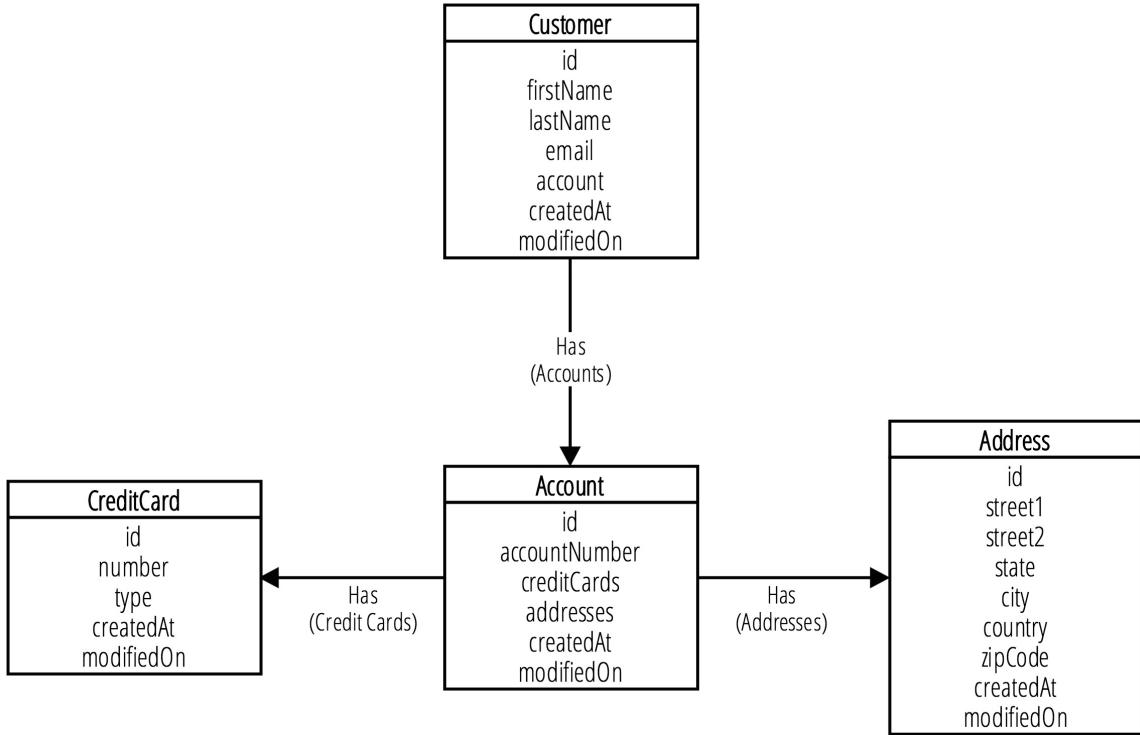


Figure 6-9. Entity Relationship Diagram for Account Context

We will create a domain class with the properties listed in the diagram above. Each domain class will have annotated properties for relationships to the Account class.

Account

Example 6-10. The `Account` domain class as a JPA entity

```

// The @Entity annotation is used mark a domain class as a JPA entity
@Entity
public class Account extends BaseEntity {

    private Long id;
    private String accountNumber;

    // The relationship from Account to CreditCards
    private Set<CreditCard> creditCards;

    // The relationship from Account to Addresses
}

```

```

private Set<Address> addresses;

public Account() {
}

public Account(String accountNumber) {
    this.accountNumber = accountNumber;
    this.creditCards = new HashSet<>();
    this.addresses = new HashSet<>();
}

@Id // The @Id annotation indicates a primary key column
@GeneratedValue(strategy = GenerationType.AUTO) ❶
public Long getId() {
    return id;
}

public String getAccountNumber() {
    return accountNumber;
}

❷
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
public Set<CreditCard> getCreditCards() {
    return creditCards;
}

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
public Set<Address> getAddresses() {
    return addresses;
}

// ... Setters omitted
}

```

❶

The `@GeneratedValue` will increment a unique ID for the `@Id` field

❷

The `@OneToMany` annotation describes a FK relationship to a JPA entity

Customer

Example 6-11. The `Customer` domain class as a JPA entity

```
public class Customer extends BaseEntity {

    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private Account account;

    public Customer() {
    }

    public Customer(String firstName, String lastName, String ema
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.account = account;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    @OneToOne(cascade = CascadeType.ALL)
    public Account getAccount() {
        return account;
    }

    // ... Other getters and setters omitted
}
```

Address

Example 6-12. The `Address` domain class as a JPA entity

```
@Entity
public class Address extends BaseEntity {

    private Long id;
    private String street1;
    private String street2;
    private String state;
```

```

private String city;
private String country;
private Integer zipCode;

@Enumerated(EnumType.STRING)
private AddressType addressType;

public Address() {
}

public Address(String street1, String street2, String state,
               String city, String country, AddressType addre
               Integer zipCode) {
    this.street1 = street1;
    this.street2 = street2;
    this.state = state;
    this.city = city;
    this.country = country;
    this.addressType = addressType;
    this.zipCode = zipCode;
}

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
public Long getId() {
    return id;
}

// ... Other getters and setters omitted

```

CreditCard

Example 6-13. The CreditCard domain class as a JPA entity

```

@Entity
public class CreditCard extends BaseEntity {

    Long id;
    private String number;

    @Enumerated(EnumType.STRING)
    private CreditCardType type;

    public CreditCard() {
    }

```

```

public CreditCard(String number, CreditCardType type) {
    this.number = number;
    this.type = type;
}

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
public Long getId() {
    return id;
}

public String getNumber() {
    return number;
}

public CreditCardType getType() {
    return type;
}

// ... Setters omitted

```

Cascading Transactions and Fetching Relationships

Notice that we have provided some parameters to the `@OneToMany` annotation.

```

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
public Set<Address> getAddresses() {
    return addresses;
}

```

- `CascadeType.ALL` indicates that committing a transaction cascades to all referenced JPA entities.
- `FetchType.EAGER` indicates that relationships will automatically be populated.

The `@OneToMany` annotation describes a foreign key relationship between tables managed by JPA entity classes. In the case of `Account`'s relationship with `Address`, we have a cardinality of `1..*` or *one-to-many*. Let's take a look at the MySQL DDL (*data description language*) query that is generated to create the relationship between `Account` and `Address`.

Example 6-14. JPA Generated DDL for Account and Address

```
CREATE TABLE account
(
    id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    created_at BIGINT,
    last_modified BIGINT,
    account_number VARCHAR(255)
);

CREATE TABLE account_addresses ①
(
    account_id BIGINT NOT NULL,
    addresses_id BIGINT NOT NULL,
    PRIMARY KEY (account_id, addresses_id)
);

CREATE TABLE address
(
    id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    created_at BIGINT,
    last_modified BIGINT,
    address_type INT,
    city VARCHAR(255),
    country VARCHAR(255),
    state VARCHAR(255),
    street1 VARCHAR(255),
    street2 VARCHAR(255),
    zip_code INT
);

ALTER TABLE account_addresses ADD FOREIGN KEY (account_id) REFERENCES account(id);
ALTER TABLE account_addresses ADD FOREIGN KEY (addresses_id) REFERENCES address(id);
CREATE UNIQUE INDEX UK_r2ahplt2rqwvx1pnd5bbo7o70 ON account_addresses
②;
```

①

A join table *account_addresses* is created to relate Account and Address records

②

A unique index is created to lookup addresses for an *account_id* key

Auditing JPA Classes

We can also apply auditing for JPA classes to record the creation date of a record as well as the time the record was last modified. This is the responsibility of the `BaseEntity` class which each of our JPA entities will inherit. Let's take a look at what the `BaseEntity` class looks like.

Example 6-15. The `BaseEntity` class provides JPA auditing

```
@MappedSuperclass ①
@EntityListeners(AuditingEntityListener.class) ②
public class BaseEntity {

    @CreatedDate ③
    private Long createdAt;

    @LastModifiedDate ④
    private Long lastModified;

    public BaseEntity() {
    }

    public Long getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Long createdAt) {
        this.createdAt = createdAt;
    }

    public Long getLastModified() {
        return lastModified;
    }

    public void setLastModified(Long lastModified) {
        this.lastModified = lastModified;
    }
}
```

①

This annotation designates a super class that JPA entities inherit from.

②

This designates an auditing callback listener that will observe the lifecycle of JPA operations.

③

This annotation will save a timestamp when a record is created.

④

This annotation will apply an updated timestamp when a record is updated.

There is one last step to enable JPA auditing for our Spring Boot application. We need to apply the `@EnableJpaAuditing` annotation to our application class for the `Account Service`.

Example 6-16. Enable JPA Auditing for a Spring Boot application

```
@SpringBootApplication
@EnableJpaAuditing
public class AccountApplication {
    public static void main(String[] args) {
        SpringApplication.run(AccountApplication.class, args);
    }
}
```

Now that we have a handle on how to create JPA entity classes with mapped super classes, we can do the same for our other domain classes in the *Account Context*. Let's now create our repositories so that we can manage our data for the *Account Service*.

JPA Repositories

Now that we have the domain classes for our *Account Service*, we need to create repositories so that we can manage the data in our MySQL database.

Example 6-17. AccountRepository Definition

```
public interface AccountRepository extends PagingAndSortingReposito
```

```
}
```

As we saw earlier, repositories, such as the `AccountRepository` interface above, are implemented at runtime and include a set of methods for managing the underlying data objects of a data store. The `PagingAndSortingRepository` provides a set of functions that include CRUD methods as well as methods to page and sort returned result sets.

Integration Testing

Now that we have our repositories in place for the `Account Service`, let's create a basic integration test to make sure it all works.

First, we'll create a new `Account` class and initialize it with the account number 12345.

Example 6-18. Create a new `Account`

```
Account account = new Account("12345");
```

Next, we need to ensure that the account is owned by a customer. Let's create a new `Customer` class and initialize it so that we can assign our new account to it.

Example 6-19. Create a new `Customer` record for Jane Doe

```
Customer customer = new Customer("Jane", "Doe", "jane.doe@gmail.c
```

Here we have created a new instance of `Customer` and initialized it for the person named Jane Doe with the e-mail address `jane.doe@gmail.com`. We've also provided the newly created `Account` class we created with account number 12345. Let's go ahead and log out the customer object to see what it looks like.

Example 6-20.

```
{
```

```

    "id": null,
    "firstName": "Jane",
    "lastName": "Doe",
    "email": "jane.doe@gmail.com",
    "account": {
        "id": null,
        "accountNumber": "12345",
        "creditCards": [],
        "addresses": [],
        "createdAt": null,
        "lastModified": null
    },
    "createdAt": null,
    "lastModified": null
}

```

Here we see the serialized JSON output of the `Customer` class we created. Notice that the `null` fields on auditing properties and the ID field have not been populated. In order to populate these fields we will need to persist the state of the `Customer` class, which will cascade to the `Account` class. Let's save the base object `Customer` using the `CustomerRepository`.

```
customer = customerRepository.save(customer);
```

Now let's see what our `Customer` object looks like after we have persisted it to the database.

Example 6-21.

```
{
    "id": 1,
    "firstName": "Jane",
    "lastName": "Doe",
    "email": "jane.doe@gmail.com",
    "account": {
        "id": 1,
        "accountNumber": "12345",
        "creditCards": [],
        "addresses": [],
        "createdAt": 1443412981435,
        "lastModified": 1443412981435
    },
    "createdAt": 1443412981430,
    "lastModified": 1443412981430
}
```

```
}
```

We can now see that the `createdAt` and `lastModified` fields for both the `Customer` and `Account` classes has been applied. We also see that the `id` properties have also been generated, corresponding to the unique primary key in the `customer` and `account` tables.

Now let's add a `CreditCard` and `Address` to the `Account` and then save the `Customer` class again.

Example 6-22. Adding `CreditCard` and `Address` to `Account`

```
// Create a new credit card for the account
CreditCard creditCard = new CreditCard("1234567801234567", Credit

// Add the credit card to the customer's account
customer.getAccount()
    .getCreditCards()
    .add(creditCard);

// Create a new shipping address for the customer
Address address = new Address("1600 Pennsylvania Ave NW", null,
    "DC", "Washington", "United States", AddressType.SHIPPING

// Add address to the customer's account
customer.getAccount()
    .getAddresses()
    .add(address);

// Apply the cascading update by persisting the customer object
customer = customerRepository.save(customer);
```

Now that we've added a new `CreditCard` and `Address` to *Jane Doe's* `Account`, both of these new objects are automatically saved in the database after persisting the `Customer` record.

Let's see what our updated `Customer` object for *Jane Doe* looks like now.

Example 6-23. JSON of Jane Doe's `Customer` record

```
{
  "id": 1,
```

```
"firstName": "Jane",
"lastName": "Doe",
"email": "jane.doe@gmail.com",
"account": {
    "id": 1,
    "accountNumber": "12345",
    "creditCards": [
        {
            "id": 1,
            "number": "1234567801234567",
            "type": "VISA",
            "createdAt": 1443412981481,
            "lastModified": 1443412981481
        }
    ],
    "addresses": [
        {
            "id": 1,
            "street1": "1600 Pennsylvania Ave NW",
            "street2": null,
            "state": "DC",
            "city": "Washington",
            "country": "United States",
            "zipCode": 20500,
            "addressType": "SHIPPING",
            "createdAt": 1443412981479,
            "lastModified": 1443412981479
        }
    ],
    "createdAt": 1443412981435,
    "lastModified": 1443412981435
},
"createdAt": 1443412981430,
"lastModified": 1443412981430
}
```

Spring Data MongoDB

The *Spring Data MongoDB* project provides repository-based data management for MongoDB.

MongoDB is a NoSQL database that is well known for its ease of use and simplified data model. MongoDB is categorized as a *document-oriented* database, which means that records are stored as hierarchical JSON-like documents.

Order Service

We are going to use *Spring Data MongoDB* and *Spring Boot* for our `Order` service, which will function as the backend service for the order context in *Cloud-native Clothing*'s domain model. Similar to the last section, we're going to walk through setting up a *Spring Data MongoDB* application. Much of what we learned creating the `Account` service will be re-applied here.

Warning

Not all Spring Data projects support running datasources an embedded in-memory database, such as with *Spring Data JPA* and H2 or HSQLDB. Since MongoDB is written in C++, it cannot be embedded as a process inside a JVM application. For this section you'll need to download and install MongoDB at <http://www.mongodb.org/>.

We can again use the *Spring Initializr* project at <http://start.spring.io> to preconfigure our project dependencies. The project dependencies we will need for *Cloud-native Clothing*'s `Order` service are in the table below.

Table 6-5. Dependencies for Cloud-native Clothing's Order Service

| Dependency | Purpose |
|------------|---------|
|------------|---------|

| | |
|-------------|---------------------------------------------|
| Spring Data | Spring Boot starter project for Spring Data |
|-------------|---------------------------------------------|

MongoDB

MongoDB

Note

Spring provides a *Spring Initializr* service at <http://start.spring.io> to help you bootstrap your Spring Boot applications. You can also follow along with the provided example projects for this chapter at <http://github.com/cloud-native-java/data>.

Let's use the following CURL command from the terminal to setup our boiler plate for the Order Service. Note that we are providing parameters for the project dependencies we listed earlier for this MongoDB project.

```
curl https://start.spring.io/starter.tgz -d dependencies=data-mon  
-d type=maven-project -d baseDir=order-service
```

Configuring Spring Data MongoDB

Since we used the Spring Initializr project to generate a Spring Boot application, we can see that our dependencies have already been configured for Spring Data MongoDB.

Example 6-24. Maven dependencies for the Order Service

```
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-mongodb</artifactId>  
  </dependency>  
  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-test</artifactId>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

Let's take a look at our main application class for the OrderApplication.

The following snippet enables the `Order Service` as a Spring Boot application with Spring Data MongoDB repositories.

Example 6-25. Enable MongoDB repositories for the Spring Boot application

```
@SpringBootApplication  
@EnableMongoRepositories ❶  
public class OrderApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(OrderApplication.class, args);  
    }  
}
```

❶

Enables Spring Data MongoDB repositories

Creating Order Service's Domain Classes

For the `Order Service` we are going to refer back to *Cloud-native Clothing's* domain model for the *Order Context*.

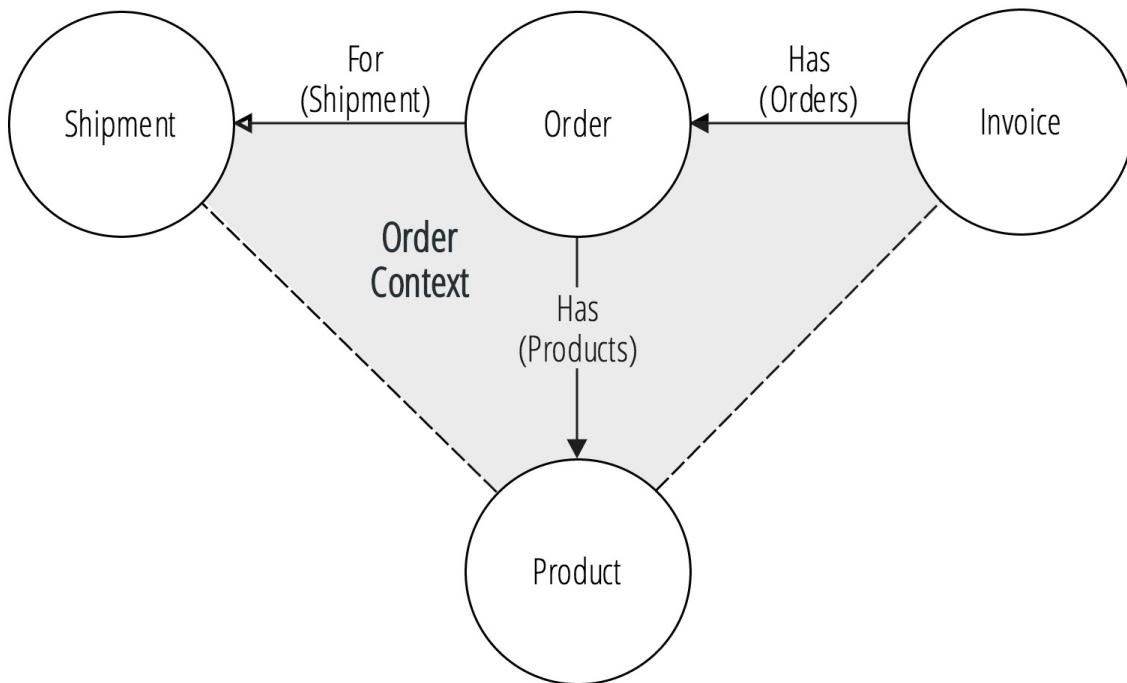


Figure 6-10. Order Context for the Order Service

In the *Order Context* we can see that we have 4 domain classes: **Invoice**, **Order**, **Product**, and **Shipment**. In order to understand how we can construct our domain classes, let's take a look at the primary use cases for the Order Service.

| Behavior | Domain Classes |
|-----------------------------------------------------------------------------------------|--------------------------|
| An order is <i>placed</i> for a set of products and their quantities. | Order, Product |
| A shipment is <i>delivered</i> to an address for an order . | Shipment, Address, Order |
| An invoice is <i>generated</i> for a set of orders of an account . | Invoice, Order, Account |

This is an oversimplified set of primary use cases than what we would normally see in an enterprise application. These three use cases describe the most salient behaviors of the `Order Service`. Notice that domain classes interact with each other. In domain-driven design it is important to break down use cases into statements that describe behaviors of an actor's interactions with domain objects.

MongoDB Document Classes

We saw earlier in the chapter that each domain class in our `Account Service` was annotated using a JPA annotation called `@Entity`. For other Spring Data projects the pattern used for domain class annotations remains the same. Each Spring Data project's semantics for annotations are specific to the traits of the model of the database. For Spring Data MongoDB we can use a `@Document` annotation on a class to indicate that it is a representation of a document in MongoDB.

Invoice

The first document class we will create is an `Invoice`. From our primary use cases we have one statement that we can use to describe how we should construct the `Invoice` domain class.

- An `invoice` is *generated* for a set of `orders` of an `account`.

The goal here is to create a model class that supports this use case's behavior without being too restrictive. When we create a new invoice in the `Order Service` we should require that the invoice can be looked up by the account number. This account number will refer back to an `Account`'s domain class that is managed by the `Account Service`. We will also provide a field for a set of orders rather than just a single order for each invoice. We also will require that a billing address is provided for the `Invoice`, this address describes where the invoice should be sent.

Note

The reason we use a document database for the order service is that it is an aggregate store. This is useful because line items in an order should not be modified after the order has been submitted by the user. The product for each line item should represent the state of the product at the time of the order.

Example 6-26. The `Invoice` domain class as a MongoDB document

```
@Document
public class Invoice extends BaseEntity {

    private String invoiceId, customerId;
    private List<Order> orders = new ArrayList<Order>();
    private Address billingAddress;
    private InvoiceStatus invoiceStatus;

    public Invoice(String customerId, Address billingAddress) {
        this.customerId = customerId;
        this.billingAddress = billingAddress;
        this.billingAddress.setAddressType(AddressType.BILLING);
        this.invoiceStatus = InvoiceStatus.CREATED;
    }

    // Getters and setters omitted...
}
```

Example 6-27. A status enum for the state of a `Invoice`

```
public enum InvoiceStatus {
    CREATED,
    SENT,
    PAID
}
```

Order

We can see from our `Invoice` domain class that we are referencing a set of `Order` classes. Let's also refer back to the primary use cases that provide us with more detail about an `Order`.

- An **order** is *placed* for a set of **products** and their quantities.

- A **shipment** is *delivered* to an **address** for an **order**.

These two statements describe several domain class interactions. To fulfill the requirements of the first statement, we will use a new object class called `LineItem` to describe a `Product` and its quantity ordered. Each `LineItem` will refer to a remote reference to a `Product` that is stored in the `InventoryService`, which we will create later in the chapter. The second use case statement describes the creation of a `Shipment` that is delivered to an `Address` for an `Order`. To fulfill this statement's requirements, we will require that an account number and shipping address is provided in the constructor of the `Order` class. Finally, we will provide a status object that describes the state of an `Order` object as it is moved from an initial state to a final state.

Example 6-28. The `order` domain class

```
@Document
public class Order extends BaseEntity {

    private String orderId;
    private String accountNumber;
    private OrderStatus orderStatus;
    private List<LineItem> lineItems = new ArrayList<>();
    private Address shippingAddress;

    public Order(String accountNumber, Address shippingAddress) {
        this.accountNumber = accountNumber;
        this.shippingAddress = shippingAddress;
        this.shippingAddress.setAddressType(AddressType.SHIPPING);
        this.orderStatus = OrderStatus.PENDING;
    }

    // Getters and setters omitted...
}
```

Example 6-29. A status enum for the state of an `order`

```
public enum OrderStatus {
    PENDING,
    CONFIRMED,
    SHIPPED,
    DELIVERED
}
```

Now we will create a `LineItem` class that describes a reference to a `Product` in the catalog and the state of that product at the time the order was placed.

Example 6-30. The `LineItem` class

```
public class LineItem {  
  
    private String name, productId;  
    private Integer quantity;  
    private Double price, tax;  
  
    public LineItem(String name, String productId, Integer quantity,  
                    Double price, Double tax) {  
  
        this.name = name;  
        this.productId = productId;  
        this.quantity = quantity;  
        this.price = price;  
        this.tax = tax;  
    }  
  
    // Getters and setters omitted...  

```

Auditing with MongoDB

Example 6-31. The auditing abstract base class

```
public class BaseEntity {  
  
    private DateTime lastModified, createdAt;  
  
    // Getters and setters omitted...  
}
```

Example 6-32. A listener can be used to audit MongoDB transactions

```
@Component  
public class BeforeSaveListener extends AbstractMongoEventListener<BaseEntity> {  
  
    @Override  
    public void onBeforeSave(BeforeSaveEvent<BaseEntity> event) {  
        // Audit logic here  
    }  
}
```

```

        DateTime timestamp = new DateTime();

        // Add a timestamp to the created date if it does not yet
        if (event.getSource().getCreatedAt() == null)
            event.getSource().setCreatedAt(timestamp);

        // Update the timestamp to the current time
        event.getSource().setLastModified(timestamp);

        super.onBeforeSave(event);
    }
}

```

Integration Tests

Now that we have created the data layer for our `Order Service`, let's walk through creating an integration test. We are going to perform the following steps in our test.

- Create a new `Order`
- Add a `LineItem` to the new `Order`
- Save the `Order` using `OrderRepository`

First, we'll create a new instance of the `Order` class and initialize it with the account number 12345 and a shipping Address.

Example 6-33. Create a new Order

```

// Create a new shipping address for the customer
Address address = new Address("1600 Pennsylvania Ave NW", null,
    "DC", "Washington", "United States", 20500);

// Create a new order
Order order = new Order("12345", address);

// Add line items
order.addLineItem(new LineItem("Best. Cloud. Ever. (T-Shirt, Men' "
    "SKU-24642", 1, 21.99, .06));

order.addLineItem(new LineItem("Like a BOSH (T-Shirt, Women's Med

```

```

    "SKU-34563", 3, 14.99, .06));

order.addLineItem(new LineItem("We're gonna need a bigger VM (T-S
    "SKU-12464", 4, 13.99, .06));

order.addLineItem(new LineItem("cf push awesome (Hoodie, Men's Me
    "SKU-64233", 2, 21.99, .06));

// Save the order
order = orderRepository.save(order);

```

Now let's take the state of the saved `Order` from the last step and generate a new `Invoice`. We will do the following steps.

- Create a new `Invoice`
- Add an `order` to the `Invoice`
- Save the `Invoice` using `InvoiceRepository`

Example 6-34. Create an Invoice

```

// Create a new invoice with the customer number and billing addr
Invoice invoice = new Invoice("918273465", billingAddress);

// Add the order to the invoice
invoice.addOrder(order);

// Save the invoice
invoice = invoiceRepository.save(invoice);

```

JSON Representation

Now let's take a look at what the representation of our invoice's object model will look like when converted to JSON. The `Invoice` model includes a representation of each of the objects we created in our integration test.

Example 6-35. JSON representation of an Invoice

```
{
  "invoiceId": "562a461b6a84baa4297fb972",

```

```
"customerId": "918273465",
"orders": [
  {
    "orderId": "562a461b6a84baa4297fb971",
    "accountNumber": "12345",
    "orderStatus": "PENDING",
    "lineItems": [
      {
        "name": "Best. Cloud. Ever. (T-Shirt, Men's Large)",
        "productId": "SKU-24642",
        "quantity": 1,
        "price": 21.99,
        "tax": 0.06
      },
      {
        "name": "Like a BOSH (T-Shirt, Women's Medium)",
        "productId": "SKU-34563",
        "quantity": 3,
        "price": 14.99,
        "tax": 0.06
      },
      ...
    ],
    "shippingAddress": {
      "street1": "1600 Pennsylvania Ave NW",
      "state": "DC",
      "city": "Washington",
      "country": "United States",
      "zipCode": 20500,
      "addressType": "SHIPPING"
    },
    "lastModified": "2015-10-23T07:37:15.595-07:00",
    "createdAt": "2015-10-23T07:37:15.558-07:00"
  }
],
"billingAddress": {
  "street1": "875 Howard St",
  "state": "CA",
  "city": "San Francisco",
  "country": "United States",
  "zipCode": 94103,
  "addressType": "BILLING"
},
"invoiceStatus": "CREATED",
"lastModified": "2015-10-23T07:37:15.599-07:00",
"createdAt": "2015-10-23T07:37:15.599-07:00"
}
```

Spring Data Neo4j

The *Spring Data Neo4j* project provides repository-based data management for Neo4j, a popular NoSQL graph database.

What is a graph database?

A graph database represents database objects as a connected graph of nodes and relationships. Nodes and relationships can contain both simple value types and complex value types.

This connected model is useful when combined with Cypher, Neo4j's declarative query language that is similar to SQL. The Cypher query language provides the ability to query and manage data using patterns, as opposed to SQL which uses a strict schema that must be defined prior to querying your data. Further, Neo4j provides the ability to define graph traversals, eliminating SQL joins that make complex queries hard to read and write.

In this section we will achieve the following goals as we use Spring Data Neo4j to create an `Inventory Service` for *Cloud-native Clothing*'s online store's backend.

- Overview of *Cloud-native Clothing*'s requirements for inventory management
- Review how Neo4j manages database objects as a connected graph
- Design a graph data model for inventory management
- Implement Spring Data domain classes as Neo4j nodes
- Create repositories to manage domain classes for Neo4j nodes

Inventory Service

We're going to break down the construction of our `Inventory Service` into two parts: design and implementation. We'll need to work through how we're going to construct a graph data model that describes our inventory context for *Cloud-native Clothing*'s online store. Let's review what the inventory context looks like from earlier.

Let's consider the example of our *Cloud-native Clothing* online store. A clothing store might have a seasonal catalog, which they can use to change the set of products on their website at the start of each quarter. Not all products in the catalog will necessarily be different. There might be some products in the store that are sold year-round. Because of this, catalogs could have a hierarchy, for example a base catalog for products offered year-round and a seasonal catalog that changes product lines each quarter.

We'll also need to connect products to the inventory over a collection of warehouses that are located around the world. Each of these warehouses may have a product from our catalog. We can track our global inventory of products and use this information to create shipments that are closest to the delivery location. This is a complex undertaking to say the least. To solve this problem we need to tackle the complexity of a connected model without spending so much time on its design.

Neo4j provides us the flexibility to model our data in an intuitive way, and we can map that data to a Spring Data application using the abstractions we have already learned in previous sections in this chapter.

Inventory Context

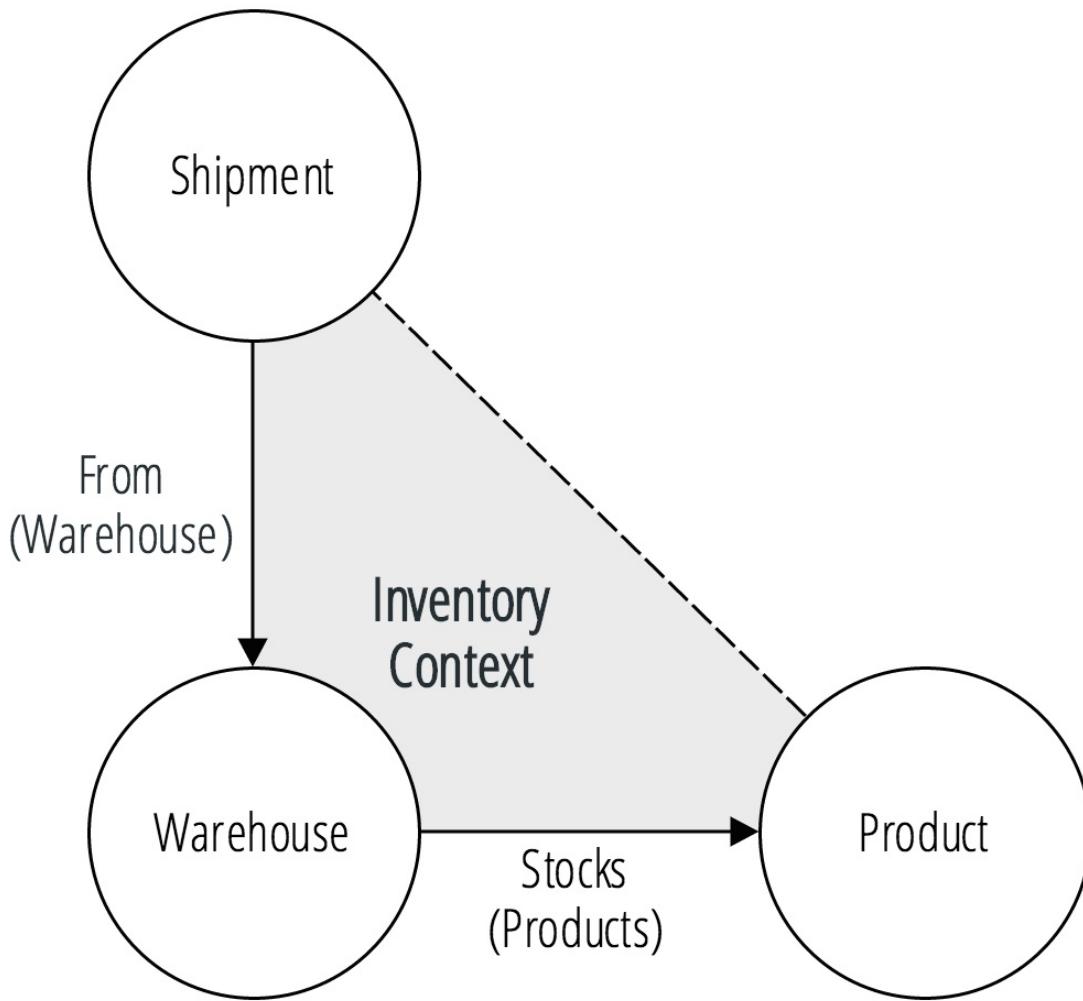


Figure 6-11. Context for the Inventory Service

The inventory context described in [Figure 6-11](#) shows a simplified view of what we will need to construct for our application. To fit this application to the requirements that we talked about earlier on, with functions such as shipping and inventory management, we'll need to design a graph model that we can use to construct our domain classes and repositories. Let's begin by doing an overview of how Neo4j's property graph data model can be used to manage connected data using Spring Data Neo4j.

Graph Data Modeling

Graph data modeling is much more intuitive than modeling domain data for a

RDBMS. In this section we're going to go over how Neo4j's property graph data model can be used to create a diagram of nodes and relationships that we will use to construct our Spring Data repositories and domain classes. Let's start by reviewing how Neo4j manages data objects as entities in a graph model.

Neo4j Entities

Neo4j has two native data entities, a *node* and a *relationship*. Nodes are used to store a map of key/value properties. Relationships also allow you to store a map of properties. Simple value types, such as a **color** or **gender**, can be stored as a property on a node. Complex value types, such as an **address**, should be themselves represented as a separate node. The relationship is used to connect nodes together, such as connecting an **address** node to a **shipment** node.

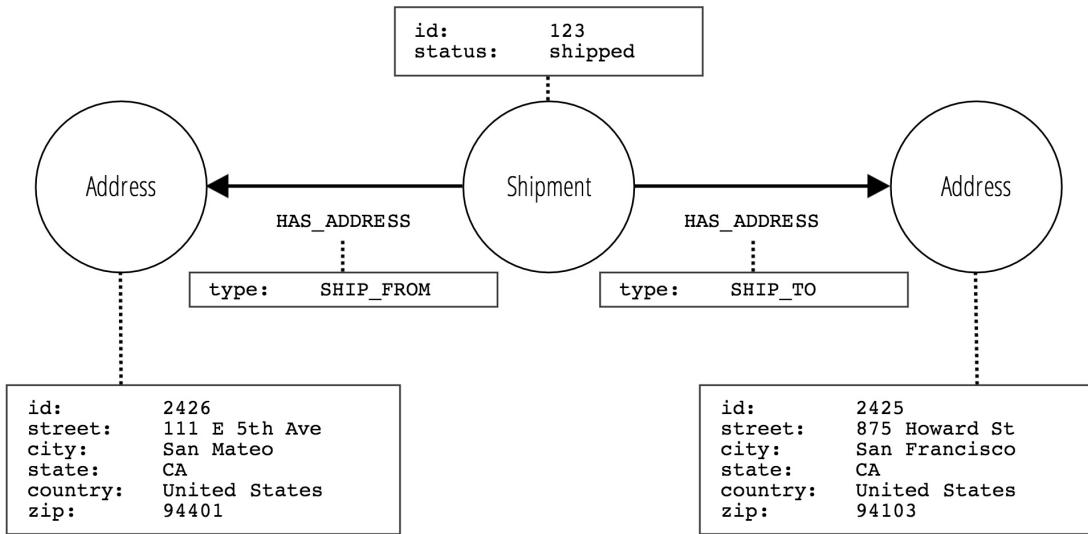


Figure 6-12. Properties on relationships are used to qualify complex value types of a node

In [Figure 6-12](#) we see a shipment node connected to two address nodes. The relationship type, **HAS_ADDRESS**, can be connected to multiple addresses. Using Cypher we can express a query to get each of a shipment's addresses.

```

MATCH (shipment:Shipment) -[r:HAS_ADDRESS]->(address)
WHERE shipment.id = 123
RETURN address

```

In this query we are first matching all shipment nodes, described by `(shipment:Shipment)`. Nodes and relationships in Cypher use ASCII art to describe graph entities as patterns. A node is encapsulated by parenthesis, with a colon symbol separating the variable that Cypher should store the results in, for example `(shipment)`, and a label that describes which group a node belongs to, such as `(:Shipment)` indicating a node has a **Shipment** label.

If we were to use the query above on the graph data described in [Figure 6-12](#), we would get back the following results.

Detailed Query Results

Query Results

```
+-----+  
| address  
+-----+  
| Node[7]{street:"875 Howard St",country:"United States",id:2425,city:"San Francisco",zip:"94103",state:"CA"} |  
| Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"} |  
+-----+  
2 rows  
54 ms
```

Here we see that we've retrieved both addresses for a specific shipment with the ID 123. What if we wanted to only query the address of the warehouse that a shipment was shipped from? We could modify the Cypher query to the following.

```
MATCH (shipment:Shipment) -[r:HAS_ADDRESS { type: "SHIP_FROM" }] ->  
WHERE shipment.id = 123  
RETURN address
```

Now we'll get back only one of the two addresses, as shown in the results below.

Detailed Query Results

Query Results

```
+-----+  
| address  
+-----+  
| Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"} |  
+-----+  
1 row  
71 ms
```

By modifying the first Cypher query to include a property type on the **HAS_ADDRESS** relationship, `() -[r:HAS_ADDRESS { type: "SHIP_FROM" }] -> ()`, which should equal **SHIP_FROM**, we are able to get back the address information of where a shipment was shipped from.

Inventory Context's Graph Model

Now that we know how Neo4j models data as graphs, we can create a graph model describing the nodes and relationships of our inventory microservice.

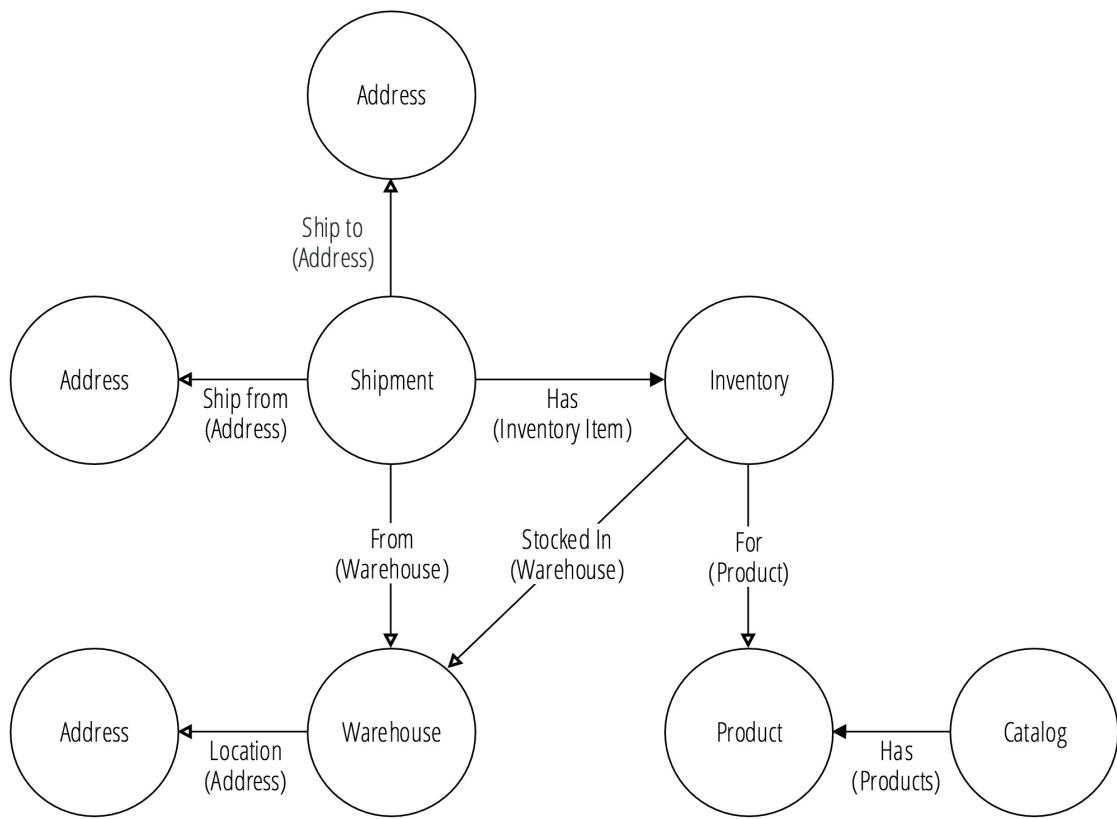


Figure 6-13. Graph model for the inventory bounded context

In [Figure 6-13](#) we see each node with its label and the relationships between the nodes, which provides us with a rough sketch of how to construct our domain classes and repositories using Spring Data Neo4j in a Spring Boot application.

Labels and Repositories

In Spring Data Neo4j, labels in our Neo4j graph model, such as the ones we have on our nodes in [Figure 6-13](#), will become our Spring Data repositories. We will have repositories for each of the following node labels, shown in our graph data model.

- Address
- Shipment

- Inventory
- Product
- Catalog
- Warehouse

Configuring Neo4j

Since Spring Data provides a project for Neo4j, we can use a set of features that take advantage of the specialized traits of a graph database.

Instead of needing to specify a database driver in my classpath like we did with MySQL, we can provide dependencies in my `pom.xml` for the Spring Data Neo4j project.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j</artifactId>
    <version>4.0.0.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Now that our Spring Boot project is configured to use Spring Data Neo4j, we can build out an application using the familiar abstractions we've used in previous examples to manage connected data in a Neo4j database.

Now we will need to create a configuration class that will be used in our Spring Boot application to externalize configuration properties to connect to an external Neo4j database.

```

@EnableNeo4jRepositories(basePackages = "demo") ❶
@Configuration
public class GraphConfiguration extends Neo4jConfiguration { ❷

    @Value("${neo4j.uri}") ❸
    private String url;

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    @Bean
    public Neo4jServer neo4jServer() {
        return new RemoteServer(url); ❹
    }

    @Bean
    public SessionFactory getSessionFactory() { ❺
        return new SessionFactory("demo.product", "demo.shipment"
            "demo.address", "demo.inventory", "demo.catalog")
    }

    @Bean
    public Session getSession() throws Exception {
        return super.getSession();
    }
}

```

❶

Indicates that Neo4j repositories are enabled for this application

❷

Extends the default Neo4j configuration class

❸

Load the Neo4j's database URI from `application.yml` properties file

❹

Create a new remote Neo4j server connection from configuration

5

TO COME

Now that we have a configuration class defined, one which will configure connection details to our Neo4j server, we can add the connection details to the project's `application.yml`.

In the `application.yml` file, we'll add the following connection details.

```
neo4j:  
  uri: http://localhost:7474
```

Domain Classes

We'll need to create domain classes for each of the following node labels described in the graph model diagram from [Figure 6-13](#).

We'll start by creating a package for each of the following items.

- Address
- Shipment
- Inventory
- Product
- Catalog
- Warehouse

Address

Example 6-36. The *Address* domain class as a Neo4j label

```

@NodeEntity
public class Address {

    @GraphId
    private Long id;

    private String street1, street2, state, city, country;
    private Integer zipCode;

    public Address() {
    }

    public Address(String street1, String street2, String state,
                   String city, String country, Integer zipCode)
    {
        this.street1 = street1;
        this.street2 = street2;
        this.state = state;
        this.city = city;
        this.country = country;
        this.zipCode = zipCode;
    }

    // ... Getters and setters omitted
}

```

Shipment

Example 6-37. The *Shipment* domain class as a Neo4j label

```

@NodeEntity
public class Shipment {

    @GraphId
    private Long id;

    @Relationship(type = "CONTAINS_PRODUCT")
    private Set<Inventory> inventories = new HashSet<>();

    @Relationship(type = "SHIP_TO")
    private Address deliveryAddress;

    @Relationship(type = "SHIP_FROM")
    private Warehouse fromWarehouse;

    private ShipmentStatus shipmentStatus;
}

```

```

public Shipment() {
}

public Shipment(Set<Inventory> inventories, Address deliveryAddress,
    Warehouse fromWarehouse, ShipmentStatus shipmentStatus) {
    this.inventories = inventories;
    this.deliveryAddress = deliveryAddress;
    this.fromWarehouse = fromWarehouse;
    this.shipmentStatus = shipmentStatus;
}

// ... Getters and setters omitted

```

Inventory

Example 6-38. The *Inventory* domain class as a Neo4j label

```

@NodeEntity
public class Inventory {

    @GraphId
    private Long id;

    private String inventoryNumber;

    @Relationship(type = "PRODUCT_TYPE", direction = "OUTGOING")
    private Product product;

    @Relationship(type = "STOCKED_IN", direction = "OUTGOING")
    private Warehouse warehouse;

    private InventoryStatus status;

    public Inventory() {
    }

    public Inventory(String inventoryNumber, Product product, Warehouse
        status) {
        this.inventoryNumber = inventoryNumber;
        this.product = product;
        this.warehouse = warehouse;
        this.status = status;
    }
}

```

```
// ... Getters and setters omitted
```

Product

Example 6-39. The *Product* domain class as a Neo4j label

```
@NodeEntity
public class Product {

    @GraphId
    private Long id;
    private String name, productId;
    private Double unitPrice;

    public Product() {
    }

    public Product(String name, String productId, Double unitPrice) {
        this.name = name;
        this.productId = productId;
        this.unitPrice = unitPrice;
    }

    // ... Getters and setters omitted
}
```

Catalog

Example 6-40. The *Catalog* domain class as a Neo4j label

```
@NodeEntity
public class Catalog {

    @GraphId
    private Long id;

    @Relationship(type = "HAS_PRODUCT", direction = "OUTGOING")
    private Set<Product> products = new HashSet<>();

    private String name;

    public Catalog() {
    }
}
```

```
public Catalog(String name) {  
    this.name = name;  
}  
  
// ... Getters and setters omitted
```

Warehouse

Example 6-41. The *Warehouse* domain class as a Neo4j label

```
@NodeEntity  
public class Warehouse {  
  
    @GraphId  
    private Long id;  
  
    private String name;  
  
    @Relationship(type="HAS_ADDRESS")  
    private Address address;  
  
    public Warehouse() {  
    }  
  
    public Warehouse(String name) {  
        this.name = name;  
    }  
  
    // ... Getters and setters omitted
```

Repositories

Spring Data Neo4j uses the same abstractions for repositories that we've used in previous examples in this chapter. We'll again use the `PagingAndSortingRepository` abstraction in our inventory service to indicate that we would like Spring Data to manage a repositories with built-in paging and sorting features.

Example 6-42. Paging and sorting repository for the Address label in Neo4j

```
package demo.address;
```

```
import org.springframework.data.repository.PagingAndSortingReposi  
public interface AddressRepository extends PagingAndSortingReposi  
}
```

In each package, containing a domain class from our graph model, we'll create a repository so we can manage our domain data. For example, the snippet from [Example 6-42](#), is a repository definition that would be located in the *address* package of the project. This `AddressRepository` will allow us to manage data for our nodes in Neo4j with the label `Address`.

Our package structure for the Inventory Service project should now look like the packages in [Figure 6-14](#).

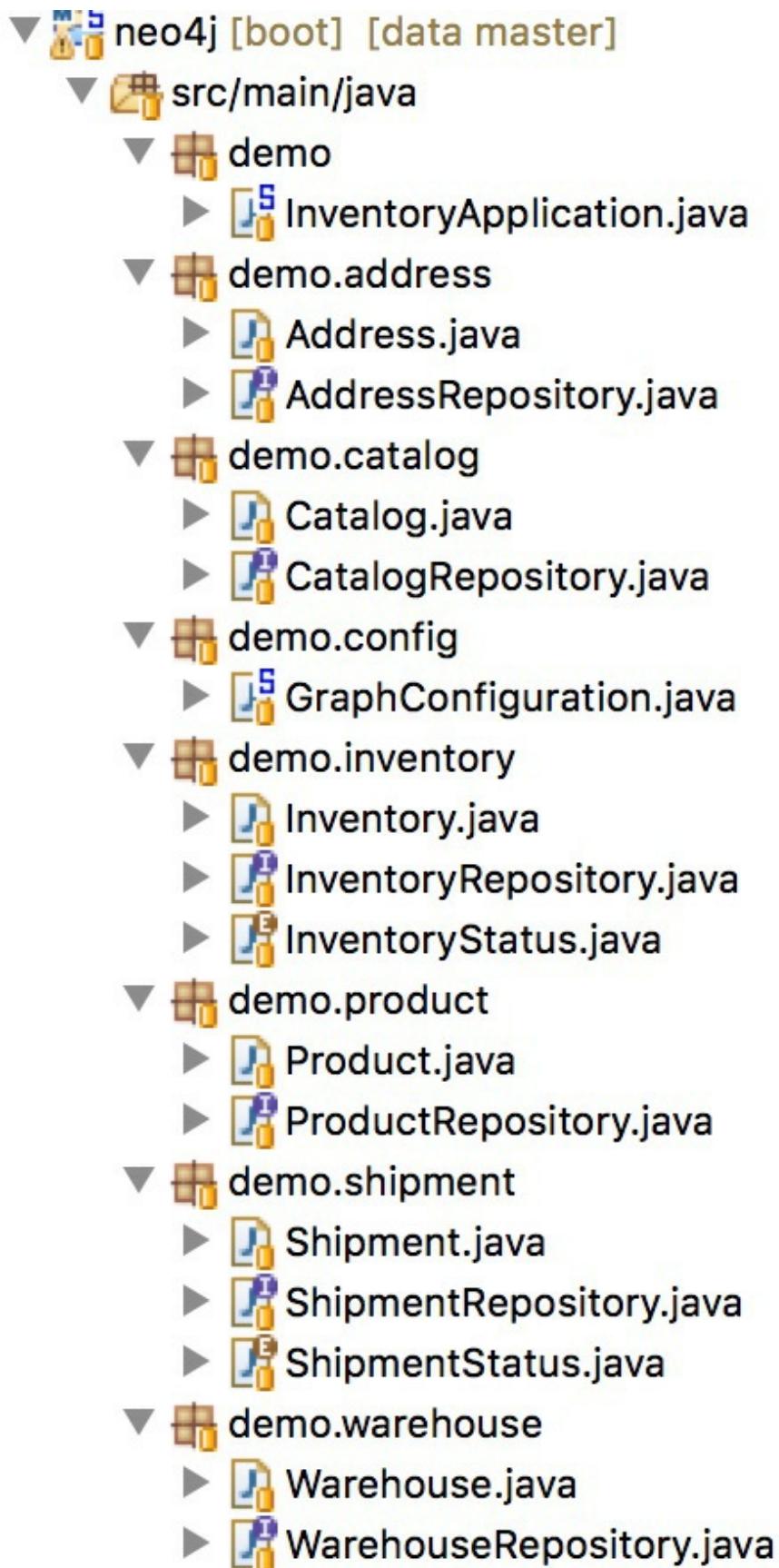


Figure 6-14. Package structure of the inventory service

Integration Testing

Now that we have created the data layer for our `Inventory Service`, let's walk through creating a simple end to end integration test. In our test, we will take the following steps.

- Create a Warehouse
- Create a list of Products
- Create a Catalog
- Create Shipment Addresses
- Create Inventory for Products
- Create a Shipment of Product Inventory

Example 6-43. A basic integration test for the Inventory Service

```
@Test
public void inventoryTest() {

    // Create a Warehouse
    Warehouse warehouse = new Warehouse("Pivotal SF");

    // Create a list of Products
    List<Product> products = Arrays.asList(
        new Product("Best. Cloud. Ever. (T-Shirt, Men's Large)", "SKU-1"),
        new Product("Like a BOSH (T-Shirt, Women's Medium)", "SKU-2"),
        new Product("We're gonna need a bigger VM (T-Shirt, Women's Large)", "SKU-3"),
        new Product("cf push awesome (Hoodie, Men's Medium)", "SKU-4")
            .stream()
            .collect(Collectors.toList()));

    productRepository.save(products);

    Product product1 = productRepository.findOne(products.get(0)).
```

```

assertThat(product1, is(notNullValue()));
assertThat(product1.getName(), is(products.get(0).getName()))
assertThat(product1.getUnitPrice(), is(products.get(0).getUni

// Create a Catalog
Catalog catalog = new Catalog("Fall Catalog");

catalog.getProducts().addAll(products);

catalogRepository.save(catalog);

Catalog catalog1 = catalogRepository.findOne(catalog.getId())

assertThat(catalog1, is(notNullValue()));
assertThat(catalog1.getName(), is(catalog.getName()));

// Create a warehouse address
Address warehouseAddress = new Address("875 Howard St", null,
    "CA", "San Francisco", "United States", 94103);

// Create a shipment address
Address shipToAddress = new Address("1600 Amphitheatre Parkwa
    "CA", "Mountain View", "United States", 94043);

addressRepository.save(Arrays.asList(warehouseAddress, shipTo

Address address1 = addressRepository.findOne(shipToAddress.ge

assertThat(address1, is(notNullValue()));
assertThat(address1.toString(), is(shipToAddress.toString()));

Address address2 = addressRepository.findOne(warehouseAddress

assertThat(address2, is(notNullValue()));
assertThat(address2.toString(), is(warehouseAddress.toString()

warehouse.setAddress(warehouseAddress);
warehouse = warehouseRepository.save(warehouse);

Warehouse warehouse1 = warehouseRepository.findOne(warehouse.

assertThat(warehouse1, is(notNullValue()));
assertThat(warehouse1.toString(), is(warehouse.toString()));

final Warehouse finalWarehouse = warehouse;

// Create a new set of inventories with a randomized inventor

```

```
Set<Inventory> inventories = products.stream()
    .map(a -> new Inventory(IntStream.range(0, 9)
        .mapToObj(x -> Integer.toString(new Random())
            .collect(Collectors.joining(""))), a, finalWar
    .collect(Collectors.toSet()));

inventoryRepository.save(inventories);

Shipment shipment = new Shipment(inventories, shipToAddress,
    warehouse, ShipmentStatus.SHIPPED);

shipmentRepository.save(shipment);

// Create a Shipment of Product Inventory
Shipment shipment1 = shipmentRepository.findOne(shipment.getI

assertThat(shipment1, is(notNullValue())));
assertThat(shipment1.toString(), is(shipment.toString()));
}
```

Spring Data Redis

The Spring Data Redis project provides Spring framework integration for the Redis key-value store.

Redis is an open source NoSQL database and is one of the most popular key-value stores in use today. While Redis is categorized as a key-value store, it can be best explained as an in-memory *data structure store*. Redis differs from most other data stores because it is able to store different kinds of complex data structures as values. What makes Redis most interesting is that it provides different sets of operations for each kind of data structure that it supports.

The data structures that Redis supports are:

- Strings
- Lists
- Sets
- Hashes
- Sorted sets
- Bitmaps and HyperLogLogs

The benefit of using a distributed data store — one designed for operations on complex data structures — is that multiple processes, applications, and servers, are able to concurrently operate on values with the same key.

Typically to do this from multiple applications requires some form of deserialization into a language's supported data structures, for instance a list or set, before operating on its value.

Redis solves this problem by providing programmatic access to perform operations on its supported data structures as an API. This means that

operations on its data structures can be applied atomically with a greater degree of transactional granularity than using serialization.

Redis is also often used for inter-process communication and messaging. In addition to specialized support for operating on data structures, Redis can be utilized as a message broker, implementing the publish/subscribe messaging pattern.

Caching

The most popular use case for Redis is caching. Spring Data Redis implements Spring framework's `CacheManager` abstraction, which makes it an excellent choice for centralized caching of records in a microservice architecture. In this section we're going to explore a Spring Boot application that uses *Spring Data Redis* to manage a cache of user records.

Architecture

As we can see in [Figure 6-15](#), the *User Service*, colored in green, is a Spring Boot application and will have the responsibility of managing resources for the `User` domain class. We can also see that there is a *User Web* application, colored in blue. This Spring Boot application will be a consumer of the *User Service* and take the role of a web application that depends on the *User* resource that is managed by the *User Service*.

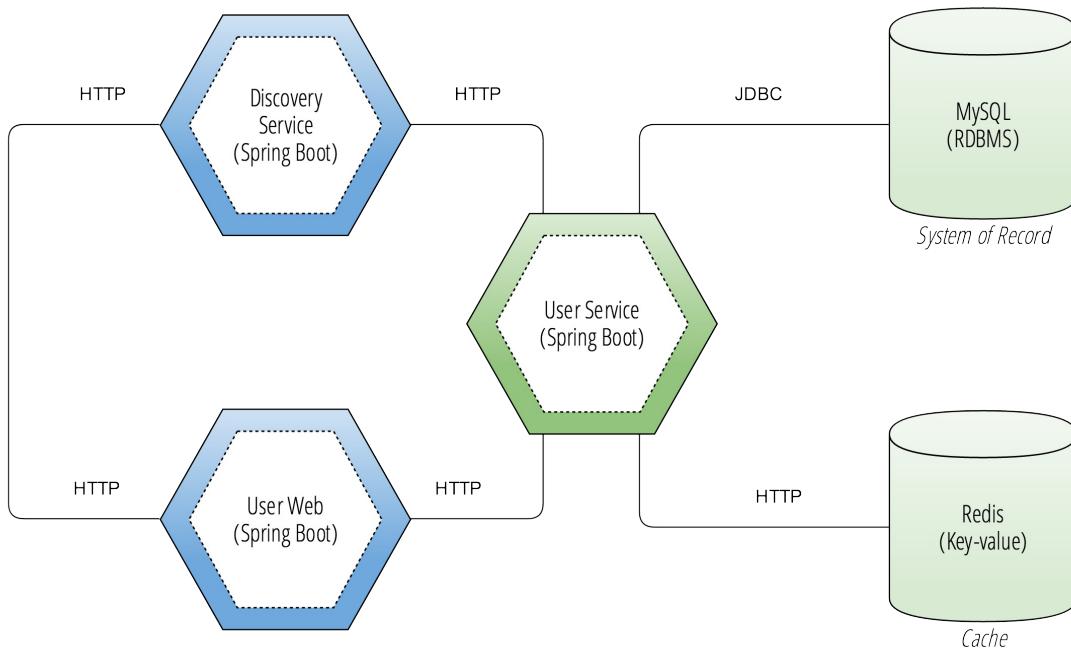


Figure 6-15. A service architecture diagram for the *User Service* and *User Web* applications

The *User Service* will provide a REST API to allow consumers to remotely manage *User* resources over HTTP. The *User Web* application will assume the role of a front-end application that hosts a single page application for managing users. This Spring Boot application will host static HTML/CSS and JavaScript and consume a REST API that is exposed from the *User Service*.

Note

The color blue indicates a service that is stateless, with no dependency on an attached database. The color green is a service that has one or more connections to a database provider and is exclusively managing persistence of a set of resources, in this case the *User* resource.

When adding a caching layer to our Spring Boot applications, we have to consider many different scenarios of how our data will be used. Caching is an important part of any microservice architecture, since there will be a great deal of demand for interaction with resources for each service that has a REST API. We can see from [Figure 6-15](#) that we have two data stores, one that is a

MySQL database and one that is a Redis server. Records will be stored and persisted to disk from the *User Service* to its attached MySQL database. When a record is requested from another service, the response will become cached by replicating the record to the Redis server, where it will be available in-memory and served to consumers for a prescribed expiration before being cached again.

The goal of caching is to increase performance and to offload database hits and connection pool utilization from our primary data store by using a secondary high availability store. Consider the scenario where we have a limited number of database connections to our MySQL database. If we exceed the number of connections that our database will allow, the database could go down or not be available until connections are recycled and released back to the pool. To solve this problem, we can use a secondary store that stores records in-memory so that a majority of traffic will be served from the in-memory cache of records, saving database connections and hits to our primary data store. This is essential in a microservice architecture where resources must be both performantly served to consumers as well as durably persisted to disk.

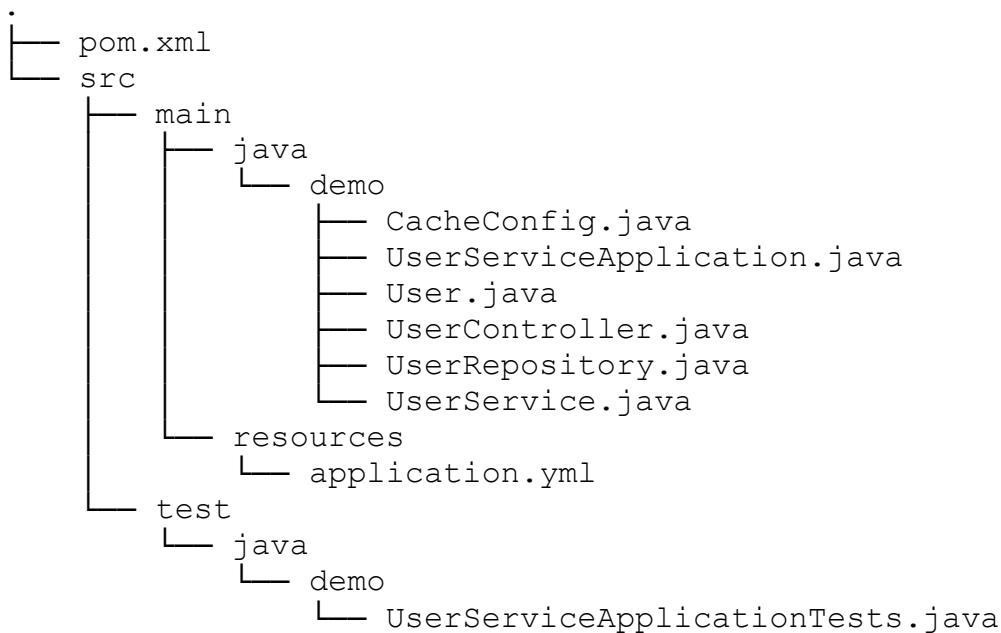
Let's take a look at the operations that we can perform on the *User* resource.

Table 6-6. User Service's HTTP routes

| Name | Mapping | Method |
|------------|-------------|--------|
| createUser | /users | POST |
| getUser | /users/{id} | GET |
| updateUser | /users/{id} | PUT |
| deleteUser | /users/{id} | DELETE |

We see here that we have a basic CRUD repository that we can use to manage *User* resources over HTTP. For this example, I've chosen to write a REST API as a controller instead of using the *Spring Data REST repositories* project. Let's take a look at what the project structure looks like for the *User Service* application.

Example 6-44. Structure of the *User Service* Spring Boot application



In [Example 6-44](#), we see the directory structure of the *User Service*, a basic Spring Boot application. Let's do an overview of each of the classes in the `demo` package of this project.

Table 6-7. Classes in the User Service's demo package

| Class | Purpose |
|-------------------------------------|--------------------------------------------------------|
| <code>CacheConfig</code> | Configures the Redis connection to be used for caching |
| <code>UserServiceApplication</code> | The Spring Boot application class |

| | |
|----------------|--------------------------------------------------------------------------------------------|
| User | The <i>User</i> domain class, configured as a JPA entity |
| UserController | The REST controller for managing <i>User</i> resources over HTTP |
| UserRepository | The Spring Data JPA repository for managing user records in H2 or MySQL |
| UserService | The service used by the UserController to manage <i>User</i> records with caching in Redis |

In [Table 6-7](#) we can see the purpose of each of the classes in the `demo` package of the *User Service*. We’re going to explore the contents of each one of these classes to see how caching using Redis can be added on top of another data store used as a system of record, in this case MySQL.

The first class we are going to explore is the `CacheConfig` class. This class is responsible for configuring the Redis connection and to initialize a `CacheManager` implementation for Redis, which is provided as a part of the *Spring Data Redis* project as `RedisCacheManager`.

Example 6-45. Class for using a configured `RedisCacheManager` and `RedisTemplate`

```
@Configuration
@EnableCaching ①
public class CacheConfig {

    @Bean
    public JedisConnectionFactory redisConnectionFactory(
        @Value("${spring.redis.port}") Integer redisPort,
        @Value("${spring.redis.host}") String redisHost) { ②
        JedisConnectionFactory redisConnectionFactory = new Jedis

        redisConnectionFactory.setHostName(redisHost);
        redisConnectionFactory.setPort(redisPort);
```

```

        return redisConnectionFactory;
    }

@Bean
public RedisTemplate redisTemplate(RedisConnectionFactory cf)
    RedisTemplate redisTemplate = new RedisTemplate();
    redisTemplate.setConnectionFactory(cf);
    return redisTemplate;
}

@Bean
public CacheManager cacheManager(RedisTemplate redisTemplate)
    RedisCacheManager cacheManager = new RedisCacheManager(re
    cacheManager.setDefaultExpiration(3000);
    return cacheManager;
}
}

```

❶

Enables Spring's annotation-driven cache management

❷

Externalizes the connection to the Redis database server

❸

Initializes the `RedisTemplate` bean, using the configured connection

❹

Configures the `CacheManager` to use the configured `RedisCacheManager` implementation

In [Example 6-45](#) we can see that we are using the `@EnableCaching` annotation on a configuration class, which will indicate that we would like to configure a `CacheManager` bean for our application. Since we would like to use Redis for caching, we can use the `RedisCacheManager` implementation and provide our `RedisTemplate` that has been configured with our externalized configuration properties in this class. Finally, we set a default expiration time of cached records, in this case we're setting the value to 3 seconds. After a record is placed into the cache, it will expire 3 seconds later.

This means that a database hit to MySQL will be made at the most, every 3 seconds for a single record.

Tip

Make sure to scrutinize the TTL expiration time that you're configuring for resources in a microservice architecture. When you have many Spring Boot applications in your architecture, there will be some cases where records are rarely updated but requested frequently. In this case, you'll want to make sure to lengthen your expiration time for these types of objects. For resources that change often and are requested frequently, you'll want to set a lower expiration time, especially if those records are being modified without cache eviction.

Example 6-46. User domain model as a JPA entity class

```
@Entity
public class User implements Serializable {

    @Id
    private String id;
    private String firstName;
    private String lastName;

    // Getters and setters are omitted...
}
```

In [Example 6-46](#), we see a basic POJO that represents our `User` domain class. This domain class is annotated using the `@Entity` annotation, which means that it is a JPA entity class. For the *User Service*, we use a primary data store as our system of record, and a secondary data store for our caching layer. For the primary data store, we'll use a MySQL database. Similar to the example from the Spring Data JPA section, we will use an embedded H2 SQL database for our *test* profile, to aid us with our unit testing. We will then make sure to configure the service to use a MySQL database connection in our *development* profile.

Now we'll take a look at the `UserController` class, which will expose a REST API for interacting with our `User` resources of the *User Service*.

Example 6-47. UserController provides a REST API to manage user resources

```
@RestController
public class UserController {

    private UserService userService;

    @Autowired
    public UserController(UserService userService) { ❶
        this.userService = userService;
    }

    ❷
    @RequestMapping(path = "users", method = RequestMethod.POST,
    public ResponseEntity<User> createUser(@RequestBody User user) {

        Assert.notNull(user);
        return Optional.ofNullable(userService.createUser(user))
            .map(result -> new ResponseEntity<>(result, HttpStatus
            .orElse(new ResponseEntity<>(HttpStatus.CONFLICT))
    }

    ❸
    @RequestMapping(path = "users/{id}", method = RequestMethod.GET)
    public ResponseEntity<User> getUser(@PathVariable(value = "id")

        return Optional.ofNullable(userService.getUser(id))
            .map(result -> new ResponseEntity<>(result, HttpStatus
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND))
    }

    ❹
    @RequestMapping(path = "users/{id}", method = RequestMethod.PUT)
    public ResponseEntity updateUser(@PathVariable(value = "id")
                                    @RequestBody User user) {

        Assert.notNull(user);
        user.setId(id);
        return Optional.ofNullable(userService.updateUser(id, user))
            .map(result -> new ResponseEntity(HttpStatus.NO_CONTENT
            .orElse(new ResponseEntity(HttpStatus.NOT_FOUND)))
    }

    ❺
    @RequestMapping(path = "users/{id}", method = RequestMethod.DELETE)
    public ResponseEntity deleteUser(@PathVariable(value = "id")
```

```
        return Optional.ofNullable(userService.deleteUser(id))
            .map(result -> new ResponseEntity<>(result, HttpStatus.OK))
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND))
    }
}
```

①

Injects the `UserService` that will manage `User` records with annotation-driven caching

②

Creates a new user, which must have a unique ID

③

Gets an existing user, which will be retrieved from the Redis cache

④

Updates an existing user, evicting any record with the same key from the Redis cache

⑤

Deletes an existing user, deleting the record and evicting it from the Redis cache

In [Example 6-47](#) we see contents of the `UserController` class. This class provides basic CRUD data management methods for the `User` domain class. Here we are using `ResponseType` and the Java 8 `Optional` class, which can be seen in use in each of our `@RequestMapping` methods.

We also see that the `Optional.ofNullable` function is used to check for `null` values that are returned from the `UserService`. When a `null` value is returned from the `UserService`, we will instead return a `ResponseType` initialized with a `HttpStatus` code that describes what the `null` value means for each CRUD operation. In each of the cases for this controller, we can see that a `null` value returned from the `UserService` means that the user record did not exist or that there was a conflict in creating a new user with the same identifier.

Tip

When caching a `ResponseEntity` by applying cache annotations directly on MVC controller methods instead of using a service, it's important to remember that the full HTTP response will be cached in Redis. This means that any headers for the Spring MVC controller's cacheable result will be served from the Redis cache. Make sure you think about caching early on when creating a service, making it easier to be mindful of how caching will impact your application when implementing new functionality.

Now let's take a look at the `UserService` class, which implements the logic for each of the REST API methods for the `User` resource from the `UserController`. This class will also contain our caching annotations for managing the insertion, retrieval, and eviction of replicated `User` records from MySQL into Redis.

Example 6-48. `UserService` manages `User` records with annotation-driven cache operations

```
@Service
public class UserService {

    private UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @CacheEvict(value = "user", key = "#user.getId()") ❶
    public User createUser(User user) {

        User result = null;

        if (!userRepository.exists(user.getId())) {
            result = this.userRepository.save(user);
        }

        return result;
    }

    @Cacheable(value = "user") ❷
}
```

```

public User getUser(String id) {
    return this.userRepository.findOne(id);
}

@CachePut(value = "user", key = "#id") ❸
public User updateUser(String id, User user) {

    User result = null;

    if (userRepository.exists(user.getId())) {
        result = this.userRepository.save(user);
    }

    return result;
}

@CacheEvict(value = "user", key = "#id") ❹
public boolean deleteUser(String id) {

    boolean deleted = false;

    if (userRepository.exists(id)) {
        this.userRepository.delete(id);
        deleted = true;
    }

    return deleted;
}
}

```

❶

Evicts the `User` record with the supplied ID from the Redis cache

❷

Gets a cached record from Redis or puts a `User` record from MySQL into the Redis cache

❸

Evicts a `User` record and replaces it with the newly updated `User` record

❹

Evicts the `User` record with the supplied ID from the Redis cache

In [Example 6-48](#) we have our `UserService` class. This class is responsible for performing cache operations as well as to manage database records for the `User` entity class in MySQL. This service is using annotation-driven caching to manage the lifecycle of a replicated record in Redis. The 4 CRUD operations in this class are fairly simple. The primary key from the `User` record is being used as the generated key for the cache manager provided for Redis. The caching annotations are using Spring Expression Language (SpEL) to access the key from the parameters of the methods.

Warning

When sharing a Redis server as a shared cache for different services, it's important to remember that the generation of the key will not include a namespace that includes the service name. This means that if you have an identifier that is the same for different domain concepts that are resources managed by different services, there will be a collision that will cause one sensitive resource to be substituted for another. In this case, you might have a service that returns an `Account` record when the request was for a `User` record. Always be careful when sharing caches. Make sure to generate keys that use a unique namespace for the service that owns a cached record.

Next Steps

In this chapter we explored how to create Spring Boot applications that take advantage of various Spring Data projects. We created a full backend for *Cloud-native Clothing*'s online store, using different Spring Data Projects for each application.

- Account Service (JPA — MySQL)
- Order Service (MongoDB)
- Inventory Service (Neo4j)

Now we will need to turn each of these examples into microservices, exposing resources from the domain as HTTP resources. After we have enabled each service to use Spring Data REST, we can add Spring Cloud to transform our services into Cloud Native microservices.

Chapter 7. Data Integration

A cloud native architecture is one that is meant to survive and thrive in the elastic world of the cloud. It is one that is designed to support ease of iteration and independent deployability. This implies process distribution and network partitions. In the [Managing Data chapter](#), we looked at how to stand up bounded contexts based on particular data technologies and expose them as services. The question then becomes, how do these nodes communicate? How do they agree upon state?

In this chapter, we'll look at a few different ways, old and new, to take data from different microservices and integrate them. One of the key concerns we'll try to address is integrity of the data in the face of distribution. The distributed systems literature is vast and comprehensive. There are seminal papers, such as “Managing Conflicts: The Bayou Distributed Database,” which break down the problem of consistency in a distributed database architecture. There is also Eric Brewer’s “CAP Theorem.” The CAP theorem states that any distributed system can have at most two of three desirable properties:

- consistency (C), which equivalent to having a single up-to-date copy of the data
- high availability (A) of that data (for updates)
- tolerance to network partitions (P)

In practice, CAP only prohibits perfect availability *and* consistency in the presence of partitions. It is rare, however, to need perfect availability *and* consistency. Instead, we'll look at patterns that allow us to arrive at a consistent state and we'll look at ways to integrate failure management patterns, or compensatory actions, in the case of failures.

The nature of the data sets with which we work has evolved to match the nature of our applications and the economics that inform their architecture,

from traditional workday-oriented, offline, batch and finite workloads to international, 24/7, always-on and infinite event and stream-based workloads.

Distributed Transactions

At first blush, we might turn to distributed transactions to guarantee consistency between services. Distributed transactions are supported in Java through the JTA API. JTA is an implementation of the X/Open XA architecture. In a 2-phase commit transaction, there's a coordinator that enlists resources in a transaction. Each resource is then told to prepare to commit. The resources respond that they're ready to proceed and then, finally, are asked to commit, all at the same time. If every resource replies that they've committed then the transaction is a success. If not, the resources are asked to rollback.

Distributed transactions are a poor fit in a cloud native world. The transaction manager is a single point of failure. It's also a very chatty single point of failure, requiring a lot of communication between each node before a transaction can be effectively handled. This communication can needlessly overwhelm networks. Distributed transactions also require that all participating resources be aware of the transaction coordinator, which isn't likely in a system full of REST APIs.

The Saga Pattern

In a sufficiently distributed system, where work spans multiple nodes and failures are normal, some activities will fail and yield inconsistent outcomes. The saga pattern was originally designed to handle long-lived transactions *on the same node*. A traditional long-lived transaction is a bottleneck in a distributed system as resources must be maintained for the duration of the transaction. These resources are then unavailable to the system. This effectively gates the system-wide throughput of a system.

Hector Garcia-Molina introduced the saga pattern in 1987. A saga is a long lived transaction that can be written as a sequence of sub-transactions that may be interleaved. It's a collection of sub-transactions (or, more usefully, we can call them *requests* in a distributed system). The transactions need to be interleavable; they can't depend on another. Each transaction must also define a compensatory transaction that undoes the transaction's effect, returning the system to a semantically consistent state. These compensatory transactions are developer-defined. This requires a bit of forethought in designing a system to ensure that the system is always in a semantically consistent state. The saga pattern trades off consistency for availability; the side effects of each subtraction are visible to the rest of the system before the whole saga has completed,

A saga is based on the idea of a saga execution coordinator. A saga execution coordinator is the keeper of the saga log. The saga log is a log that keeps trace of ongoing transactions and records progress durably. A saga execution coordinator keeps no state, however. Should there be a failure, then it's trivial to spin up a new saga execution coordinator to replace it. The SEC keeps track of which transactions in a saga are in flight and how far they've progressed. If a saga transaction fails, the SEC must start the compensatory transaction. If the compensatory transaction, the SEC will retry it. This has a few implications for our architecture. Saga transactions should be designed for at-most-once semantics. Compensatory saga transactions should be designed for at-least-once semantics; they must be idempotent and leave no observable side-effects if they're executed multiple times.

Batch workloads with Spring Batch

Batch processing has a long history. Batch processing refers to the idea that a program processes *batches* of input data at the same time. Historically, this represented a more efficient way of utilizing computing resources, amortizing the cost of machine(s) over interactive work (when operators were using the machine) and non-interactive work in the evening hours, when the machine would otherwise be idle. Today, in the era of the cloud with virtually infinite and ephemeral computing capacity, efficient utilization of a machine isn't a particularly compelling reason to adapt batch processing.

Batch processing also offers a very efficient way to work large data sets. Sequential data - SQL data, .csv files, etc. - in particular, lends itself to being processed in batches. Expensive resources, like files, SQL table cursors and transactions, may be preserved over a chunk of data, allowing processing to continue more quickly.

Batch processing also supports the logical notion of a *window* - an upper and lower bound that delimits one set of data from another. Perhaps the window is temporal: all records from the last 60 minutes, or all logs from the last 24 hours. Perhaps the window is logical; the first 1,000 records, or all the records with a certain property. Windows are useful ways to break down large data sets.

If your data lends itself to definition in terms of a window, then it's possible to process that window in smaller chunks. A chunk is an efficient, albeit resource-centric, division of a batch of data. Suppose you want to visit every record in a product catalog database spanning 1,000,000,000 rows. It would be naive to `select * from PRODUCT`, as you'd very quickly overwhelm most systems. Instead, visit a thousand (or ten thousand!) records at a time. Process them in a chunk, then move forward.

Batch provides processing efficiencies, assuming your system can tolerate slightly stale data. Many systems can; you probably won't need that rollup report for the last week's sales until the end of the week, for example.

Enter Spring Batch. Spring Batch is a framework that's designed to support processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It has become a de-facto standard for batch processing on the JVM, even becoming the inspiration for the Java EE JBatch specification. It has at its heart the notion of a `Job` which in turn might have multiple `Step`s which in turn have optional `ItemReader`, `ItemProcessor` and `ItemWriter`.

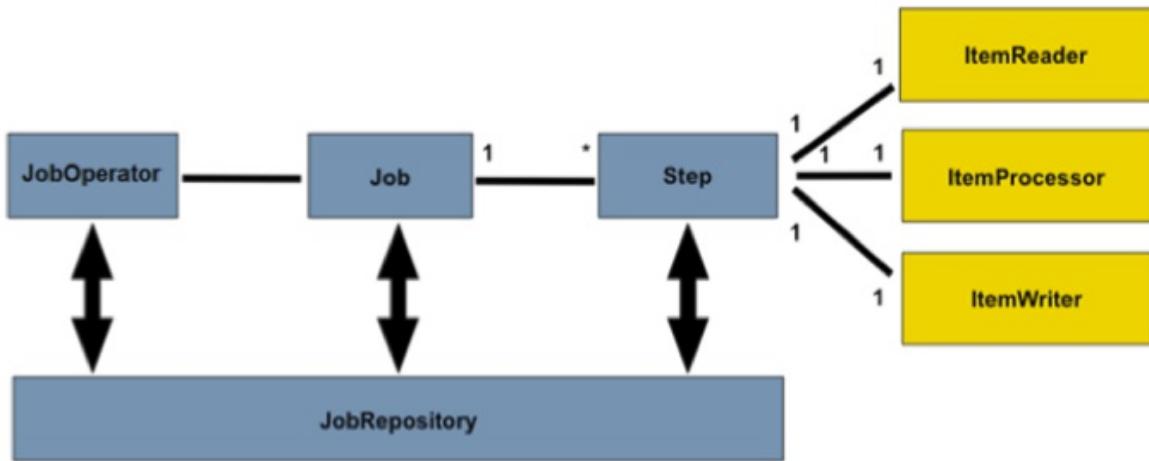


Figure 7-1. The domain of a Spring Batch job

Batch jobs have multiple steps. A step is meant to do some sort of preparation, or staging, of data before it's sent off to the next step. You may guide the flow of data from one step to another with routing logic - conditions, concurrence and basic looping. Steps might simply define generic business functionality in a *tasklet*. In this case you're using Spring Batch to orchestrate the sequence of execution. Steps may also define `ItemReader`, `ItemProcessor` and `ItemWriter` implementations for a more defined processing.

An `ItemReader` takes input from the outside world (`.csv` or XML documents, SQL databases, directories, etc.) and then adapts it into something that we can work with logically in a job, the *item*. The item can be anything; it could be a record from a database or could be a paragraph from a text file or it could be record from a `.csv` file or a stanza in an XML document. Typically Spring Batch provides lots of useful existing, out-of-the-box `ItemReader`. The results of an item here is an item that extends into the next component

which is optionally an item processor. Item processor is typically something that you would provide. Processor takes the results and the results from the item meter does something with it and then send it off optionally to an item writer. Item meters read one item at a time. Those items and gets into processor. Finally multiple items are accumulated until this time as it reaches the specified chunk size. Finally all accumulated records are sent to an ItemWriter. Bring batch to be provided the item writer that you're going to need out-of-the-box. The contract for the item leader item writer and processor are very simple.

Example 7-1. a trivial job with one step

```
package processing;

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

@SpringBootApplication
public class BatchApplication {

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

Spring Batch is stateful; it keeps metadata tables for all of the jobs running in a database. The database records, among other things, how far along the job is, what its exit code is, whether it skipped certain rows (and whether that aborted the job or it was just skipped). Operators (or autonomous agents) may use this information to decide whether to re-run the job or to intervene manually.

It's dead simple to get everything running. When the Spring Boot auto-configuration for Spring Batch kicks in, it looks for a `DataSource` and attempts to create the appropriate metadata tables based on schema on the classpath, automatically.

In the example, we configure two levels of fault tolerance: we configure that a certain step could be retried, two times, before it's considered an error. In our example, we're using a third party webservice which may or may not be

available. You can simulate the service's availability by turning off your machine's internet connection. You'll observe that it'll fail to connect, throw an `HttpStatusCodeException` subclass, and then be retried. We also want to skip records that aren't validated, so we've configured a skip policy that, whenever an exception is thrown that can be assigned to the `InvalidEmailException` type, skips the processing on that row. You'll observe that the `jlong` record fails to validate and isn't observed in the resulting writes to the database.

Once we've done the work of chunking our dataset, we can often benefit from concurrency, handling multiple chunks in concurrence on the same node or arming the processing out to different nodes in a cluster. Here we would benefit from the elasticity of a cloud platform like Cloud Foundry.

Spring Cloud Task is a generic abstraction that is designed to manage and make observable processes that run to completion and then terminate. We'll look at it later, but suffice it to say that it works with any Spring Boot-based service that defines an implementation of `CommandLineRunner` or `ApplicationRunner`, both of which are simple callback interfaces that, when perceived on a Spring bean, are given a callback with the application's `String [] args` array from the `main(String args [])` method. Spring Boot automatically configures a `CommandLineRunner` that runs any existing Spring Batch `Job` instances in the Spring application context. So, our jobs are *already* prime candidates to be run and managed as jobs with Spring Cloud Task!

Scheduling

One question that often comes up here is, “How do I schedule these jobs?” If you design your jobs as small singly focused Spring Batch jobs that live in a Spring Boot executable `.jar`, then there’s no reason for simple workloads in a small, off-cloud environment, that you couldn’t use good ‘ol `cron`. Maybe that’s enough. You might even look at commercial schedulers like BMC, Flux Scheduler, or Autosys. If you’d rather have input into how the jobs are scheduled from within your code you might look to the

`ScheduledExecutorService` in the JDK or even move up the abstraction a bit and leverage Spring’s `@Scheduled` annotation, which in turn delegates the `Executor` infrastructure in the JDK. The main flaw with this approach is that there’s no bookeeping down on whether a job was run or not, and there’s no builtin notion of a cluster. What happens if the node running the jobs should die?

If you want something a bit more robust you might checkout Spring’s integration with the [the Quartz Enterprise Job scheduler](#). Quartz runs well in a cluster and should have what you need to get the job done, even if it is a bit heavy. Another approach is to spin up a leader node that sends messagess to other nodes in a cluster when they need to perform some work. The leader node would need to be stateful, or you risk running the same work twice. Spring Cloud Cluster is a framework to support uses cases around leadership election and distributed locks. It makes it easy to transactionally rotate one node out of leadership and another one in. It provides implementations that work with Apache Zookeeper, Hazelcast and Redis. Such a leader node would farm work to other nodes in a cluster, on a schedule. The communication between nodes could be something as simple as messaging. We’ll explore messaging a little later in this chapter.

Cloud Foundry also includes a scheduler as part of the rewritten engine, Diego, that powers it.

There are many answers here. I wouldn’t worry too much about this requirement because, as we’ll see, it’s much easier to think of the world in

terms of events and messaging.

Isolating Failures and Graceful Degradation

We see that Spring Batch can *retry* the processing on a given record if something should go wrong. This is particularly useful in a distributed system where resources may, or may not, be available. It does this using a library called Spring Retry. Spring Retry was extracted from Spring Batch and is now usable outside of Spring Batch. You can use Spring Retry to call downstream services that may or may not be available. Let's look at a simple REST client that calls another service. If the call succeeds, we'll return the service response, however if the call fails it'll throw an exception which Spring Retry will route to a handler method annotated with the `@Recover` method. The recovery method returns the same response as the recoverable method. In our example, it'll return the string `OHAI`. It'll do this for as many attempts as you permit it. By default, it'll call three times. It'll backoff for increasingly longer periods before it exhausts its retries, failing. Use the `@EnableRetry` annotation on a configuration class to activate Spring Retry.

Example 7-2. Spring Retry helps us retry, with increasing backoff periods between attempts, to invoke potentially flaky downstream services.

CODE TO COME ① ②

①

this is the method that may fail, and if it does it'll throw an exception. If we wanted to retry and recover on a specific type of failure we could specify a specific type of `Exception`

②

you may specify as many recovery methods as you like, typed by the `Exception` that they recover from.

Spring Retry might be just what you need for quick and easy recovery and

smart backoff heuristics. Sometimes, however, services may need more time to recover and the deluge of requests and retries can negatively impact their ability to come online. We can use a *circuit-breaker* to statefully gate requests based on whether previous requests have succeeded, or to fallback to a recovery method as we did with Spring Retry.

A circuit breaker achieves some of the same things as Spring Retry. It's a component that, when there's a risk of traffic overwhelming the system, gates the traffic and diverts it to a recovery method immediately. This is different than Spring Retry in that Spring Retry doesn't divert traffic proactively. We can use the Spring Cloud integration for the Netflix Hystrix circuit breaker to achieve a slightly more sophisticated effect as we did with Spring Retry.

Example 7-3. this circuit breaker gives our downstream service time to breathe

CODE TO COME ① ②

①

this is the method that may fail, and if it does it'll throw an exception. If we wanted to retry and recover on a specific type of failure we could specify a specific type of `Exception`

②

you may specify as many recovery methods as you like, typed by the `Exception` that they recover from.

The circuit breaker is said to be either open or closed. If it's closed, then the circuit breaker will attempt to invoke the happy path and then, on exception, call the fallback. If the circuit is open, the circuit breaker will forward traffic directly to the recovery method. Eventually, however, the circuit will close again and attempt to reintroduce traffic. If it fails again, it'll go back to open. The circuit breaker also emits a server-sent event stream that we can monitor using the Hystrix Dashboard and Spring Cloud Turbine. For more on that, check [consult the discussion on observable applications, later on](#).

Right now the distinction between Spring Retry is a little thin. I tend to go with Spring Retry unless I need some of the things that only Hystrix can

offer, like the built-in support for observability. It is our hope that at some point Spring Retry will cover some of the more nuanced use cases in the Hystrix circuit breaker.

If you're using a platform like Cloud Foundry, then there's no reason a downstream service need be down for long. Cloud Foundry has a heartbeat mechanism that will automatically restart a downed service, and it can even be made to scale a service up automatically as demand spikes. Spring Retry and the circuit breaker make it easy for the client experience to degrade gracefully in the inevitable case of a service failure. Netflix and other high performing organizations will build in graceful degradation. As an example, you might imagine calls to a hypothetical search engine service failing. It would impact the user experience if the client were shown a stacktrace! Instead, technologies like Spring Retry and the circuit breaker make it simple to do *something* for the client, albeit perhaps not what the client was expecting.

In these examples, our recovery method was an upfront concern; we *had* to think about the failure case and dealing with the compensatory transaction required should there be a failure. This is a virtue. Assuming that errors will happen and then confronting the possibility upfront promotes more robust systems.

We've looked at how to build resilience in the case of a service failure for which we have no other immediate recourse. This sort of things works well if we care about immediate side-effects. Later, we'll look at messaging as a way of guaranteeing that *eventually* state will be consistent between services, even if a service is down at the moment.

Task Management

Spring Boot knows what to do with our `Job`; when the application starts up Spring Boot runs all `CommandLineRunner` instances, including the one provided by Spring Boot's Spring Batch auto-configuration. From the outside looking in, however, there is no logical way to know how to *run* this job. We don't know, necessarily, that it describes a workload that will terminate and produce an exit status. We don't have common infrastructure that captures the start and end time for a task. There's no infrastructure to support us if an exception occurs. Spring Batch surfaces these concepts, but how do we deal with things that aren't Spring Batch `Job` instances, like `CommandLineRunner` or `ApplicationRunner` instances? Spring Cloud Task helps us here. Spring Cloud Task provides a way of identifying, executing and interrogating, well, tasks! Let's look at a simple example. Start a new Spring Boot project with nothing in it and then add `org.springframework.cloud:spring-cloud-task-core`.

Example 7-4.

```
package task;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.cloud.task.repository.TaskExplorer;
import org.springframework.context.annotation.Bean;
import org.springframework.data.domain.PageRequest;

@EnableTask
❶ @SpringBootApplication
public class HelloTask {

    private Log log = LogFactory.getLog(getClass());

    @Bean
```

```
CommandLineRunner runAndExplore(TaskExplorer taskExplorer
    return args -> ❷
        taskExplorer.findAll(new PageRequest(0, 1)).forEach(
            taskExecution -> log.info(taskExe
    }

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

❶

activate Spring Cloud Task

❷

inject the Spring Cloud Task `TaskExplorer` to inspect the current running task's execution

You gain similar support for Spring Batch workloads with Spring Cloud Task's Batch integration (`org.springframework.cloud:spring-cloud-task-batch`). Spring Cloud Task gives us a uniform way to talk about, manage and introspect workloads that terminate. We'll revisit Spring Cloud Task in our discussion of Spring Cloud Data Flow, later.

Process-Centric Integration with Workflow

While Spring Batch gives us rudimentary workflow capabilities, its purpose is to support processing of large datasets. In this section we'll look at workflow. Workflow is the practice of explicitly modeling the progression of work through a system of autonomous (and human!) agents. Workflow systems define a state machine and constructs for modeling the progression of that state towards a goal. Workflow systems are designed to work closely with the business who may have their own designs on a given business process. Workflow overlaps a lot with the ideas of business process management and business process modeling (both, confusingly, abbreviated as BPM).

Workflow systems support ease of modeling processes. Typically, workflow systems provide design-time tools that facilitate visual modeling. The main goal of this is to arrive at an artifact that minimally specifies the process for both business and technologists. A process model is not directly executable code, but instead a series of steps. It is up to developers to provide appropriate behavior for the various states. Workflow systems typically provide a way to store and query the state of various processes. Like Spring Batch, workflow systems also provide a built-in mechanism to design compensatory behavior in the case of failures.

From a design perspective, workflow systems help keep your services and entities stateless and free of what would otherwise be irrelevant process state. We've seen many systems where an entity's progression through fleeting, one time processes was represented as booleans (`is_enrolled`, `is_fulfilled`, etc.) on the entity and in the database themselves. These flags muddy the design of the entity for very little gain.

Workflow systems map roles to sequences of steps. These roles correspond more or less to swimlines in UML. A workflow engine, like a swimlane, may describe *human* tasklists as well as autonomous activities.

Is workflow for everybody? Decidedly not. We've seen that workflow is most useful for organizations that have complex processes, or are constrained by regulation and policy and need a way to interrogate process state. It's optimal for collaborative processes where humans and services are used to drive towards a larger business requirement (imagine loan approval, legal compliance, insurance reviews, document revision, document publishing, etc).

It simplifies design to free your business logic from the strata of state required to support auditing and reporting of processes. We look at it in this section because it, like batch processing, provides a meaningful way to address failure in a complex, multi-agent and multi-node process. It is possible to model a saga execution coordinator on top of a workflow engine, although a workflow engine is not itself a saga execution coordinator. We get a lot of the implied benefits, though, if used correctly. We'll see later that workflows also lend themselves to horizontal scale on a cloud -environment through messaging infrastructure.

Let's walk through a simple exercise. Suppose you have a signup process for a new user. The business cares about the progression of new signups as a metric. Conceptually, the signup process is a simple affair: the user submits a new signup payload through a form, which then must be validated and - if there are errors - fixed. Once the form is correct and accepted, a confirmation email must be sent. The user has to click on the confirmation email and trigger a well-known endpoint. The user may do this in a minute or in two weeks. Either way, the long-lived transaction is still valid. We could make the process even more sophisticated and specify timeouts and escalations within the definition of our workflow. For now, however, this is an interesting enough example that involves both autonomous and human work towards the goal of enrolling a new customer.

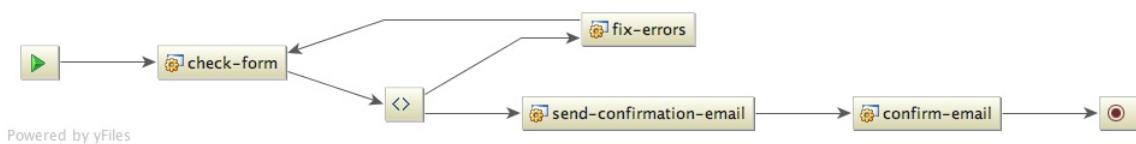


Figure 7-2. a signup process modeled using BPMN 2.0 as viewed from IntelliJ IDEA's yFiles-powered BPMN 2.0 preview.

Enter Alfresco's Activiti project. Activiti is a *business process engine*. It supports process definitions in an XML standard called BPMN 2.0. BPMN 2.0 enjoys robust support across multiple vendors' tooling and IDEs. You define a BPMN business process using tools from, say, WebMethods and can then execute it in Activiti (or vice-versa!). Activiti also provides a modeling environment, is easy to deploy standalone or use in a cloud-based services, and is Apache 2 licensed. It also provides a Spring Boot auto-configuration.

In Activiti, a `Process` defines a process itself. A `ProcessInstance` is a single execution of a given process. It's executional, not definitional. It's made unique by process variables, which parameterize the process' execution.

The Spring Boot auto-configuration expects any BPMN 2.0 documents to be in the `src/main/resources/processes` directory of an application, by default. Let's take a look at the markup for the BPMN 2.0 `signup` process.

Example 7-5. the `signup.bpmn20.xml` business process definition

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:activiti="http://activiti.org/bpmn"
    typeLanguage="http://www.w3.org/2001/XMLSchema"
    expressionLanguage="http://www.w3.org/1999/XPath"
    targetNamespace="http://www.activiti.org/bpmn2.0">

    <process name="signup" id="signup">
        ①
        <startEvent id="start"/>

        ②
        <sequenceFlow sourceRef="start" targetRef="check-form"/>

        ③
        <serviceTask id="check-form" name="check-form" activiti:>
            <sequenceFlow sourceRef="check-form" targetRef="form-comp

        ④
        <exclusiveGateway id="form-completed-decision-gateway"/>
```

```

<sequenceFlow name="formOK" id="formOK" sourceRef="form-c
    <conditionExpression xsi:type="tFormalExpression">>${f
</sequenceFlow>

<sequenceFlow id="formNotOK" name="formNotOK" sourceRef=
    <conditionExpression xsi:type="tFormalExpression">>${
</sequenceFlow>

⑤
<userTask name="fix-errors" id="fix-errors">
    <humanPerformer>
        <resourceAssignmentExpression>
            <formalExpression>customer</formalExpression>
        </resourceAssignmentExpression>
    </humanPerformer>
</userTask>

<sequenceFlow sourceRef="fix-errors" targetRef="check-for
    <conditionExpression xsi:type="tFormalExpression">>${f
</sequenceFlow>

⑥
<serviceTask id="send-confirmation-email" name="send-conf
    <resourceAssignmentExpression>
        <formalExpression>customer</formalExpression>
    </resourceAssignmentExpression>
</serviceTask>

<sequenceFlow sourceRef="send-confirmation-email" targetR
    <conditionExpression xsi:type="tFormalExpression">>${f
</sequenceFlow>

⑦
<userTask name="confirm-email" id="confirm-email">
    <humanPerformer>
        <resourceAssignmentExpression>
            <formalExpression>customer</formalExpression>
        </resourceAssignmentExpression>
    </humanPerformer>
</userTask>

<sequenceFlow sourceRef="confirm-email" targetRef="end"/>
    <endEvent id="end"/>
</process>
</definitions>

```

①

the first state is the `startEvent`. All processes have a defined `start` and `end`.

②

the `sequenceFlow` elements serve as guidance to the engine about where to go next. They're logical, and are represented as lines in the workflow model itself.

③

a `serviceTask` is a state in the process. Our BPMN 2.0 definition uses the Activiti-specific `activiti:expression` attribute to delegate handling to method, `execute(ActivityExecution)` on a Spring bean (called `checkForm`). Spring Boot's auto-configuration activates this behavior.

④

After the form is submitted and checked, the `checkForm` method contributes a boolean *process variable* called `formOK`, whose value is used to drive a decision downstream. A process variable is context that's visible to participants in a given process. Process variables can be whatever you'd like, though we tend to keep them as trivial as possible to be used later to act as claim-checks for real resources elsewhere. This process expects one input process variable, `customerId`.

⑤

if the form is invalid, then work flows to the `fix-errors` user task. The task is contributed to a worklist and assigned to a human. This tasklist is supported through a Task API in Activiti. It's trivial to query the tasklist and start and complete tasks. The tasklist is meant to model work that a human being must perform. When the process reaches this state, it pauses, waiting to be explicitly completed by a human at some later state. Work flows from the `fix-errors` task back to the `checkForm` state. If the form is now fixed, work proceeds to the

⑥

if the form is valid, then work flows to the `send-confirmation-email` service task. This simply delegates to another Spring bean to do its work, perhaps using SendGrid.

⑦

It's not hard to imagine what has to happen here: the email should contain some sort of link that, when clicked, triggers an HTTP endpoint that then completes the outstanding task and moves the process to completion.

This process is trivial but it provides a clean representation of the moving pieces in the system. We can see that we will need *something* to take the inputs from the user, resulting in a customer record in the database and a valid `customerId` that we can use to retrieve that record in downstream components. The customer record may be in an invalid state (the email might be invalid), and may well need to be revisited by the user. Once things are in working order, state moves to the step where we send a confirmation email that, once received and confirmed, transitions the process to the terminal state.

Let's look at a simple REST API that drives this process along. For brevity, we've not extrapolated out an iPhone client or an HTML5 client, but it's certainly a natural next step. The REST API is kept as simple as possible for illustration. A *very* logical next step might be to use hypermedia to drive the clients interactions with the REST API from one state transition to another. For more on this possibility [checkout the REST chapter](#) and the discussion of hypermedia and HATEOAS.

Example 7-6. the `SignupRestController` that drives the process

```
package com.example;

import org.activiti.engine.RuntimeService;
import org.activiti.engine.TaskService;
import org.activiti.engine.task.TaskInfo;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.web.bind.annotation.*;

import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
```

```

@RestController
@RequestMapping("/customers")
class SignupRestController {

    public static final String CUSTOMER_ID_PV_KEY = "customer

        private final RuntimeService runtimeService;
        private final TaskService taskService;
        private final CustomerRepository customerRepository;
        private Log log = LogFactory.getLog(getClass()));

①
    @Autowired
    public SignupRestController(RuntimeService runtimeService
                                TaskService taskService, CustomerReposito
        this.runtimeService = runtimeService;
        this.taskService = taskService;
        this.customerRepository = repository;
    }

②
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<?> startProcess(@RequestBody Custom
        Assert.notNull(customer);
        Customer save = this.customerRepository.save(new
            .getFirstName(), customer.getLastName());

        String processInstanceId = this.runtimeService
            .startProcessInstanceByKey(
                "signup",
                Collections.singl
                    L
                this.log.info("started sign-up. the processInstan
                    + processInstanceId);

        return ResponseEntity.ok(save.getId());
    }

③
    @RequestMapping(method = RequestMethod.GET, value = "/{cu
    public List<String> readErrors(@PathVariable String custo
        return this.taskService.createTaskQuery().active(
            .taskName("fix-errors").includePr
            .processVariableValueEquals(CUSTO
                .list().stream().map(TaskInfo::ge
                .collect(Collectors.toList()));
    }
}

```

④

```
@RequestMapping(method = RequestMethod.POST, value = "/{c
public void fixErrors(@PathVariable String customerId,
                      @PathVariable String taskId, @RequestBody

Customer customer = this.customerRepository.findO
                      .parseLong(customerId));
customer.setEmail(fixedCustomer.getEmail());
customer.setFirstName(fixedCustomer.getFirstName(
customer.setLastName(fixedCustomer.getLastName())
this.customerRepository.save(customer);

this.taskService
                      .createTaskQuery()
                      .active()
                      .taskId(taskId)
                      .includeProcessVariables()
                      .processVariableValueEquals(CUSTO
                      .list()
                      .forEach(
                          t -> {
                              log.info(
                                  taskServ
                                  );
                          }
                      );
}
```

⑤

```
@RequestMapping(method = RequestMethod.POST, value = "/{c
public void confirm(@PathVariable String customerId) {
    this.taskService.createTaskQuery().active().taskN
                      .includeProcessVariables()
                      .processVariableValueEquals(CUSTO
                      .list().forEach(t -> {
                          log.info(t.toString());
                          taskService.complete(t.ge
                      });
    this.log.info("confirmed email receipt for " + cu
}
}
```

①

the controller will work with a simple Spring Data JPA repository as well as two services that are autoconfigured by the Activiti Spring Boot

support. The `RuntimeService` lets us interrogate the process engine about running processes. The `TaskService` lets us interrogate the process engine about running human tasks and tasklists.

②

the first endpoint accepts a new `customer` record and persists it. The newly minted customer is fed as an input process variable into a new process, where the customer's `customerId` is specified as a process variable. Here, the process flows to the `check-form` serviceTask which will nominally check that the input email is valid. If the email is valid, then a confirmational email is sent. Otherwise, the customer will need to address any errors in the input data.

③

if there are any errors, we want the user's UI experience to arrest forward progress, so the process queues up a user task. This endpoint queries for any outstanding tasks for this particular user and then returns a collection of outstanding tasks IDs. Ideally, this might also return validation information that can drive the UX for the human using some sort of interactive experience to enroll.

④

Once the errors are addressed client-side, the updated entity is persisted in the database and the outstanding task is marked as complete. At this point the process flows the `send-confirmation-email` state which will send an email that includes a confirmation link that the user will have to click to confirm the enrollment.

⑤

The final step, then, is a REST endpoint that queries any outstanding email confirmation tasks for the given customer.

We start this process with a potentially invalid entity, the customer, whose state we need to ensure is correct before we can proceed. We model this process to support iteration on the entity, backtracking to the appropriate step for so long as there's invalid state.

Let's complete our tour by examining the beans that power the two serviceTask elements, check-form, and send-confirmation-email, in the process.

Both the CheckForm and SendConfirmationEmail beans are uninteresting. They're simple Spring beans. CheckForm implements a trivial validation that ensures the first name and last name are not null and then uses the Mashape email validation REST API introduced in the discussion on Spring Batch to validate that the email is of a valid form. The CheckForm bean is used to validate the state of a given Customer record and then contributes a process variable to the running ProcessInstance, a boolean called formOK, that then drives a decision as to whether to continue with the process or force the user to try again.

Example 7-7. the CheckForm bean that validates the customer's state

```
package com.example;

import org.activiti.engine.RuntimeService;
import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collections;
import java.util.Map;

import static org.apache.commons.lang3.StringUtils.isEmpty;

@Service
class CheckForm {

    private final RuntimeService runtimeService; ①
    private final CustomerRepository customerRepository;
    private final EmailValidationService emailValidationService;

    @Autowired
    public CheckForm(EmailValidationService emailValidationService,
                    RuntimeService runtimeService, CustomerRepository
        this.runtimeService = runtimeService;
        this.customerRepository = customerRepository;
        this.emailValidationService = emailValidationService
    }
}
```

②

```
public void execute(ActivityExecution e) throws Exception
    Long customerId = Long.parseLong(e.getVariable("c
        String.class));
    Map<String, Object> vars = Collections.singletonM
        validated(this.customerRepository
    this.runtimeService.setVariables(e.getId(), vars)
}
```

private boolean validated(Customer customer) {
 return !isEmpty(customer.getFirstName())
 && !isEmpty(customer.getLastName()
 && this.emailValidationService
 .isEmailValid(cus
}

}

①

inject the Activiti RuntimeService

②

the ActivityExecution is a context object. You can use it to access process variables, the process engine itself, and other interesting services.

③

then use it to articulate the outcome of the test

The SendConfirmationEmail is of the same basic form.

Event Driven Architectures with Spring Integration

We've looked thus far at processing that we initiate, on our schedule. The world doesn't run on our schedule. We are surrounded by events that drive everything we do. The logical windowing of data is valuable, but some data just can't be treated in terms of windows. Some data is continuous and connected to events in the real world. In this section we'll look at how to work with data driven by events.

When we think of events, most of us probably think of messaging technologies. Messaging technologies, like JMS, RabbitMQ, Apache Kafka, Tibco Rendezvous and IBM MQSeries. These technologies let us connect different autonomous clients through messages sent to centralized middleware, in much the same way that e-mail lets humans connect to each other. The message broker stores delivered messages until such time as the client can consume and respond to it (in much the same way as an e-mail inbox does).

Most of these technologies have an API, and a usable client that we can use. Spring has always had good low-level support for messaging technologies; you'll find sophisticated low-level support for the JMS API, AMQP (and brokers like RabbitMQ), Redis and Apache Geode.

There are many types of events in the world, of course. Receiving an email is one. Receiving a tweet another. A new file in a directory? That's an event too. An XMPP-powered chat message? Also an event. Your MQTT-powered microwave sending a status update? That's also an event.

It's all a bit overwhelming! If you're looking at the landscape of different event sources out there then you're (hopefully) seeing a lot of opportunity *and* of complexity. Some of the complexity comes from the act of integration itself. How do you build a system that depends on events from these various systems? One might address integration in terms of point-to-point

connections between the various event sources. This will result eventually in *spaghetti architecture*, and it's a mathematically poor idea, too, as every integration point needs a connection with every other one. It's a binomial coefficient: $n(n-1) / 2$. Thus, for six services you'd need 15 different point-to-point connections!

Instead, let's take a more structured approach to integration with Spring Integration. At the heart of Spring Integration are the Spring framework `MessageChannel` and `Message<T>` types. A `Message<T>` object has a payload and a set of headers that provide metadata about the message payload itself. A `MessageChannel` is like a `java.util.Queue`. `Message<T>` objects flow through `MessageChannel` instances.

Spring Integration supports the integration of services and data across multiple otherwise incompatible systems. Conceptually, composing an integration flow is similar to composing a pipes-and-filters flow on a UNIX OS with `stdin` and `stdout`:

Example 7-8. an example of using the pipes-and-filters model to connect otherwise singly focused command line utilities

```
cat input.txt | grep ERROR | wc -l > output.txt
```

Here, we take data from a source (the file `input.txt`), pipe it to the `grep` command to filter the results and keep only the lines that contain the token `ERROR`, and then pipe it to the `wc` utility which we use to count how many lines there are. Finally, the final count is written to an output file, `output.txt`. These components - `cat`, `grep`, and `wc` - know nothing of each other. They were not designed with each other in mind. Instead, they know only how to read from `stdin` and write to `stdout`. This normalization of data makes it very easy to compose complex solutions from simple atoms. In the example, the act of `cat`ing` a file turns data into data that any process aware of `stdin can read. It *adapts* the inbound data into the normalized format, lines of strings. At the end, the redirect (`>`) operator turns the normalized data, lines of strings, into data on the file system. It *adapts* it. The pipe (`|`) character is used to signal that the output of one component should flow to the input of another.

A Spring Integration flow works the same way: data is normalized into `Message<T>` instances. Each `Message<T>` has a payload and headers - metadata about the payload in a `Map<K, V>` - that are the input and output of different messaging components. These messaging components are typically provided by Spring Integration, but it's easy to write and use your own. There are all manner of messaging components supporting all of the [the Enterprise Application Integration patterns]

(<http://www.enterpriseintegrationpatterns.com/>) (filters, routers, transformers, adapters, gateways, etc.). The Spring framework

`MessageChannel` is a named conduit through which `Message<T>`'s flow between messaging components. They're pipes and, by default, they work sort of like a ``java.util.Queue``. Data in, data out.

Messaging Endpoints

These `MessageChannel` objects are connected through messaging endpoints, Java objects that do different things with the messages. Spring Integration will do the right thing when you give it a `Message<T>` or just a `T` for the various components. Spring Integration provides a component model that you might use, or a Java DSL. Each messaging endpoint in a Spring Integraton flow may produce an output value which is then sent to whatever is downstream, or `null`, which terminates processing.

Inbound gateways take incoming requests from external systems, process them as '`Message<T>`'s, and send a reply. Outbound gateways take '`Message<T>`'s, forward them to an external system, and await the response from that system. They support request and reply interactions.

An **inbound adapter** is a component that takes messages from the outside world and then turns them into a Spring `Message<T>`. An **outbound adapter** does the same thing, in reverse; it takes a Spring `Message<T>` and delivers it as the message type expected by the downstream system. Spring Integration ships with a proliferation of different adapters for technologies and protocols including MQTT, Twitter, email, (S)FTP(S), XMPP, TCP/UDP, etc.

There are two types of inbound adapters: either a polling adapter or an event drive adapter. The inbound polling adapter is configured to automatically pull a certain interval or rate an upstream message source.

A **gateway** is a component that handles both requests and replies. For example, an inbound gateway would take a message from the outside world, deliver it to Spring Integration, and then deliver a reply message back to the outside world. A typical HTTP flow might look like this. An outbound gateway would take a spring integration message and deliver it to the outside world, then take the reply and delivered back in the spring integration. You might see this when using the RabbitMQ broker and you've specified a reply destination.

A **filter** is a component that takes incoming messages and then applies some

sort of condition to determine whether the message should proceed. Think of a filter like an `if (...) test` in Java.

A **router** takes an incoming message and then applies some sort test to determine where downstream to send that message downstream. Think of a router as a switch statement.

A **transformer** takes a message and does something with it, optionally enriching or changing it, and then it sends the message out.

A **splitter** takes a message and then, based on some property of the message, divides it into multiple smaller messages that are then forwarded downstream. You might for example have an incoming message for an order and then forward a message for each line item in that order to some sort of fulfillment flow.

An **aggregator** takes multiple messages, correlated through some unique property and then synthesizes a message that is sent downstream.

The integration flow is aware of the system, but the involved components used in the integration don't need to be. This makes it easy to compose complex solutions from small, otherwise silo'd services. The act of building a Spring integration flow is rewarding in of itself. It forces a logical decomposition of services; they must be able to communicate in terms of messages that contain payloads. The schema of the payload is the contract. This property is very useful in a distributed system.

From Simple Components, Complex Systems

Spring Integration supports *event-driven architectures* because it can help detect and then respond to events in the external world. For example, you can use Spring Integration to poll a filesystem every 10 seconds and publish a `Message<T>` whenever a new file appears. You can use Spring Integration to act as a listener to messages delivered to a Apache Kafka topic. The adapter handles responding to the external event and frees you from worrying too much about originating the message and lets you focus on handling the message once it arrives. It's the integration equivalent of dependency injection!

Dependency injection leaves component code free of worries about resource initialization and acquisition and leaves it free to focus on writing code with those dependencies. Where did the `javax.sql.DataSource` field come from? Who cares! Spring wired it in, and it may have gotten it from a Mock in a test, from JNDI in a classic application server, or from a configured Spring Boot bean. Component code remains ignorant of those details. You can explain dependency injection with the “Hollywood principal:” “don’t call me, I’ll call you!” Dependant objects are provided to an object, instead of the object having to initialize or lookup the resource. This same principle applies to Spring Integration: code is written in such a way that its ignorant of where messages are coming from. It simplifies development considerably.

So, let’s start with something simple. Let’s look at a simple example that responds to new files appearing in a directory, logs the observed file, and then dynamically routes the payload to one of two possible flows based on a simple test, using a router.

We’ll use the Spring Integration Java DSL. The Java DSL works very nicely with lambdas in Java 8. Each `IntegrationFlow` chains components together implicitly. We can make explicit this chaining by providing connecting `MessageChannel` references.

Example 7-9. an example of using the pipes-and-filters model to connect otherwise singly focused command line utilities

```
package eda;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannel;
import org.springframework.integration.dsl.file.Files;
import org.springframework.messaging.MessageChannel;

import java.io.File;

@Configuration
public class IntegrationConfiguration {

    private Log log = LogFactory.getLog(getClass());

    @Bean
    IntegrationFlow etlFlow(
        @Value("${input-directory:${HOME}/Desktop")
        // @formatter:off
        return IntegrationFlows
            .from(Files.inboundAdapter(direct
                consumer -> consu
            .handle(File.class, (file, header
                log.info("we noticed a ne
                return file;
            })
            .routeToRecipients(
                spec -> spec
                    .get();
        // @formatter:on
    }
}
```

```

        boolean hasExt(Object f, String ext) {
            File file = File.class.cast(f);
            return file.getName().toLowerCase().endsWith(ext);
        }

❸
@Bean
MessageChannel txt() {
    return MessageChannels.direct().get();
}

❹
@Bean
MessageChannel csv() {
    return MessageChannels.direct().get();
}

❺
@Bean
IntegrationFlow txtFlow() {
    return IntegrationFlows.from(txt()).handle(File.c
        log.info("file is .txt!");
        return null;
    }).get();
}

❻
@Bean
IntegrationFlow csvFlow() {
    return IntegrationFlows.from(csv()).handle(File.c
        log.info("file is .csv!");
        return null;
    }).get();
}
}

```

❻

configure a Spring Integration inbound File adapter. We also want to configure how the adapter consumes incoming messages and at what millisecond rate the poller that sweeps the directory should scan.

❼

this method announces that we've received a file and then forward the

payload onward

③

route the request to one of two possible integration flows, derived from the extension of the file, through well-known `MessageChannel` instances

④

the channel through which all files with the `.txt` extension will travel

⑤

the channel through which all files with the `.csv` extension will travel

⑥

the `IntegrationFlow` to handle files with `.txt`

⑦

the `IntegrationFlow` to handle files with `.csv`

The channel is a logical decoupling; it doesn't matter what's on either the other end of the channel, so long as we have a pointer to the channel. Today the consumer that comes from a channel might be a simple logging `MessageHandler<T>`, as in this example, but tomorrow it might instead be a component that writes a message to Apache Kafka. We can also begin a flow from the moment it arrives in a channel. How, exactly, it arrives in the channel is irrelevant. We could accept requests from a REST API, or adapt messages coming in from Apache Kafka, or monitor a directory. It doesn't matter so long as we somehow adapt the incoming message into a `java.io.File` and submit it to the right channel.

Let's revisit our existing batch solution and connect it to events. Our batch job works by processing a file. We must explicitly launch the job or schedule the job. We could schedule a `cron` job for midnight and process everything, but it'd be more efficient to avoid idle time and launch the batch job anytime a new file to be processed appears in a directory. Spring Integration provides a file inbound adapter that lets us achieve this. We'll listen for messages coming from the file inbound adapter, transform it into a message whose

payload is a JobLaunchRequest which in turn contains JobParametes and a Job to launch. Finally, the JobLaunchRequest is forwarded to a JobLaunchingGateway which then returns as its output a JobExecution object that we inspect to decide where to route execution. If a job completes normally, we'll move the input file to a directory of completed jobs, or we'll move the file to an error directory.

Example 7-10. launching a Spring Batch job in response to an event

```
package edabatch;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.integration.launch.JobLaunchRequ
import org.springframework.batch.integration.launch.JobLaunchingG
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannel
import org.springframework.integration.dsl.file.Files;
import org.springframework.integration.file.FileHeaders;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.messaging.MessageChannel;
import org.springframework.util.Assert;
import org.springframework.util.StreamUtils;

import java.io.*;
import java.util.List;

@Configuration
public class IntegrationConfiguration {

    private Log log = LogFactory.getLog(getClass());

    @Bean
    MessageChannel invalid() {
        return MessageChannels.direct().get();
    }

    @Bean
```

```

MessageChannel completed() {
    return MessageChannels.direct().get();
}

❶
@Bean
IntegrationFlow etlFlow(
    @Value("${input-directory:${HOME}/Desktop")
    JobLauncher launcher, Job job) {
    // @formatter:off
    return IntegrationFlows
        .from(Files.inboundAdapter(direct
            consumer -> consu
            .
            .handle(File.class,
                (file, headers) -

❷
JobParame
            .
            .
            .
            return Me

            }
        }

❸
            .handle(new JobLaunchingGateway(l
            .routeToRecipients(
                spec -> spec.recipient
                    m
                    .
                    .
                    .
                    .get();
            // @formatter:on
        }

❹
private boolean jobFinished(Object payload) {
    return JobExecution.class.cast(payload).getExitSt

```

```

        .equals(ExitStatus.COMPLETED);
    }

@Bean
IntegrationFlow finishedJobsFlow(
    @Value("${completed-directory:${HOME}/Desktop}")
    JdbcTemplate template) {
    // @formatter:off
    return IntegrationFlows
        .from(completed())
        .handle(JobExecution.class,
            (je, headers) ->
                String og
                    File file
                    mv(file,
                        List<Cont

    contacts.
    return nu
}) .get();
    // @formatter:on
}

@Bean
IntegrationFlow invalidFileFlow(
    @Value("${error-directory:${HOME}/Desktop")
    // @formatter:off
    return IntegrationFlows
        .from(this.invalid())
        .handle(JobExecution.class,
            (je, headers) ->
                String og
                    File file
                    mv(file,
                        return nu
}) .get();
    // @formatter:on
}

private void mv(File in, File out) {
    // @formatter:off

```

```

        try {
            Assert.isTrue(out.exists() || out.mkdirs(
                try (InputStream inStream = new BufferedI
                    new FileInputStream(in));
                    OutputStream outStream =
                    new FileOutputStream(out);
                    StreamUtils.copy(inStream, outStr
                }
            ) catch (Exception e) {
                throw new RuntimeException(e);
            }
            // @formatter:on
        }
    }
}

```

①

this `IntegrationFlow` starts off the same as in the previous example

②

setup some `JobParameters` for our Spring Batch job using the `JobParametersBuilder`

③

forward the job and associated parameters to a `JobLaunchingGateway`

④

test to see if the job exited normally by examining the `JobExecution`

Message Brokers, Bridges, the Competing Consumer Pattern and Event-Sourcing

The last example was fairly straightforward. It's a simple flow that works in terms of messages (events). We're able to begin processing as soon as possible without overwhelming the system. As our integration is built in terms of channels, and every component could consume or produce messages with a channel, it would be trivial to introduce a message broker in between the inbound file adapter and the node that actually launches the Spring Batch job. This would help us scale out the work across a cluster, like one powered by Cloud Foundry.

That said, you're not likely going to be integrating file systems in a cloud-native architecture. You will, however, probably have other, non-transactional, event sources (message producers) and sinks (message consumers) with which you'll want to integrate.

We could use the saga pattern and design compensatory transactions for every service with which we integrate and the possible failure conditions, but we might be able to get away with something simpler if we use a message broker. Message brokers are conceptually very simple: as messages are delivered to the broker, they're stored and delivered to connected consumers. If there are no connected consumers, then the broker will store the messages and redeliver them upon connection of a consumer.

Message brokers typically offer two types of destinations (think of them as mailboxes): publish-subscribe and point-to-point. A public-subscribe destination delivers one message to all connected consumers. It's a bit like broadcasting a message over a megaphone to a full room.

Publish-subscribe messaging supports event-collaboration, wherein multiple systems keep their own view or perspective of the state of the world. As new

events arrive from different components in the system, with updates to state, each system updates its local view of the system state. Suppose you had a catalog of products. As new entries were added to the product-service, it might publish an event describing the delta. The search-engine service might consume the messages and update its internal Elasticsearch index. The inventory-service might update its own internal state in an RDBMS. The recommendation-service might update its internal state in a Neo4J service. These systems no longer need to *ask* the product-service for anything, the product-service *tells*. If you record every event in a log, you have the ability to do temporal queries, analyzing the state of the system at any time since in the past. If any service should fail, its internal state may be replayed entirely from the log. You can *cherry pick* parts of the state by replaying state up until a point and perhaps skipping over some anomalous event. Yes, this *is* how a version control system works! This approach is called *event-sourcing*, and it's very powerful. Message brokers like Apache Kafka have the ability to selectively consume messages given an offset. You could, of course, re-read *all* the messages which would give you, in effect, a poor man's event-source. There are also some purpose-built event-source technologies like [Chris Richardson's Eventuate platform](#).

A point-to-point destination delivers one message to one consumer, even if there are multiple connected consumers. This is a bit like telling one person a secret. If you connect multiple consumers and they all work as fast as they can to drain messages from the point-to-point destination then you get a sort of load-balancing: work gets divided by the number of connected consumers. This approach, called the competing consumers pattern, simplifies load-balancing work across multiple consumers and makes it an ideal way to leverage the elasticity of a cloud computing environment like Cloud Foundry, where horizontal capacity is elastic and (virtually) infinite.

Message brokers also have their own, resource-local notion of a transaction. A producer may deliver a message and then, if necessary, withdraw it, effectively rolling the message back. A consumer may accept delivery of a message, attempt to do something with it, and then acknowledge the delivery or - if something should go wrong - return the message to the broker, effectively rolling back the delivery. *Eventually* both sides we'll agree upon the state. This is different than a distributed transaction in that the message

broker introduces the variable of time, or temporal decoupling. In so doing it simplifies the integration between services. This property makes it easier to reason about state in a distributed system. You can ensure that two otherwise non-transactional resources will - *eventually* agree upon state. In this way, a message broker *bridges* the two otherwise non-transactional resources.

A message broker complicates the architecture by adding another moving part to the system. Message brokers have well-known recipes for disaster recovery, backups, and scale out. An organization will need to know how to do this and then they can reuse that across all services. The alternative is that *every* service be forced to re-invent or, less desirably, go without these qualities. If you're using a platform like Cloud Foundry which already manages the message broker for you, then using a message broker should be a *very* easy decision for it.

Logically, message brokers make a lot of sense as a way by which we can connect different services. Spring Integration provides ample support here in the way of adapters that produce and consume messages from a diverse set of brokers.

Spring Cloud Stream

While Spring Integration is on solid footing to solve the problems of service-to-service communication with message brokers, it might seem a bit too clumsy in the large. We want to support messaging with the same ease as we think about REST-based interactions with Spring. We're not going to connect our services using Twitte, or e-mail. More likely, we'll use RabbitMQ or Apache Kafka or similar message brokers with known interaction modes. We could explicitly configure inbound and outbound RabbitMQ or Apache Kafka adapters, of course. Instead, let's move up the stack a little bit, to simplify our work and remove the cognitive dissonance of configuring inbound and outbound adapters every time we want to work with another service.

Spring integration does a great job of decoupling component in terms of `MessageChannel` objects. A `MessageChannel` is a nice level of indirection. From the perspective of our application logic, a channel represents a logical conduit to some downstream service that will route through a message broker.

In this section, we'll look at Spring Cloud Stream. Spring Cloud Stream sits atop Spring Integration, with the channel at the heart of the inteaction model. It implies conventions and supports easy externalization of configuration. In exchange, it makes the common case of connecting services with a message broker much cleaner and more concise.

Let's have a look at a simple example. We'll build a producer and a consumer. The producer will expose a REST API endpoint that, when invoked, publishes a message into two channels. One for *broadcast*, or publish-subscribe-style messaging, and the other for point-to-point messaging. We'll then standup a consumer to accept these incoming messages.

Spring Cloud Stream makes it easy to define channels that are then connected to messaging technologies. We can use *binder* implementations to, by convention, connect to a broker. In this example, we'll use RabbitMQ, a

popular message broker that speaks the AMQP specification. The binder for Spring Cloud Stream's RabbitMQ support is `org.springframework.cloud:spring-cloud-starter-stream-rabbit`. There are clients and bindings in dozens of languages, and this makes RabbitMQ (and the AMQP specification in general) a fine fit for integration across different languages and platforms. Spring Cloud Stream builds on Spring Integration (which provides the necessary inbound and outbound adapters) and Spring Integration in turn builds on Spring AMQP (which provides the low-level `AmqpOperations`, `RabbitTemplate`, a RabbitMQ `ConnectionFactory` implementation, etc.). Spring Boot autoconfigures a `ConnectionFactory` based on defaults or properties. On a local machine with an unadulterated RabbitMQ instance, this application will work out of the box.

A Stream Producer

The centerpiece of Spring Cloud Stream is a *binding*. A binding defines logical references to other services through `MessageChannel` instances that we'll leave to Spring Cloud Stream to connect for us. From the perspective of our business logic, these downstream or upstream messaging-based services are unknowns on the other side of `MessageChannel` objects. We don't need to worry about how the connection is made, for the moment. Let's define two channels, one for broadcasting a greeting to all consumers and another for sending a greeting point-to-point, once, to whichever consumer happens to receive it first.

Example 7-11. a simple `MessageChannel`-centric greetings producer

```
package stream.producer;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface ProducerChannels {

    ❶
    String DIRECT = "directGreetings";
    String BROADCAST = "broadcastGreetings";

    @Output(DIRECT)
    ❷
    MessageChannel directGreetings();

    @Output(BROADCAST)
    MessageChannel broadcastGreetings();
}
```

❶

by default the name of the stream, which we'll work with in other parts of the system, is based on the `MessageChannel` method itself. It's useful to provide a String constant in the interface so we can reference without any magic strings.

②

Spring Cloud Stream provides two annotations, `@Output` and `@Input`. An `@Output` annotation tells Spring Cloud Stream that messages put into the channel will be sent out (usually, and ultimately, through an outbound channel adapter in Spring Integration).

We need to give Spring Cloud Stream an idea of what to do with data sent into these channels which we can do with some well-placed properties in the environment. Here's what our producer node's `application.properties` looks like.

Example 7-12. a simple `MessageChannel`-centric greetings producer

```
spring.cloud.stream.bindings.broadcastGreetings.destination = gre  
①  
spring.cloud.stream.bindings.directGreetings.destination = greeti  
②  
spring.rabbitmq.addresses=localhost
```

①

in these two lines the sections just after

`spring.cloud.stream.bindings.` and just before `.destination` have to match the name of the Java `MessageChannel`. This is the application's local perspective on the service it's calling. The bit after the `=` sign is the agreed upon rendez-vous point for the producer and the consumer. Both sides need to specify the exact name here. This is the name of the destination in whatever broker we've configured.

②

we're using Spring Boot's auto-configuration to create a `RabbitMQConnectionFactory` which Spring Cloud Stream will depend on.

Let's look now at a simple producer that stands up a REST API that then publishes messages to be observed in the consumer.

Example 7-13. a simple `MessageChannel`-centric greetings producer

```
package stream.producer.channels;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.http.ResponseEntity;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;

@SpringBootApplication
@EnableBinding(ProducerChannels.class)
❶
public class StreamProducer {

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

❷
@RestController
class GreetingProducer {

    private final MessageChannel broadcast, direct;

    @Autowired
    GreetingProducer(ProducerChannels channels) {
        this.broadcast = channels.broadcastGreetings();
        this.direct = channels.directGreetings();
    }

    @RequestMapping("/hi/{name}")
    ResponseEntity<String> hi(@PathVariable String name) {
        String message = "Hello, " + name + "!";
        this.direct.send(MessageBuilder.withPayload("Dire
            .build()));

        this.broadcast.send(MessageBuilder.withPayload("B
            .build());
        return ResponseEntity.ok(message);
    }
}
```

```
    }  
}
```

①

the `@EnableBinding` annotation activates Spring Cloud Stream

②

we inject the hydrated `ProducerChannels` and then dereference the required channels in the constructor so that we can send messages whenever somebody makes an HTTP request at `/hi/{name}`.

③

this is a regular Spring framework channel, so it's simple enough to use the `MessageBuilder` API to create a `Message<T>`.

Style matters, of course, so while we've reduced the cost of working with our downstream services to an interface, a few lines of property declarations and a few lines of messaging-centric channel manipulation, we *could* do better if we used Spring Integration's messaging gateway support. A messaging gateway, as a design pattern, is meant to hide the client from the messaging logic behind a service. From the client perspective, a gateway may seem like a regular object. This can be very convenient. You might define an interface and synchronous service today and then extract it out as a messaging gateway based implementation tomorrow and nobody would be the wiser. Let's revisit our producer and, instead of sending messages directly with Spring Integration, let's send messages using a messaging gateway.

Example 7-14. a messaging gateway producer implementation.

```
package stream.producer.gateway;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.stream.annotation.EnableBinding;  
import org.springframework.http.ResponseEntity;  
import org.springframework.integration.annotation.Gateway;  
import org.springframework.integration.annotation.IntegrationComp
```

```

import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;

@SpringBootApplication
@EnableBinding(ProducerChannels.class)
①
@IntegrationComponentScan
②
public class StreamProducer {

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

③
@MessagingGateway
interface GreetingGateway {

    @Gateway(requestChannel = ProducerChannels.BROADCAST)
    void broadcastGreet(String msg);

    @Gateway(requestChannel = ProducerChannels.DIRECT)
    void directGreet(String msg);
}

@RestController
class GreetingProducer {

    private final GreetingGateway gateway;

    ④
    @Autowired
    GreetingProducer(GreetingGateway gateway) {
        this.gateway = gateway;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
    ResponseEntity<String> hi(@PathVariable String name) {
        String message = "Hello, " + name + "!";
        this.gateway.directGreet("Direct: " + message);
        this.gateway.broadcastGreet("Broadcast: " + message);
        return ResponseEntity.ok(message);
    }
}

```

```
    }  
}
```

①

the `@EnableBinding` annotation activates Spring Cloud Stream, as before

②

many Spring frameworks register stereotype annotations for custom components, but Spring Integration can also turn interface definitions into beans, so we need a custom annotation to activate Spring Integration's component-scanning to find our declarative, interface-based messaging gateway

③

the `@MessagingGateway` is one of the many messaging endpoints supported by Spring Integration (as an alternative to the Java DSL, which we've used thus far). Each method in the gateway is annotated with `@Gateway` where we specify on which message channel the arguments to the method should go. In this case, it'll be as though we sent the message onto a channel and called `.send(Message<String>)` with the argument as a payload.

④

the `GreetingGateway` is just a regular bean, as far as the rest of our business logic is concerned.

A Stream Consumer

On the other side, we want to accept delivery of messages and log them out. We'll create channels using an interface. It's worth reiterating: these channel names *don't* have to line up with the names on the producer side; only the names of the destinations in the brokers do.

Example 7-15. the channels for incoming greetings

```
package stream.consumer;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface ConsumerChannels {

    String DIRECTED = "directed";
    String BROADCASTS = "broadcasts";

    ①
    @Input(DIRECTED)
    SubscribableChannel directed();

    @Input(BROADCASTS)
    SubscribableChannel broadcasts();

}
```

①

the only thing worth noting here is that these channels are annotated with `@Input` (naturally!) and that they return a `MessageChannel` subtype that supports *subscribing* to incoming messages, `SubscribableChannel`.

Remember, all bindings in Spring Cloud Stream are publish-subscribe by default. We can achieve the effect of exclusivity and a direct connection with a consumer group. Given, say, ten instances of a consumer in a group named `foo`, only one instance would see a message delivered to it. In a sufficiently distributed system, we can't be assured that a service will always be running,

so we'll take advantage of durable subscriptions to ensure that messages are redelivered as soon as a consumer connects to the broker.

Example 7-16. the `application.properties` for our consumer

```
spring.cloud.stream.bindings.broadcasts.destination = greetings-p  
①  
spring.cloud.stream.bindings.directed.destination = greetings-p2p  
spring.cloud.stream.bindings.directed.group = greetings-p2p-group  
spring.cloud.stream.bindings.directed.durableSubscription = true  
②  
  
server.port=0  
  
spring.rabbitmq.addresses=localhost  
  
①  
  
②
```

this should look fairly familiar given what we just covered in the producer

②

here we configure a destination, as before, but we *also* give our directed consumer an exclusive consumer group. Only one node among all the active consumers in the group `greetings-p2p-group` will see an incoming message. We ensure that the broker will store and redeliver failed messages as soon as a consumer is connected by specifying that the binding has a `durableSubscription`.

Finally, let's look at the Spring Integration Java DSL based consumer. This should look very familiar.

Example 7-17. a consumer driven by the Spring Integration Java DSL

```
package stream.consumer.integration;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.stream.annotation.EnableBinding;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.messaging.SubscribableChannel;
import stream.consumer.ConsumerChannels;

@SpringBootApplication
@EnableBinding(ConsumerChannels.class)
❶
public class StreamConsumer {

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

    @Bean
    IntegrationFlow direct(ConsumerChannels channels) {
        return incomingMessageFlow(channels.directed(), "direct")
    }

    @Bean
    IntegrationFlow broadcast(ConsumerChannels channels) {
        return incomingMessageFlow(channels.broadcasts(),
    }
}
```

as before, we see `@EnableBinding` activates the consumer channels.

②

This `@Configuration` class defines two `IntegrationFlow` flows that do basically the same thing take the incoming message, transform it by capitalizing it, and then logging it. One flow listens for the broadcasted greetings and the other for the direct, point-to-point greetings. We stop processing by returning `null` in the final component in the chain.

You can try it all out easily. Run one instance of one of the producer nodes (whichever one) and run three instances of the consumer. Visit `http://localhost:8080/hi/World` and then observe in the logs of the three consumers that all three have the message sent to the broadcast channel and one (although there's no telling which, so check all of the consoles) will have the message sent to the direct channel. Raise the stakes by killing all of the consumer nodes, then visiting `http://localhost:8080/hi/Again`. All the consumers are down, but we specified that the point-to-point connection be durable, so as soon as you restart one of the consumers you'll see the direct message arrive and logged to the console.

Spring Cloud Data flow

As we've moved through this chapter, we've moved forward and up the abstraction stack, where possible. We just looked at Spring Cloud Stream which makes short work of connecting messaging based microservices to each other. As far as our services are concerned, data comes in from a channel and it leaves through a channel. This is very similar to the way `stdin` and `stdout` on the command line work. Data-in and data-out. The transport is well-understood and all that the components need to understand in order to interoperate is what kind of payloads to produce or consume. In a UNIX `bash` environment, it's easy to compose arbitrarily complex solutions out of singly focused command line utilities, piping data through `stdin` and `stdout`. We can do the same thing with our Spring Cloud-based messaging microservices. Spring Cloud Stream raises the abstraction level so that we can focus on the business logic and ignore the details of communication. We can compose and orchestrate our messaging microservices with Spring Cloud Data Flow.

At the heart of Spring Cloud Data Flow are the concepts of streams and tasks. A **stream** represents a logical stringing together of different Spring Cloud Stream-based modules. Spring Cloud Data Flow ultimately launches the different services and overrides their default Spring Cloud Stream binding destinations so that data flows from one node to another in the way we describe. A **task** is any process whose execution status we want to inspect but that we also expect to, ultimately, terminate.

Spring Cloud Data Flow provides a powerful approach to orchestrate and compose Spring Cloud Stream-based messaging microservices and Spring Cloud Task-based jobs in a cloud environment. It sits on top of the notion of a deployer, which is an SPI that, well, *deploys* services for us and manages them on top of a distribution fabric like Cloud Foundry, Kubernetes or Apache YARN. In this example, we'll look at the *local* Spring Cloud Data Flow implementation, which needs only compiled `.jar` artifacts deployed into the local Maven repository to work.

A Spring Cloud Data Flow server provides a REST API to interrogate and manipulate streams and tasks. You can drive this API through the shell or, more usefully, through the Spring Cloud Data Flow shell. The shell might target any number of different Data Flow server instances, just as `git` can target any instance of a Git repository on any server.

Let's standup a Local Spring Cloud Data Flow Server. We can switch to a different implementation, but for ease of development let's stick with the local server. Start a new project in the usual way (from [the Spring Initializr](#), naturally!) and add `org.springframework.cloud:spring-cloud-starter-dataflow-server-local` to your application's Maven build, along with Spring Boot itself. Annotate the main class with `@EnableDataFlowServer` and then start it up. By default Spring Cloud Data Flow will spin up on port 9393.

Example 7-18. a bare Spring Cloud Data Flow Server

```
package dataflow;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.dataflow.server.EnableDataFlowSe
❶
❷ @EnableDataFlowServer
❸ @SpringBootApplication
public class DataFlowServer {

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

❶

this works just like the Spring Cloud Config Server, the Spring Cloud Hystrix Dashboard, Spring Cloud Eureka Server, etc.

With that done, launch the server and it'll spin up on port 9393. We can't do anything with it yet, though. We need a client! We could use the `RestTemplate` and work with all the various APIs. The APIs are handily laid

out as HATEOAS links when you visit `http://localhost:9393`. We can also visit the Spring Boot Actuator endpoints exposed under `http://localhost:9393/management/` by default.

By default, our Data Flow server is a blank canvas. We can register custom applications and tasks, as we like, but there are many interesting applications that come from the [Spring Cloud Stream modules project](#). Let's point our Data Flow server to this rich set of existing modules. Spring Cloud Data Flow knows about tasks and applications. Applications are easily subdivided into **sources** (where messages are produced), **processors** (where an incoming message is somehow processed and then sent out) and **sinks** where an incoming message is consumed. A source and a sink correspond more or less to the notion of a Spring Integration inbound adapter and outbound adapter, respectively. The Spring Cloud Data Flow local server doesn't know about these many modules, but we can import their definitions easily. Modules are connected through Spring Cloud Stream bindings, which may in turn interconnect using RabbitMQ, Redis or Apache Kafka.

Example 7-19. importing the RabbitMQ-bound modules.

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabb
```

We could at this point interact with the server using the REST API, but that's nowhere near as fun (or productive!) as using the Spring Cloud Data Flow shell. Start a new project from [the Spring Initializr](#) and add the Spring Cloud Data Flow Shell (`org.springframework.cloud:spring-cloud-dataflow-shell`) to the build. The Shell uses the Spring Shell project which in turn has its own banner contribution. You can disable the default Spring Boot banner by setting `spring.main.banner-mode` to `off` in your `application.properties`. Then, on your operating system's command line, run `mvn clean spring-boot:run`.

Tip

You could launch this program from your IDE but some IDEs have trouble with interactive shells in the IDE console output, so better to use an actual OS shell.

Run the shell and then issue `app list` and you should see output like this.



1.0.0.BUILD-SNAPSHOT

Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
dataflow:>app list

| source | processor | sink | task |
|----------------|----------------------|---------------------|------|
| file | bridge | aggregate-counter | |
| ftp | filter | cassandra | |
| http | groovy-filter | counter | |
| jdbc | groovy-transform | field-value-counter | |
| jms | httpclient | file | |
| load-generator | pmml | ftp | |
| rabbit | scriptable-transform | gemfire | |
| sftp | splitter | gpfdist | |
| tcp | transform | hdfs | |
| time | | jdbc | |
| trigger | | log | |
| twitterstream | | rabbit | |
| | | redis | |
| | | router | |
| | | tcp | |
| | | throughput | |
| | | websocket | |

dataflow:>■

Figure 7-3. the Spring Cloud Data Flow shell when connected to the Spring Cloud Data Flow server with the default modules in place.

Streams

Not bad! Let's take the server for a spin. We'll create a simple stream using two default apps; `time`, which produces a new message on a timer, and `log` which logs incoming messages to stdout. In the Spring Cloud Data Flow shell, run the following:

Example 7-20.

```
dataflow:>stream create --name ticktock --definition "time | log"  
Created new stream 'ticktock'
```

❶

```
dataflow:>stream deploy --name ticktock  
Deployed stream 'ticktock'
```

❷

this defines a stream that takes produced times and then pipes them to the logs

❸

the stream needs to be deployed. This will invoke the underlying deployer that in turn launches the module. In the local case, it basically does `java -jar ..` on the resolved `.jar` for the module. On Cloud Foundry it'll start up new app instances for each application.

`time` and `log` are each full Spring Boot applications. They expose well-known channels: `output` for the source `time`, `input` for the sink `log`) and Spring Cloud Data Flow stitches them together, arranging for messages that leave the `log` Spring Boot application's `output` channel to be routed to the `time` Spring Boot application's `input` channel. In the console of the Data Flow server you'll observe output confirming that the involved applications have been launched and are running:

Example 7-21. the logs confirming that the applications have been launched and

pointing us to the various logs

```
...
2016-06-07 01:37:45.568  INFO 24156 --- [nio-9393-exec-1] o.s.web
2016-06-07 01:37:51.244  INFO 24156 --- [nio-9393-exec-3] o.s.c.d
    Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn
2016-06-07 01:37:51.259  INFO 24156 --- [nio-9393-exec-3] o.s.c.d
    Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn
...
...
```

The `stdout` logs for the `log` application should reflect a never-ending stream of new timestamps.

Let's look at a slightly more involved stream that integrates a custom processor component and sits between the `time` and `log` applications. Add `org.springframework.integration: spring-integration-java-dsl`, `org.springframework.cloud: spring-cloud-starter-stream-rabbit` to the classpath to activate Spring Cloud Stream (and the RabbitMQ binder) and Spring Integration's Java DSL.

Example 7-22. A simple Spring Integration processor that accepts input messages on a well-defined `MessageChannel`, `input`, and sends the processed input message out on `output`

```
package stream;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

@MessageEndpoint
❶ @EnableBinding(Processor.class)
❷ @SpringBootApplication
public class ProcessorStreamExample {

    @ServiceActivator(inputChannel = Processor.INPUT, outputC
```

```

        public Message<String> process(Message<String> in) {
            return MessageBuilder.withPayload("{" + in.getPay
                .copyHeadersIfAbsent(in.getHeader
            }

        public static void main(String[] args) {
            SpringApplication.run(ProcessorStreamExample.clas
        }
    }
}

```

①

marks this class as a Spring Integration messaging endpoint

②

activates the Spring Cloud Stream binding that connects our `input` and `output` channels to the broker

③

take any incoming message and surround the payload in `{` and `}`.

Run a `mvn clean install` to install the task into the local Maven repository and note its coordinates. We'll register it using the Spring Cloud Data Flow shell and then deploy a stream where it figures as a component.

Example 7-23. using the shell to register and deploy a stream

```
dataflow:>app register --name brackets --type processor --uri mav
Successfully registered application 'processor:brackets'
```

②

```
dataflow:>stream create --name bracketedticktock --definition "ti
Created new stream 'bracketedticktock'
```

③

```
dataflow:>stream list
```

| Stream Name | Stream Definition | Status |
|-------------------|-----------------------|------------|
| bracketedticktock | time brackets log | undeployed |

④

```
dataflow:>stream deploy --name bracketedticktock  
Deployed stream 'bracketedticktock'
```

①

we must first register the application so that Spring Cloud Data Flow is aware of it

②

create a stream that stitches the `time`, `log` and the newly-registered `bracket` component together.

③

confirm that the stream's been registered

④

deploy the stream

Tasks

Spring Cloud Data Flow records the stream and task definitions in the metadata repository it maintains, backed by a SQL database. It uses H2 by default, so we can hit the ground running, though of course you could override the H2 database.

Let's kick off a simple task and see what that looks using the `timestamp` task.

Example 7-24.

```
dataflow:>task create --name whattimeisit --definition timestamp  
Created new task 'whattimeisit'
```

②

```
dataflow:>task list
```

| Task Name | Task Definition | Task Status |
|--------------|-----------------|-------------|
| whattimeisit | timestamp | unknown |

③

```
dataflow:>task launch --name whattimeisit  
Launched task 'whattimeisit'
```

④

```
dataflow:>task list
```

| Task Name | Task Definition | Task Status |
|--------------|-----------------|-------------|
| whattimeisit | timestamp | complete |

①

create a task that launches the very simple Spring Cloud Task called `timestamp` and prints out the value

②

enumerate the existing tasks

③

launch the task and give it a logical name by which to refer to it later

④

list the tasks. In this case the task is complete and we see that reflected.

Inspect the logs and you should see the time printed.

Let's look at a slightly more involved task that manages a Spring Batch Job.

Add `org.springframework.cloud:spring-cloud-task-core`,
`org.springframework.cloud:spring-cloud-task-batch`, and
`org.springframework.boot:spring-boot-starter-batch` to the classpath
to activate Spring Batch, Spring Cloud Task and the Spring Cloud Task
bridge.

Example 7-25. A simple Spring Batch job defined in a Spring Cloud Task example

```
package task;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.step.tasklet.TaskletStep;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.context.annotation.Bean;

@EnableTask
①
@EnableBatchProcessing
②
@SpringBootApplication
public class BatchTaskExample {
```

```

    @Bean
    Job hello(JobBuilderFactory jobBuilderFactory,
              StepBuilderFactory stepBuilderFactory) {

        // @formatter:off

        Log log = LoggerFactory.getLog(getClass());

        TaskletStep step = stepBuilderFactory
            .get("one")
            .tasklet((stepContribution, chunk
                log.info(
                log.info(
                    chunkCont

                    return Re
                }).build();

        return jobBuilderFactory.get("hello").start(step)

        // @formatter:on
    }

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

```

①

activate Spring Cloud Task

②

activate Spring Batch

③

create a simple Spring Batch job that has one Step which uses a Spring Batch Tasklet instead of specifying a chunked ItemReader , ItemProcessor and ItemWriter, etc., in the interest of keeping things

simple. A tasklet in Spring Batch is a place to insert any sort of generic logic. This particular tasklet logs the arguments passed in.

Run a `mvn clean install` to install the task into the local Maven repository and note its coordinates. We'll register it usnig the Spring Cloud Data Flow shell and then launch an instance of it.

Example 7-26. using the shell to register and launch a Batch task

```
dataflow:>app register --name args --type task --uri maven://cnj:  
Successfully registered application 'task:args'
```

②

```
dataflow:>task create --definition "args --p1=1 --p2=2" --name ar  
Created new task 'args'
```

③

```
dataflow:>task launch --name args  
Launched task 'args'
```

④

```
dataflow:>task list
```

| Task Name | Task Definition | Task Status |
|-----------|-----------------|-------------|
| args | args | complete |

①

we must first register the application so that Spring Cloud Data Flow is aware of it

②

create a (potentially) parameterized instance of the application.

③

launch an instance of the task

④

note the task's exit status

Next Steps

We've only begun to scratch the surface of the possibilities in data processing. Naturally, each of these technologies speaks to a large array of different technologies themselves. We might, for example, use Apache Spark or Apache Hadoop in conjunction with Spring Cloud Data Flow. They also compose well. It's trivial to scale out processing of a Spring Batch job across a cluster using Spring Cloud Stream as the messaging fabric.

Chapter 8. Using Spring Boot with Java EE

In this chapter we'll look at how to integrate Spring Boot applications with Java EE. Java EE, for our purposes, is an umbrella name for a set of APIs and, sometimes, runtimes - *Java EE application servers*. Java EE application servers, like [RedHat's WildFly AS](#) - the application server formerly named JBoss Application Server - provide implementations of these APIs. We'll look at how to build applications that leverage Java EE APIs outside of a Java EE application server. If you're building a brand new application today, you don't need this chapter. This chapter is more useful for those with existing functionality trapped in an application server that want to move to a microservices architecture. For a broader discussion of moving ("forklifting") legacy applications to a cloud platform like Cloud Foundry with minimal refactoring, see the chapter on [Chapter 10](#).

Spring acts as a consumer of Java EE APIs, where practical. It doesn't *require* any of them, or any of the myriad XML configuration files that go along with them. Wherever possible, Spring supports consuming Java EE APIs à la carte, independent of a full Java EE application server. Spring applications should ideally be portable across environments, including embedded web applications, application servers, and virtually any platform-as-a-service (PaaS) offering. Spring also works in any application server, as well.

Compatibility and Stability

Spring 4.2 (the baseline release for Spring Boot 1.3 and later) supports Java SE 6 or later (specifically, the minimum API level is JDK 6u18, which was released in early 2010). Oracle has stopped publishing updates - save for security fixes - for Java SE 6 and Java SE 7. Consider moving to Java SE 8.

Spring also supports Java EE 6+ (which came out in 2009). In practical terms, Spring 4 is Servlet 3.0+ focused but Servlet 2.5 compatible. Spring Boot, which builds on Spring, works best with Servlet 3.1 or later. Spring Boot's approach to dependencies ensure that you get the latest and greatest, as soon as possible, though you can revert to older Servlet 3.0 generation web servers if you like.

Java EE isn't a large market. The market consistently favors, by a wide margin, [Apache's Tomcat](#) (a project that Pivotal contributes heavily to), Eclipse's Jetty, and RedHat's Wildfly. What's up for debate is whether [WildFly](#) is in the number two position or whether Jetty is. My friend Arun Gupta (who works at RedHat as their head of evangelism) put together a nice blog on [WildFly's penetration in the last 5 years](#). In all the charts, Tomcat is #1, and WildFly's in the top three. WildFly, let's say, is the most popular *full* Java EE application server implementation, and it represents the sole Java EE implementation with any significant marketshare and less than a third of the market.

Put another way: most developers don't deploy to Java EE-compatible containers. Spring brings useful Java EE APIs to the growing majority that aren't running Java EE application servers.

Java EE offers some very compelling APIs. The Servlet API, JSR 330 (`javax.inject`) and JSR 303 (`javax.validation`), JCache, JDBC, JPA, JMS, etc., to name a few, are very useful and - where practical for users - Spring supports them. Indeed, sometimes Spring offers support months or years before certified Java EE implementations, as it did with JSR 107 (the JCache API).

As developers move to the cloud, and as a result, to a cloud-native architecture, we see decreasing value in relying on Java EE application servers; the all-or-nothing approach goes against the grain of building cloud-native systems composed of small, singly focused, scalable services.

Some Java EE APIs are very useful, but they're notoriously slow to evolve. This suggests that the benefits of slow-to-evolve, standardized Java EE APIs are best reaped at layers where there is little volatility. JDBC, which provides the interface for decades-old SQL-based databases, is an example of a perfectly reasonable API.

Java EE 8, as a standard, is due to be released (*very tentatively*) in the early part of 2017. At the time of this writing, the Java EE 8 timeline is a few months off-track. A release in 2017 implies a production timeline of 2019 or longer (because the first major, commercially supported Java EE 7 application server came from IBM more than two years after the release of the standard). Java EE 8 offers no support for NoSQL, or cloud-based applications, but it *does*, very importantly, finally offer [a standard API for binding JSON to Java objects](#)! This support is very similar to what Spring supports through the *non-standard* Jackson and GSON libraries in one form or another since 2009. Your choice is simple: prefer business value, or prefer standards. If you want standards, then choose them where they are not differentiators, where they may as well be commoditized: HTTP, REST, JDBC, etc., are good examples. Another great reason to choose a standard is for ubiquity: JPA is a *very* ubiquitous technology and is used in *many* Spring applications, outside of Java EE application servers. Community and the associated hiring pool, in this case, are the features.

Software serves a purpose, usually a business purposes, and businesses don't compete by relying on sometimes decades-old, inferior solutions. If we, on the JVM, are to thrive then we must be able to develop and deploy solutions competitively. Imagine suggesting to a Node.js or Rails developer that they *may* get JSON support in 2019!

Compatibility is a feature, too. Spring tries to retain backwards-compatibility. An application written using Spring 1.0 can in 99% of situations be upgraded with a drop-in replacement of the newer release libraries of Spring, even on older generations of Java EE. Replace the dependencies for an application

written against J2EE (what Java EE was called before it was rebranded ten years or so ago from J2EE and J2SE and J2ME to *Java EE*, *Java SE*, and *Java ME*) with modern Java EE dependencies and chunks of the code - based on EJB 1.0-2.1, for example - would not compile, let alone run.

With that quick discussion of the wisdom of Java EE in modern applications behind us, let's look at a few commonplace APIs that you'll find work marvelously in Spring Boot applications.

Dependency Injection with JSR 330 (and JSR 250)

Spring has long offered various approaches to providing configuration. Spring doesn't care, really, where it learns about your objects and how you wire them together. In the beginning there was the XML configuration format. Later, Spring introduced component scanning to discovery and register components with stereotype annotations like `@Component`. In 2006 the Spring Java Configuration project was born. This approach describes beans to Spring in terms of objects returned from methods that are annotated with the `@Bean` annotation. You might call these bean definition *provider methods*.

Meanwhile, at Google, the amazing Bob Lee introduced Guice, which also provides the ability to define bean definition provider methods, like the Spring Java configuration project. These two options, along with a few others, evolved independently and each garnered a large community.

Let's skip ahead to 2007, 2008, and 2009 and formative days of Java EE 6. The team behind JBoss' Seam who had developed their own dependency injection technology attempted to define a standard, JSR 299 (CDI), to define what it means to be a dependency injection technology. Naturally, neither Seam nor JSR 299 looked anything like Spring or Guice, the two most entrenched and popular technologies, *by any stretch*, so Spring founder Rod Johnson and Guice founder Bob Lee proposed JSR 330. JSR 330 defines a common set of annotations for the surface area of dependency injection frameworks *that impact business component code*. This left each dependency injection container to differentiate in the way configuration itself is handled.

Tip

JSR 330 was not offered as a large, tedious specification with hundreds of pages, but instead as a tiny set of JavaDocs for the handful of annotations and a single interface in the proposed API. Many (particularly those behind JSR 299) felt it was an affront to the JCP

process, and it was even suggested that it could never make it through the JCP process in time for inclusion in Java EE 6, but it did! At the time, it was the fastest JCP standard ever.

JSR 330 is natively supported, of course, by Spring and Guice and - because cooler heads eventually prevailed - by JSR 299 implementations. It's so common in fact other DI technologies like Dagger, which is optimized for compile-time code-generation and mobile environments like Android, also support it. If you need to have portable dependency injection, use JSR 330.

You'll commonly use a few annotations with JSR 330, assuming you have `javax.inject : javax-inject : 1` on the classpath, to define, resolve and inject bean references.

- `@Inject` is equivalent to Spring's `@Autowired` annotation. It identifies injectable constructors, methods and fields.
- `@Named` is more or less equivalent to Spring's various stereotype annotations like `@Component`. They mark a bean to be registered with the container and it can be used to give the bean a String-based ID by which it may be registered.
- `@Qualifier` can be used to *qualify* beans by type (or String ID). This is, naturally, almost identical to Spring's `@Qualifier` annotation.
- `@Scope` is more or less analogous to Spring's `@Scope` annotation and is used to tell the container what the lifecycle of a bean is. You might, for example, specify that a bean is `session` scoped, that is - it should live and die along the lines of an HTTP request.
- `@Singleton` tells the container that the bean should be instantiated only once. This is Spring's default, but it's such a common concept that it was worth making sure all implementations support it.

JSR 330 also defines one interface, `javax.inject.Provider<T>`. Business components can inject instances of a given bean, `Foo`, directly or using a `Provider<Foo>`. This is more or less analogous to Spring's `ObjectFactory<T>` type. Compared to injecting an instance directly, the

`Provider<T>` instance can be used to retrieve multiple instances of a given bean, handle *lazy* semantics, break circular dependencies, and abstract containing scope.

Tip

JSR 250, which comes from Java EE 5, is *also* supported natively in Spring if the types are on the classpath (they are in newer versions of the JDK). You may have seen these annotations if you've ever used

`@javax.annotation.Resource`, for example. These annotations are commonly used in EJB 3 code, but I've never really seen them used elsewhere. They can make it easy for developers moving code over from EJB 3 environments where references resolution is signalled with `@Resource`.

Building REST APIs with JAX-RS (Jersey)

Spring Boot makes it dead simple to stand up REST APIs using JAX-RS. JAX-RS is a standard and requires an implementation. Our example [demonstrates Boot's JAX-RS auto-configuration for Jersey 2.x](#) in the GreetingEndpoint. The example uses the Spring Boot starter org.springframework.boot : spring-boot-starter-jersey.

Example 8-1. The JAX-RS `GreetingEndpoint`

```
package demo;

import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Named
❶ @Path("/hello")
❷ @Produces({MediaType.APPLICATION_JSON})
❸ public class GreetingEndpoint {

    @Inject
    private GreetingService greetingService;

    @POST
❹     public void post(@QueryParam("name") String name) { ❺
        this.greetingService.createGreeting(name);
    }

    @GET
    @Path("/{id}")
    public Greeting get(@PathParam("id") Long id) {
        return this.greetingService.find(id);
    }
}
```

```
    }  
}
```

❶

JSR 330's `@Inject` annotation

❷

JAX-RS's `@Path` annotation is functionally equivalent to Spring MVC's `@RequestMapping`. It tells the container under what route this endpoint should be exposed.

❸

Spring MVC's `@RequestMapping` provides a `produces` and `consumes` attribute that lets you specify what content-types a given endpoint can consume, or produce. In JAX-RS, this mapping is done with standalone annotations.

❹

The HTTP verb, also otherwise specified in Spring MVC's `@RequestMapping` annotation, is specified here as a standalone annotation.

❺

The `@QueryParam` annotation tells JAX-RS to inject any incoming request parameters (`?name==..`) as method arguments. In Spring MVC, you'd use `@RequestParam`-annotated method arguments, instead.

Jersey requires a `ResourceConfig` subclass to enable basic features and register components.

Example 8-2. The `ResourceConfig` subclass that configures Jersey

```
package demo;  
  
import org.glassfish.jersey.jackson.JacksonFeature;  
import org.glassfish.jersey.server.ResourceConfig;
```

```

import javax.inject.Named;

@Named
❶
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        this.register(GreetingEndpoint.class); ❷
        this.register(JacksonFeature.class); ❸
    }
}

```

❶

we register the endpoint using JSR 330, though we could just as easily have used any of Spring's stereotype annotations here. They're interchangeable.

❷

we need to explicitly register our JAX-RS endpoint.

❸

we need to tell JAX-RS that we want to handle JSON marshalling. Java EE doesn't have a built-in API for marshalling JSON (as we discussed above, it'll tentatively be available in Java EE 8), but you can use Jersey-specific *feature* implementations to plugin popular JSON-marshalling implementations like Jackson.

Spring Boot auto-configures the Jersey

`org.glassfish.jersey.servlet.ServletContainer` as both a `javax.servlet.Servlet` and a `javax.servlet.Filter` that listens for all requests relative to the application root.

In JAX-RS, message encoding and decoding is done through the `javax.ws.rs.ext.MessageBodyWriter` and `javax.ws.rs.ext.MessageBodyReader` SPIs, somewhat akin to Spring MVC's `org.springframework.http.converter.HttpMessageConverter` hierarchy. By default, JAX-RS does not have many useful message body readers and writers enabled. The example registers a `JsonFeature` in the `ResourceConfig` subclass to support JSON encoding and decoding.

JAX-RS is a capable API, but it's forever constrained to evolve slower, and be supported by a smaller community that is itself fractured across a handful of implementations. If you have code using JAX-RS, it's comforting to know it works, but consider using a more mature REST toolkit, otherwise. Spring offers a richer, integrated Spring MVC-based stack complete with MVC, REST, HATEOAS, OAuth, SSE, and websocket support.

JTA and XA Transaction Management

Resource-Local Transactions with Spring's PlatformTransactionManager

Fundamentally, transactions work more or less the same way: a client begins work, does some work, commits the work and - if something goes wrong - restores (*rolls back*) the state of a resource to how it was before the work began. Implementations across the numerous resources vary considerably. JMS clients create a *transacted* Session that is then committed. JDBC Connection can be made to *not* auto-commit work, which is the same as saying it batches work, and then commit when explicitly instructed to do so.

Resource-local transactions should be your default approach for transaction management. You'll use them with various transactional Java EE APIs like JMS, JPA, JMS, CCI, and JDBC. Spring supports numerous other transactional resources like AMQP-brokers such as [RabbitMQ](#), [the Neo4j graph database](#), and [Pivotal Gemfire](#). Unfortunately, each of these transactional resources offers a different API for initiating, committing, or rolling back work. To simplify things, Spring provides the PlatformTransactionManager hierarchy. There are numerous, pluggable implementations of PlatformTransactionManager that adapt the various notions of transactions to a common API. Spring is able to manage transactions through implementations of this hierarchy. Spring provides tiered support for transactions.

At the lowest level, Spring provides the TransactionTemplate. The TransactionTemplate wraps a PlatformTransactionManager bean and uses it to manage a transaction given a unit of work which the client provides as an implementation of TransactionCallback. Let's first wire everything up. We have some things that will be common to the following two examples - a database DataSource, a DataSourceInitializer, a JdbcTemplate, a PlatformTransactionManager, etc. - so we'll define them in a shared Java configuration class. This Java configuration also defines a RowMapper, which is a JdbcTemplate callback interface that maps rows of results to Java objects.

```

package basics;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.datasource.DataSourceTransactionM
import org.springframework.jdbc.datasource.SimpleDriverDataSource
import org.springframework.jdbc.datasource.init.DataSourceInitial
import org.springframework.jdbc.datasource.init.ResourceDatabaseP
import org.springframework.transaction.PlatformTransactionManager

import javax.sql.DataSource;

@Configuration
public class TransactionalConfiguration {

    ①
    @Bean
    DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDri
        dataSource
            .setUrl("jdbc:h2:mem:test;DB_CLOS")
        dataSource.setDriverClass(org.h2.Driver.class);
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    ②
    @Bean
    DataSourceInitializer dataSourceInitializer(DataSource ds
        @Value("classpath:/schema.sql") Resource
        @Value("classpath:/data.sql") Resource da
    DataSourceInitializer init = new DataSourceInitia
    init.setDatabasePopulator(new ResourceDatabasePop
    init.setDataSource(ds);
    return init;
}

    ③
    @Bean
    JdbcTemplate jdbcTemplate(DataSource ds) {
        return new JdbcTemplate(ds);
}

```

```

④
@Bean
RowMapper<Customer> customerRowMapper() {
    return (resultSet, i) -> new Customer(resultSet.get
        resultSet.getString("FIRST_NAME")
        resultSet.getString("LAST_NAME"))
}

⑤
@Bean
PlatformTransactionManager transactionManager(DataSource ds) {
    return new DataSourceTransactionManager(ds);
}
}

```

①

define a `DataSource` that just talks to an in-memory embedded H2 database

②

define a `DataSourceInitializer` that runs schema and data initialization DDL

③

a Spring `JdbcTemplate` which reduces common JDBC calls to one liners

④

a `RowMapper` is a callback interface that `JdbcTemplate` uses to map SQL `ResultSet` objects into objects, in this case of type `Customer`.

⑤

the `DataSourceTransactionManager` bean adapts the `DataSource` transactions to Spring's `PlatformTransactionManager` hierarchy.

We can use Spring's low-level `TransactionTemplate` to demarcate transaction boundaries explicitly.

```
package basics.template;
```

```

import basics.Customer;
import basics.TransactionalConfiguration;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbac
import org.springframework.transaction.support.TransactionTemplat

@Configuration
@ComponentScan
@Import(TransactionalConfiguration.class)
public class TransactionTemplateApplication {

    public static void main(String args[]) {
        new AnnotationConfigApplicationContext(
            TransactionTemplateApplication.cl
    }

❶
@Bean
TransactionTemplate transactionTemplate(PlatformTransacti
    return new TransactionTemplate(txManager);
}

@Service
class CustomerService {

    private JdbcTemplate jdbcTemplate;
    private TransactionTemplate txTemplate;
    private RowMapper<Customer> customerRowMapper;

    @Autowired
    public CustomerService(TransactionTemplate txTemplate,
                           JdbcTemplate jdbcTemplate, RowMapper<Cust
        this.txTemplate = txTemplate;
        this.jdbcTemplate = jdbcTemplate;
        this.customerRowMapper = customerRowMapper;
    }

    public Customer enableCustomer(Long id) {

```

```

②
    TransactionCallback<Customer> customerTransaction
        TransactionStatus transactionStat

        String updateQuery = "update CUSTOMER set
            jdbcTemplate.update(updateQuery, Boolean.

        String selectQuery = "select * from CUSTO
            return jdbcTemplate.queryForObject(select
                id);
        };

③
    Customer customer = txTemplate.execute(customerTr

    LogFactory.getLog(getClass()).info(
        "retrieved customer # " + custome

    return customer;
}
}

```

①

the `TransactionTemplate` requires only a `PlatformTransactionManager`

②

a `TransactionCallback` defined with Java 8 lambdas. The body of this method will be run in a valid transaction and committed and closed upon completion.

③

the return value could be whatever you want, but I think this is in itself an interesting clue: transactions should ultimately be towards obtaining one, valid, biproduct.

The `TransactionTemplate` should be familiar if you've ever used any of Spring's other template implementations. The `TransactionTemplate` defines transaction boundaries explicitly. It's very useful when you want to cordon

off sections of logic in a transaction, and to control when that transaction starts and stops.

Use annotations to declaratively demarcate transactional boundaries. Spring has long supported declarative transaction rules, both external to the business components to which the rules apply, and inline. The most popular way to define transaction rules as of Spring 1.2, released in May 2005 just after Java SE 5, is to use Java annotations.

So, what's all this to do with Java EE? Spring supports declarative transaction management with its `@Transactional` annotation. It can be placed on a type or on individual methods. The annotation can be used to specify transactional qualities such as propagation, the exceptions to rollback for, etc.

A year later, Java EE 5 debuted and included EJB 3 which defined an annotation-based way to define transaction boundaries with its `javax.ejb.TransactionAttribute` annotation. Spring *also* honors this annotation if it's discovered in Spring beans. The `TransactionAttribute` works well for EJB based business components, but the approach falls apart fast outside of EJB based components. JTA 1.2, which was included in Java EE 7 in 2013 (but which, as we discussed above, isn't readily available and not at all supported in production as of April 2015), defined `javax.transaction.Transactional` as a general purpose transaction boundary annotation, more or less like Spring's `@Transactional` from 8 years earlier. Spring *also* honors this annotation, if present.

The configuration to make this work is basically the same as with the `TransactionTemplate`, except that we turn on annotation-based transaction boundaries with the `@EnableTransactionManagement` annotation and don't need the `TransactionTemplate` bean anymore.

```
package basics.annotation;

import basics.Customer;
import basics.TransactionalConfiguration;
import basics.template.TransactionTemplateApplication;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
```

```

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.EnableTransacti

@Configuration
@ComponentScan
@Import(TransactionalConfiguration.class)
@EnableTransactionManagement
❶
public class TransactionalApplication {

    public static void main(String args[]) {
        new AnnotationConfigApplicationContext(
            TransactionTemplateApplication.cl
    }
}

@Service
class CustomerService {

    private JdbcTemplate jdbcTemplate;
    private RowMapper<Customer> customerRowMapper;

    @Autowired
    public CustomerService(JdbcTemplate jdbcTemplate,
                           RowMapper<Customer> customerRowMapper) {
        this.jdbcTemplate = jdbcTemplate;
        this.customerRowMapper = customerRowMapper;
    }

    // @org.springframework.transaction.annotation.Transactio
❷
    // @javax.ejb.TransactionAttribute ❸
    @javax.transaction.Transactional
❹
    public Customer enableCustomer(Long id) {

        String updateQuery = "update CUSTOMER set ENABLED
jdbcTemplate.update(updateQuery, Boolean.TRUE, id

        String selectQuery = "select * from CUSTOMER wher
Customer customer = jdbcTemplate.queryForObject(s
customerRowMapper, id);

```

```
        LogFactory.getLog(getClass()).info(
            "retrieved customer # " + customer);
    }
}
```

①

@EnableTransactionManagement turns on the transaction processing. It requires that a valid PlatformTransactionManager be defined somewhere.

②

You could use Spring's

```
@org.springframework.transaction.annotation.Transactional
```

③

or EJB's @javax.ejb.TransactionAttribute

④

or JTA 1.2's @javax.transaction.Transactional

Prefer Spring's @Transactional variant. It exposes more power than the EJB3 alternative in that it is able to expose transactional semantics (things like transaction suspension and resumption) that the EJB-specific @TransactionAttribute (and indeed EJB itself) don't expose, and doesn't require an extra JTA or EJB dependency. It is nice to know that it will work, either way, however.

These examples require only a DataSource driver and
org.springframework.boot : spring-boot-starter-jdbc.

Global Transactions with the Java Transaction API (JTA)

Spring makes working with *one* transactional resource easy. But what is a developer to do when trying to transactionally manipulate more than one transactional resource, e.g., a *global transaction*? For example, how should a developer transactionally write to a database *and* acknowledge the receipt of a JMS message? *Global* transactions are the alternative to *resource-local* transactions; they involve *multiple* resources in a transaction. To ensure the integrity of a global transaction, the coordinator must be an agent independent of the resources enlisted in a transaction. It must be able to guarantee that it can replay a failed transaction, and that it is itself immune to failures. JTA (necessarily) adds complexity and state to the process that a resource-local transaction avoids. Most global transaction managers speak the [X/Open XA protocol](#) which lets transactional resources (like a database or a message broker) enlist and participate in global transactions. Java EE provides a very handy API on top of this protocol called [JTA](#).

Integration is, at least in theory, simple. For our purposes, it suffices to know that JTA exposes two key interfaces - the [javax.transaction.UserTransaction](#) and [javax.transaction.TransactionManager](#). The UserTransaction supports the usual suspects: `begin()`, `commit()`, `rollback()`, etc. Clients use this object to begin a global transaction. While the global transaction is open, JTA-aware resources like a JTA-aware JDBC `javax.sql.XADataSource`, or a JTA-aware JMS `javax.jms.XAConnectionFactory` may *enlist* in the JTA transaction. They communicate with the global transaction coordinator and follow a protocol to either atomically commit the work in all enlisted resources or rollback. It is up to each atomic resource to support, or not support, the JTA protocol, which itself speaks the XA protocol.

In a Java EE container, the `UserTransaction` object is *required* to exist in a JNDI context under the binding `java:comp/UserTransaction`, so it's easy for Spring's `JtaTransactionManager` to locate it. This simple interface is enough to handle *basic* transaction management chores. Notably, it is *not*

sophisticated enough to handle sub-transactions, transaction suspension and resumption, or other features typical of quality JTA implementations. Instead, use a `TransactionManager` instance, if it's available. Java EE application servers are *not* required to expose this interface, and *rarely* expose it under the same JNDI binding. Spring knows the well-known contexts for many popular Java EE application servers, and will attempt to automatically locate it for you.

Tip

Often, the bean that implements `javax.transaction.UserTransaction` *also* implements `javax.transaction.TransactionManager`!

JTA can also be run outside of a Java EE container. There are numerous popular third party (and open-source) JTA implementations, like Atomikos and Bitronix. Spring Boot provides [auto-configurations for both](#).

[Atomikos](#) is commercially supported and provides an open-source base edition. Let's look at an example - in `GreetingService` - that uses JMS to send notifications and JPA to persist records to an RDBMS as part of a global transaction.

```
package demo;

import org.springframework.jms.core.JmsTemplate;

import javax.inject.Inject;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Collection;

@Named
@Transactional
❶
public class GreetingService {

    @Inject
    private JmsTemplate jmsTemplate; ❷

    @PersistenceContext
```

```

❸
private EntityManager entityManager;

public void createGreeting(String name, boolean fail) { ❹
    Greeting greeting = new Greeting(name);
    this.entityManager.persist(greeting);
    this.jmsTemplate.convertAndSend("greetings", gree
        if (fail) {
            throw new RuntimeException("simulated err
        }
    }

public void createGreeting(String name) {
    this.createGreeting(name, false);
}

public Collection<Greeting> findAll() {
    return this.entityManager.createQuery(
        "select g from " + Greeting.class
        Greeting.class).getResultList();
}

public Greeting find(Long id) {
    return this.entityManager.find(Greeting.class, id
}
}

```

❶

We tell Spring that all public methods on the component are transactional.

❷

This code uses Spring's JmsTemplate JMS client to work with a JMS resource.

❸

This code uses the spring-boot-starter-data-jpa support so can inject a JPA EntityManager with JPA's @PersistenceContext annotation as per normal Java EE conventions.

❹

This method takes a boolean that triggers an exception if `true`. This exception triggers a rollback of the JTA transaction. You'll only see evidence that two of the three bodies of work completed after the code is run.

We demonstrate this in `GreetingServiceClient` by creating three transactions and simulating a rollback on the third one. You should see printed to the console that there are two records that come back from the JDBC `javax.sql.DataSource` data source and two records that are received from the embedded JMS `javax.jms.Destination` destination.

```
package demo;

import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;
import java.util.logging.Logger;

@Named
public class GreetingServiceClient {

    @Inject
    private GreetingService greetingService;

    @PostConstruct
    ①
    public void afterPropertiesSet() throws Exception {
        greetingService.createGreeting("Phil");
        greetingService.createGreeting("Dave");
        try {
            greetingService.createGreeting("Josh", true);
        } catch (RuntimeException re) {
            Logger.getLogger(Application.class.getName())
                .caught exception...");
        }
        greetingService.findAll().forEach(System.out::println);
    }
}
```

①

We haven't seen `@PostConstruct` yet. It's part of JSR 250, and is semantically the same as Spring's `InitializingBean` interface. It defines a callback to be invoked *after* the beans dependencies - be they

constructor arguments, JavaBean properties, or fields.

Tip

Spring Boot will automatically setup JPA based on the configured `DataSource`. This example [uses Spring Boot's embedded `DataSource` support](#). If an embedded database (like [H2](#), which is what we're using here, or Derby, and HSQL) is on the classpath, and no `javax.sql.DataSource` is explicitly defined, Spring Boot will create a `DataSource` bean for use.

Spring Boot makes it dead simple to setup a JMS connection, as well. Just like the embedded `DataSource`, Spring Boot can also [create an embedded `JMS ConnectionFactory`](#) using RedHat's [HornetQ](#) message broker in embedded mode assuming the correct types are on the classpath (`org.springframework.boot: spring-boot-starter-hornetq`) and a few properties are specified:

```
spring.hornetq.mode=embedded  
spring.hornetq.embedded.enabled=true  
spring.hornetq.embedded.queues=greetings
```

With that in place, just add the requisite Spring Boot starters (`org.springframework.boot: spring-boot-starter-hornetq`) and HornetQ support (`org.hornetq: hornetq-jms-server`) to activate the right auto-configurations.

If you wanted to connect to traditional, non-embedded instances, it's straightforward to either specify properties in `application.yml` or `application.properties` like `spring.datasource.url` and `spring.hornetq.host`, or simply define `@Bean` definitions of the appropriate types.

This example uses the Spring Boot Atomikos starter (`org.springframework.boot: spring-boot-starter-atomikos`) to configure Atomikos and the XA-aware JMS and JDBC resources.

In today's [distributed world, consider global transaction managers an architecture smell](#). Distributed transactions gate the ability of one service to

process transactions at an independent cadence. Distributed transactions imply that state is being maintained across multiple services when they should ideally be in a single microservice. There are other patterns for state synchronization that promote horizontal scalability and temporal decoupling, centered around messaging. Be sure to check out chapter [Link to Come] the section on messaging and integration.

Deployment in a Java EE Environment

This example also uses the Wildfly (from RedHat) application server's [awesome Undertow embedded HTTP server](#) instead of (the default) Apache Tomcat. It's as easy to use Undertow as it is to use Jetty or Tomcat - just exclude `spring-boot-starter-tomcat` and add `spring-boot-starter-undertow!` This contribution originated as a third-party pull request and I've really enjoyed it because it's very fast. You can as easily use Eclipse's Jetty project if you disable Apache Tomcat. To do this, define a dependency for `org.springframework.boot: spring-boot-starter-tomcat` and then set its scope to provided. This will have the effect of ensuring that Tomcat is *not* on the classpath. From there, you need only add `org.springframework.boot: spring-boot-starter-jetty` or `org.springframework.boot: spring-boot-starter-undertow!`

Though this example uses a lot of fairly familiar Java EE APIs, this is still just typical Spring Boot, so by default you can run this application using `java -jar ee.jar` or easily deploy it to process-centric [platforms-as-a-service](#) offerings like Heroku or [Cloud Foundry](#).

If you want to deploy it to a standalone application server like (like Apache Tomcat, or Websphere, or anything in between), it's straightforward to convert the build into a `.war` and deploy it accordingly to any Servlet 3 container. In most cases, it's as simple as making `provided` or otherwise excluding the Spring Boot dependencies that provide Servlet APIs (like `spring-boot-starter-tomcat` or `spring-boot-starter-jetty`) that the Java EE container is going to provide. If you use the [Spring Initializr](#), then you can skip this step by simply choosing `war` from the packaging drop-down.

For Servlet based web applications, you'll need to also add a Servlet initializer class. This is the programmatic equivalent of `web.xml`. Spring Boot provides a base class,

`org.springframework.boot.context.web.SpringBootServletInitializer` that will automatically standup the Spring Boot based auto-configuration and machinery in a standard Servlet 3 fashion. Note that any embedded web-container features - like SSL configuration and port management - will not work when applications are deployed like this.

```
package demo;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInit

public class GreetingServletInitializer extends SpringBootServlet

    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder) {
        return builder.sources(Application.class);
    }
}
```

If you deploy the application to a more classic application server, Spring Boot can take advantage of the application server's facilities, instead. It's dead-simple to consume a JNDI-bound [JMS ConnectionFactory](#), [JDBC DataSource](#) or [JTA UserTransaction](#).

Final Word

We would question a lot of these APIs. Do you **really** need distributed, multi-resource transactions?. JPA's a nice API for talking to a SQL-based `javax.sql.DataSource`, but Spring Data repositories (which include support for JPA, of course, go further and simplify straight JPA considerably. They *also* include support for Cassandra, MongoDB, Redis, Riak, Couchbase, Neo4j, and an increasingly long list of alternative technologies. Repositories are reduced to a single interface definition for the common cases. Do you really need all of this? It might well be that you do, and the choice is yours, but hopefully this is a stopgap on the road to a microservice architecture.

In the context of microservices, Java EE goes against the grain of what a modern microservice architecture needs. Microservices are about small singly focused services with as minimal heft as possible. Some modern application servers can be very tiny, but the principal is the same: why pay for what you don't need? Why go so far to decouple your services from each other only to be satisfied with unneeded coupling of the infrastructure that runs the code?

Application servers were built for a different era (the late 1990s) when RAM was at a premium. The application server promised application isolation and monitoring and consolidated infrastructure, but today it's fairly easy to see it doesn't do a particularly good job of any of that. An application server doesn't protect applications from contention from other applications beyond the class loader. It can't isolate CPU, RAM, filesystems, and even in the JVM itself one component (like the JTA coordinator) can starve the web request processing pool. Instead, these things are provided by the operating system. The operating system cares and knows about processes, not application servers.

It's very common to see applications with the application server configuration (or even the application server itself!) checked into source control systems alongside the application. This (worrisome) approach implies that nobody's trying to squeeze RAM out of their machines by collocating applications on the same application server. Instead, applications need certain

application-specific services and with a Java EE container the path of least resistance is to access those from the container. But that's not the easiest way. Spring applications enjoy security, messaging, distributed transactions, centralized configuration, service resolution, and much more without being coupled to an application server.

Many developers use the application server (or web servers like Apache Tomcat, to which Pivotal is a leading contributor) because it presents a consistent operations burden. Operations know how to deploy, scale and manage the application server. Fair enough. Today, however, the container of choice is something like Docker. Docker exposes a consistent management surface area for operations to work with. What runs inside is entirely opaque.

Docker only cares about processes. There's no assumption that your application will even expose an HTTP endpoint. Indeed, there's often no reason that it should. There's nothing about microservices that implies HTTP and REST, common though it is. By moving away from the application server and moving to `.jar` based deployments, you can use HTTP (and a Servlet container) if and only if it's appropriate. I like to think of it like this: what if the protocol *du jour* was FTP, and not HTTP? What if the whole world accessed services exposed through an FTP server? Does it feel just as architecturally unassailable to deploy our applications to an FTP server? No? Then why do we deploy them to HTTP, EJB, JMS, JNDI and XA/Open servers?

In this chapter we've tried to show that Spring plays well with Java EE services and APIs, and that those APIs can be handy, even in the world of microservices. What we hope you'll take away is that there's no need for the Java EE application server itself. It's just going to slow you down.

Chapter 9. Service Brokers

When developing and running cloud native applications, it can often be the case that a connection to middleware services may be required as a dependency at runtime. In “[Talking to Backing Services](#)”, we talked about how Cloud Foundry applications can talk to backing services using service brokers. In this chapter we will explore in-depth how to use and develop custom service brokers.

Cloud Foundry

When developing cloud native applications that are deployed and operated using a cloud platform, such as Cloud Foundry, there becomes a need to provision middleware services using the platform. The result of building platform services are that developers using the platform are able to take advantage of service offerings as bindings for their application deployments.

An example of a service offering may be something as simple as a Cloud Foundry application that has a dependency on a particular database provider. A cloud platform would allow a developer to provision a database as a service instance. The service instance can then be applied as a binding to their application in a particular runtime environment. In the example of a database as a service, the connection details and credentials are provided to the application at runtime, and are specific to the environment that the application was deployed to.

Services Marketplace

The Cloud Foundry *services marketplace* is a catalog of service offerings that are exposed to users of the platform. The users of the platform can bind their Cloud Foundry applications to service instances. Each service offering in the marketplace is registered as a service broker application. These service brokers are applications that expose a REST API for managing a catalog of service instances, as we talked about in [Chapter 10](#).

Creating Services

Each service in the marketplace provides a set of plans that are used in the creation of a new service instance. Plans are described by service brokers in order to segment the usage requirements of applications that will bind to a service instance. Plans can also be used as a pricing model for marketplace services. For example, for a service that provisions a database instance, a plan may vary the price of running a service instance depending on the amount of resources the database will need. There may be a plan for running the database using a shared VM and a plan that provisions a dedicated VM to host the database.

Binding Applications

When a developer or operator creates a new service instance using the Cloud Foundry CLI, the instance becomes available for applications to bind to it.

Using the Cloud Foundry CLI tool, the command `create-service` is used to create a new service instance.

Example 9-1. Creating a new service using the CF CLI

NAME :

`create-service` - Create a service instance

USAGE :

`cf create-service SERVICE PLAN SERVICE_INSTANCE [-c PARAMETERS]`

We see in [Example 9-1](#) a snippet from the output of the CF CLI for the `cf create-service` command. In this snippet we can see the input parameters that are needed to create a new service instance. The first parameter we will need is the name of the *service* offering from the marketplace catalog. Using the `cf marketplace` command I can get a list of the services that have available plans for my user account.

Using this command on Pivotal's hosted Cloud Foundry instance, called *Pivotal Web Services*, I find that there is a `rediscloud` service in the marketplace with a variety of plans. The `rediscloud` service in the marketplace allows me to create a service instance that will provide me with a Redis database deployment that I can bind my application to. I am able to take a deeper look at each of the service plans for this Redis offering using the command `cf marketplace -s rediscloud`.

Example 9-2. A list of service plans for a Redis deployment

```
$ cf marketplace -s rediscloud
```

```
Getting service plan information for service rediscloud for user.  
OK
```

| service plan | description | free or paid |
|--------------|--------------------|--------------|
| 100mb | Basic | paid |
| 250mb | Mini | paid |
| 500mb | Small | paid |
| 1gb | Standard | paid |
| 2-5gb | Medium | paid |
| 5gb | Large | paid |
| 10gb | X-Large | paid |
| 50gb | XX-Large | paid |
| 30mb | Lifetime Free Tier | free |

In [Example 9-2](#) we can see a list of plans for the `rediscloud` marketplace service. Here we see that we have different paid tiers of resources and a `30mb` plan that is provided as a free tier. Let's go ahead and create a new service instance of Redis that uses this free tier plan of `30mb`.

Example 9-3. Create a new Redis service instance

```
cf create-service rediscloud 30mb my-redis
```

Now that we have created a new service instance, named `my-redis`, we are able to bind an existing application to it. Let's assume here that we have an application named `user-api`, which houses a Spring Boot application that uses Redis for caching users stored in a MySQL database. We can automatically connect to the Redis service instance after binding it to our Spring Boot application. The command to bind to a service instance is `cf bind-service`.

Example 9-4. Binding a service instance to an app

NAME:

`bind-service` - Bind a service instance to an app

USAGE:

```
cf bind-service APP_NAME SERVICE_INSTANCE [-c PARAMETERS_AS_JS]
```

In [Example 9-4](#) we can see the usage details for the `cf bind-service` command. Since we know that the application name we want to bind to is `user-api` and the service instance name is `my-redis`, we can now bind our user application to our newly created Redis database instance.

Example 9-5. Bind my-redis service instance to the user-api app

```
cf bind-service user-api my-redis
```

In [Example 9-5](#) I have created a service binding from my `user-api` application to the `my-redis` instance we created earlier. When binding an application to a service instance in Cloud Foundry, a service broker will provide a set of system environment variables that describe the service instance to the application. We can see the result of this for our `my-redis` binding by looking at the `user-api` environment variables using the `cf env` `user-api` command.

Example 9-6. Display the environment variables for user-api application

```
$ cf env user-api

Getting env variables for app user-api in org user-org...
OK

System-Provided:
{
  "VCAP_SERVICES": { ❶
    "rediscloud": [
      {
        "credentials": { ❷
          "hostname": "pub-redis-17709.us-east-1-3.7.ec2.redislabs.com",
          "password": "GC9hwRzWO20BskMI",
          "port": "17709"
        },
        "label": "rediscloud",
        "name": "my-redis", ❸
        "plan": "30mb", ❹
        "tags": [
          "Data Stores",
          "Data Store",
          "Caching",
          "Messaging and Queuing",
          "key-value",
          "caching",
          "redis"
        ]
      }
    ]
  }
}
```

```
}
```

❶

VCAP_SERVICES describe all service instance bindings for an app

❷

The credentials are provided for the bound service instance

❸

The name of a service instance we created as my-redis from
rediscloud

❹

The plan we chose when creating the my-redis free tier instance

In [Example 9-6](#) we can see the environment variable contents for the VCAP_SERVICES key that are now available after binding user-api to the my-redis service instance. We see here an example of attaching backing services as resources during runtime in a cloud native application. We are adhering to the *twelve-factor application* style of configuration by making a backing service's connection settings available in the environment.

Cloud Foundry Service Brokers

In this section we explored how to create service instances and bindings using the service catalog in Cloud Foundry's marketplace. We also saw how service brokers inject connection details into an application deployment using the system-provided `VCAP_SERVICES` environment variable.

Now that we understand how service brokers are used, let's now explore the mechanics behind the Cloud Foundry platform that will allow us to develop our own custom service brokers using Spring Boot.

Cloud Controller

Cloud Foundry is composed of a number of web service modules that communicate over HTTP — resembling some of the distributed system architectures that we've talked about in this book. The composition of these service modules are exposed as one seamless REST API — and in doing so — becomes the primary interface for clients that trigger the orchestration of an underlying cloud provider's virtual infrastructure. This REST API is called the *Cloud Controller*, and it is the primary contract that describes the functions of the Cloud Foundry platform.

The Cloud Controller also maintains a database for many of the domain resources that are used when interacting with Cloud Foundry — such as orgs, spaces, user roles, service definitions, service instances, apps and more. Since the Cloud Controller is the owner of both service definitions and service instances, it is this very module of Cloud Foundry that allows for the lifecycle management of external service brokers, which can take the form of databases and other middleware services.

Service Broker API

The *Service Broker API* is a consumer-driven REST API contract that describes the expectations of the Cloud Controller for third-party service broker applications. Service brokers are not a part of the Cloud Foundry release. Brokers are only added after a Cloud Foundry release has been provisioned — and are attached as backing services that must be registered with the Cloud Controller API.

Table 9-1. Service broker routes

| Name | Route |
|---------------------------------|---------------------------------------------|
| getCatalog | /v2/catalog |
| deleteServiceInstanceBinding | /v2/service_instances/{instanceId}/service_ |
| createServiceInstanceBinding | /v2/service_instances/{instanceId}/service_ |
| getLastServiceInstanceOperation | /v2/service_instances/{instanceId}/last_ope |
| getServiceInstance | /v2/service_instances/{instanceId} |
| createServiceInstance | /v2/service_instances/{instanceId} |
| deleteServiceInstance | /v2/service_instances/{instanceId} |
| updateServiceInstance | /v2/service_instances/{instanceId} |

In [Table 9-1](#) we see a service broker's REST API contract as HTTP routes and verbs. These methods must be implemented by the service broker using the REST resource representations that are expected to be exchanged with the cloud controller.

Example 9-7. A service catalog that is exposed by an Amazon S3 service broker

```
{  
  "services": [  
    {  
      "id": "00a3b868-9cf9-4ad3-a6b0-e867740cbef0",  
      "name": "amazon-s3",  
      "description": "Amazon S3 simple storage as a backing servi  
      "bindable": true,  
      "plans": [  
        {  
          "id": "ac8fdb55-3223-41e9-a5f5-eca6f8fd40c0",  
          "name": "s3-basic",  
          "description": "Amazon S3 bucket with unlimited storage  
          "free": true  
        }  
      ],  
      "plan_updateable": false,  
      "metadata": {  
        "longDescription": "A backing service with unlimited Amaz  
        "documentationUrl": "http://aws.amazon.com/s3",  
        "providerDisplayName": "Amazon S3",  
        "displayName": "Amazon S3",  
        "supportUrl": "http://aws.amazon.com/s3"  
      }  
    }  
  ]  
}
```

Implementing a Service Broker with Spring Boot

Service brokers can be created using any programming language since the Cloud Controller speaks the language of REST. In this section we'll be taking the lessons learned throughout this book and create a JVM-based service broker using Spring Boot.

Creating service brokers are an essential part of extending the Cloud Foundry platform. There are open source examples available of creating service brokers, which are provided in a variety of different languages. As a part of this book we've provided you with a Spring Boot application that implements an Amazon S3 service broker. Since the code in the template is rather comprehensive, we'll only explore the major pieces of the project that allow it to integrate with Amazon S3.

Amazon S3 Service Broker

There are many use cases that service brokers can solve for. Let's consider a similar service model that is provided through Amazon Web Services. AWS provides a collection of tools that an application can take advantage of when hosted on their platform. An example of this might be a database that is provided as a service, or a log aggregation tool. There are many different options for tooling that are available through AWS, with more services covering more use cases that are being added each year.

The parallel of this, for developing cloud native applications with Cloud Foundry, is that you'll be managing and operating these tools yourself. At first glance that might seem like a lot of extra work, but the benefit is that there is already a large repository of open source service brokers available for you to operate with a distribution of the Cloud Foundry platform.

The example we're going to go over in this chapter is a Spring Boot application that allows developers to provision an Amazon S3 bucket through the platform and bind it to their applications to be used for file storage and retrieval. One of the main tenets of cloud native is that applications should be operated in an ephemeral environment, which means we can't just go ahead and store files on the server that our application is running on. We'll need to attach some form of file storage as a backing service to our applications.

There are two concerns that any Cloud Foundry service broker will need to solve. The first is the functional requirements of integrating with the Cloud Controller. This means that we'll need to create a REST API that implements the expected contract that will allow the Cloud Controller to communicate with the broker.

The second concern we'll be solving for is what we can consider to be the meat of any service broker. The meat I am referring to is the *orchestration of virtual resources*, a workflow that is responsible for provisioning backing services for other applications on the platform to consume. It's important to remember that the service broker itself is just one piece of the puzzle. It's the

provider that takes requests from Cloud Foundry and performs some automation of virtual infrastructure, which is called *orchestration*.

Warning

It's important to understand the different nuances of the term "orchestration" when hearing it in different but similar contexts. In the context of cloud computing, orchestration simply means the necessary infrastructure automation that is required to support the operation of applications. The term is also used in the context of software architecture when referring to the patterns of how a collection of applications will communicate.

The Service Catalog

The first step for implementing a service broker is to create a *service catalog* that describes what your broker will do. The catalog will also provide a collection of service plans that can be used to describe different aspects of how a backing service will be created and operated. Let's take for example a database that is provided as a service through a service broker. As a user of this service broker, we will be provided with a set of plans that describe how we would like our database to be deployed. In this scenario, we might be provided with an option that deploys the database as a single node, or another high availability option that deploys the database as a replicated cluster of multiple nodes.

For our Amazon S3 broker we'll only provide a single plan, which is an S3 bucket with unlimited file storage.

In [Example 9-8](#), we have the basic definition of what we will need for our service catalog. The service broker itself can contain a collection of different service offerings. Our Amazon S3 broker will only contain one service. The `Catalog` object contains a collection of `ServiceDefinition`. Seeing that we do not have other CRUD operations on these objects in this service, we can assume that it is fine to externalize their details as configuration.

Example 9-8. The interface contract for a broker's service catalog

```
public interface CatalogService {  
    Catalog getCatalog();  
    ServiceDefinition getServiceDefinition(String serviceId);  
}
```

To persist the broker's various domain resources, ones such as the `Catalog` and `ServiceDefinition`, we're going to need to connect our application to a database. For the Spring Boot reference application, I've chosen to use Spring Data JPA to connect to a MySQL database. I've also decided to

initialize this database with a set of externalized configuration properties that will be used for the S3 broker's `ServiceCatalog` when the application is started with a `cloud` configuration profile.

Tip

You'll need to decide on an appropriate method to manage the catalog and service definitions of your broker, one that either relies on externalized configuration or the creation of an administrative interface for your broker.

The `ServiceDefinition` is a rather simple model, and we saw it earlier this chapter in the JSON snippet found in [Example 9-7](#). In the reference application you'll find the configuration for these located in the `application.yml`.

Example 9-9. The externalized configuration properties for the Amazon S3 broker

```
broker:
  providerDisplayName: "Amazon S3"
  documentationUrl: "http://aws.amazon.com/s3"
  supportUrl: "http://aws.amazon.com/s3"
  displayName: "Amazon S3"
  longDescription: "A backing service with unlimited Amazon S3 storage"
  imageUrl: "/logo.png"
  basicPlan:
    id: "ac8fdb55-3223-41e9-a5f5-eca6f8fd40c0"
    name: "s3-basic"
    description: "Amazon S3 bucket with unlimited storage"
    free: true
  definition:
    id: "00a3b868-9cf9-4ad3-a6b0-e867740cbef0"
    name: "amazon-s3"
    description: "Amazon S3 simple storage as a backing service"
    bindable: true
```

In the snippet from [Example 9-9](#), you'll find each of the attributes that will be expected by the Cloud Controller in order to make the service broker available on the marketplace. The end result is that these various attributes will be served as an HTTP response from the `/v2/catalog` endpoint of our Spring Boot service broker, and will provide the necessary details for users who will be creating a *service instance* through Cloud Foundry.

Service Instances

Now that we have a service catalog for our Amazon S3 broker, we'll also need to provide a mechanism to store and retrieve *service instances* that are created on the `s3-basic` plan we configured in [Example 9-9](#). A *service instance* is a domain object that tracks the lifecycle of the infrastructure resources that we are orchestrating. In this case it's an AWS IAM (*Identity and Access Management*) user and a corresponding S3 bucket that is only visible to the new user.

When creating a new service instance, we'll need to integrate with AWS using their provided Java SDK to manage the creation of both the IAM user and S3 bucket. This is what I referred to earlier as “the meat of our broker”. After we've successfully created these new resources by integrating with AWS, we'll need to persist their details in our MySQL database, so that we can provide connection information to applications that bind to the new service instance.

Example 9-10. The interface contract for a broker's service instances

```
public interface ServiceInstanceService {  
  
    ServiceInstance createServiceInstance(❶  
        CreateServiceInstanceRequest createService  
        throws ServiceInstanceExistsException, Se  
  
    ServiceInstance getServiceInstance(String serviceInstance  
        throws ServiceInstanceDoesNotExistExcepti  
  
    ServiceInstance deleteServiceInstance(❸  
        DeleteServiceInstanceRequest deleteService  
        throws ServiceBrokerException, ServiceIns  
  
    ServiceInstance updateServiceInstance(❹  
        UpdateServiceInstanceRequest updateService  
        throws ServiceInstanceUpdateNotSupportedException, ServiceBrokerException, ServiceInst  
    }  
}
```

①

Used by the Cloud Controller to create a new service instance

②

Used by the Cloud Controller to get an existing service instance

③

Used by the Cloud Controller to delete a service instance

④

Used by the Cloud Controller for updating a service instance

In [Example 9-10](#) we have the interface contract that defines the methods that will be expected by the Cloud Controller to manage the lifecycle of service instances that are orchestrated by the broker. In the case of our Amazon S3 broker, each method in the interface represents the implementation of a repository pattern that controls the lifecycle of resources on AWS that provide exclusive access to a S3 bucket. The two resources that will be managed by the broker are a new S3 bucket and an IAM user with access to the new bucket.

Example 9-11. Implementation for creating a new S3 service instance

```
@Override  
public ServiceInstance createServiceInstance(  
    CreateServiceInstanceRequest createServiceInstanceRequest  
    throws ServiceInstanceExistsException, ServiceBrokerExcep  
  
    ServiceInstance serviceInstance = new ServiceInstance(createS  
  
    // Check that the service instance does not already exist  
    if (serviceInstanceRepository.exists(serviceInstance.getService  
        throw new ServiceInstanceExistsException(serviceInstance)  
  
    try {  
        // Create a new S3 bucket and a IAM user to manage it  
        S3User user = s3Service.createBucket(  
            serviceInstance.getServiceInstanceId());
```

```

        // Set the credential information to access the S3 bucket
        serviceInstance.setCredential(
            new Credential(user.getCreateUserResult().getUser
                user.getAccessKeyId(), user.getAccessKeyS

            // Save the new service instance
            serviceInstance = serviceInstanceRepository.save(serviceI
        } catch (Exception ex) {
            log.error(ex);
            throw new ServiceBrokerException(ex);
        }

        return serviceInstance;
    }
}

```

In [Example 9-11](#) we'll find the implementation of the `createServiceInstance` method from [Example 9-10](#). Here we have a workflow that will take the `CreateServiceInstanceRequest` and use the details within to create a new `S3User` and S3 bucket using the AWS SDK. Finally, the new `ServiceInstance` is saved to the `ServiceInstanceRepository` containing the credentials that will be used to bind applications to a service instance.

When creating a new service broker for any variety of use cases, the first and most important question we should ask ourselves is *what information will applications need to connect to our service?* Going back again to the example of a service broker for provisioning a database, the information that applications will need to connect to a service instance is the *URL* and the *credentials* of the database.

We can consider that any service instance that is created using the platform is indeed a *backing service* to the application instances that depend on it. The functional result of a backing service is that all necessary information on how to consume it are injected into the application's container as environment variables. This process is what we refer to in Cloud Foundry as *binding to a service instance*. The result of binding to a service instance is that the connection information of how to consume the service is injected into an application's container. When the application is staged in a container and then started, it will be able to locate the environment variables for the service instance's connection information. It will then use these details to locate and connect to the running instance of the service.

For our Amazon S3 service broker, the next step is to implement the functionality that allows applications to be bound to available service instances.

Service Bindings

It can often be a confusing decision to determine the dividing line between offering a function as a service through the platform, or implementing it in each application. There is a good rule of thumb for determining when to delegate function in your application to a backing service through the platform.

If you need to implement the same functionality in all of your applications, that functionality should instead be provided *as a service* through the platform.

The goal of cloud native application development is to reduce the amount of time spent on any development that isn't central to the business logic of your applications. The platform serves us in more than being an operational tool for running applications in the cloud. The platform is a machine that strips away the code that turns an application into legacy and instead provides it as a replaceable service. For example, many applications in an architecture may need the ability to store files. By providing a service broker that can provide applications with a connection to an Amazon S3 bucket, each application will not need to spend the time worrying about implementing custom code to make storing files possible. The only thing that developers of the application will need to do is to bind to a service instance and start storing and retrieving files.

Example 9-12. The interface contract for a broker's service instance bindings

```
public interface ServiceInstanceBindingService {  
    ServiceInstanceBinding createServiceInstanceBinding(❶  
        CreateServiceInstanceBindingRequest createServiceInst  
        throws ServiceInstanceBindingExistsException, Service  
  
    ServiceInstanceBinding deleteServiceInstanceBinding(❷  
        DeleteServiceInstanceBindingRequest deleteServiceInst  
        throws ServiceBrokerException;  
}
```

❶

Used by the Cloud Controller for creating new service instance bindings

❷

Used by the Cloud Controller for deleting existing service instance bindings

In [Example 9-12](#) we have the interface that defines the functions that will be used by the Cloud Controller to create and delete bindings to service instances from applications running on the platform. The `ServiceInstanceBinding` will track an application that has been provided connection details and the credentials to use a `ServiceInstance` on the platform. Each `ServiceInstance` that is created by this broker will be represented as a single bucket that is shared by applications that reference it through the creation of a `ServiceInstanceBinding`.

Tip

It's important to remember that once we've created a `ServiceInstanceBinding` on the broker, the Cloud Controller will become the system of record for the binding, which makes the credentials an immutable object that can only be changed by recreating the binding.

After binding an application to the Amazon S3 service instance, the application will be staged with the credential information required to access file storage APIs for interacting with a bucket.

Example 9-13. An application's environment reveals a binding to an Amazon S3 service instance

```
{  
  "VCAP_SERVICES": {  
    "amazon-s3": [  
      {  
        "credentials": { ❶  
          "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
          "secretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCY",  
          "userName": "6372e1ff-0de5-4b38-bb03-05876e997439" ❷  
        }  
      }  
    ]  
  }  
}
```

```
        },
        "label": "amazon-s3",
        "name": "s3-service",
        "plan": "s3-basic",
        "provider": null,
        "syslog_drain_url": null,
        "tags": []
    }
]
}
}
```

①

The credentials for connecting to the S3 bucket

②

Also the name of the S3 bucket for the IAM user

In [Example 9-13](#) we see the output of the Cloud Foundry CLI command `cf env app-name`. This command will display the environment settings that will be staged within the application's container. Each Cloud Foundry application has a list of `VCAP_SERVICES`, which contains a list of service instance bindings for an application. Here we can see the credentials we need to access an Amazon S3 bucket that was orchestrated when the service instance was created.

Securing the Service Broker

The Cloud Controller will expect that the S3 service broker's API is secured with HTTP basic authentication. Every request that is made from the Cloud Controller to the S3 broker will contain an `Authentication` header that encodes the username and password that the broker must validate.

The Spring Boot application for the S3 broker uses the Spring Security project to implement HTTP basic authentication.

Example 9-14. Configure web security to use HTTP basic authentication and default credentials

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapt

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth
        throws Exception {

        // Configures in-memory authentication for this reference
        auth.inMemoryAuthentication()
            .withUser("admin")
            .password("admin")
            .roles("admin");
    }
}
```

In [Example 9-14](#) we are configuring the S3 broker application to use in-memory authentication with a simple default credential. Spring Security will be configured to use basic authentication by default in a Spring Boot application. In a more sophisticated service broker example you may want to create an administration tool that allows operators of the platform to manage the broker's settings.

Warning

Security is a major consideration when creating service brokers that extend the platform. It's important that access to the broker's APIs are restricted to communication with the Cloud Controller only. Since the Cloud Controller only supports HTTP basic authentication, if the broker were to have its APIs accessible through a public host or route it would be a major security vulnerability that could expose the internals of the underlying infrastructure that is managed by the platform.

Deploying the Service Broker

Now that we've created our Amazon S3 service broker, we can begin testing that it works by deploying it to a distribution of Cloud Foundry called *PCF Dev*. While the service broker can be configured to run on any distribution of Cloud Foundry, PCF Dev is a light-weight distribution of Cloud Foundry that is created by Pivotal and can be installed on your local workstation.

Note

We will use PCF Dev throughout this book for examples that use Cloud Foundry, and it can be downloaded and installed from <http://pivotal.io/pcf-dev>.

After installing PCF Dev on your machine, you will have full access to a local distribution of Cloud Foundry on your workstation. While it is fine to deploy the S3 broker to any distribution of Cloud Foundry, PCF Dev is a good resource for testing custom Spring Boot service brokers when you do not have immediate access to a hosted distribution that is deployed on top of an IaaS provider.

Releasing with BOSH

There are multiple options for deploying custom Cloud Foundry service brokers. A popular option for releasing service brokers is to use a tool named BOSH (*BOSH outer shell*). BOSH is a release engineering tool for managing deployments that can use a variety of different CPIs (*Cloud Provider Interface*) to communicate with an IaaS. A CPI is a executable component of the BOSH architecture that is used as a translation layer to communicate with APIs exposed by an IaaS provider. The CPI takes the proprietary APIs of an IaaS and translates its comparable functions to a common domain language that is used in BOSH to describe complex deployments.

The reason that service brokers frequently are released using BOSH is a common source of confusion for JVM-based developers who are creating brokers using Spring Boot. Why should we use BOSH to release a custom service broker instead of deploying the application with Cloud Foundry itself? After all, isn't the point of developing applications on a platform to be able to use it for deployment? The short answer to this question is that it *depends* on the virtualized compute resources you intend to manage with your custom broker.

If our broker only needs to orchestrate with a single IaaS, it is fine to use the HTTP-based APIs of an IaaS to manage its resources. This is similar to what we've done with our Amazon S3 service broker, which is the reason why we do not need to create a BOSH release. The benefit of using BOSH is that it is a tool that helps to manage service broker releases that need to orchestrate with the IaaS layer that a Cloud Foundry distribution is running on.

The common reason to bundle service brokers in a BOSH release are for those kinds of brokers that need to manage service instances that are composed of large distributed systems. This variety of service broker will need a more fine-grained control of storage devices and network resources provisioned in the IaaS layer. For our S3 broker we do not need to worry about creating a BOSH release because our broker has no need to communicate with the IaaS layer. Because of this fact, we can operate the

broker as a regular application deployment using Cloud Foundry's Cloud Controller API.

Releasing with Cloud Foundry

Since we've decided that our S3 service broker does not need to be bundled as a BOSH release, we can go ahead and deploy it as an application on Cloud Foundry. The first thing we need to do is to describe our application deployment in a Cloud Foundry manifest file.

Example 9-15. The manifest.yml file describes the Amazon S3 service broker deployment

```
---
applications:
- name: s3-broker
  path: ./target/s3-service-broker.jar ❶
  memory: 600M
  instances: 1
  timeout: 180 ❷
  host: s3-broker ❸
  services:
    - s3-broker-db ❹
```

❶

The relative path to the broker artifact compiled as a JAR

❷

How many seconds to wait before the application's first health check

❸

The host name that will be used as a part of the HTTP route to contact the S3 broker

❹

The MySQL service instance to bind to after staging the artifact in its container

The Cloud Foundry deployment manifest in [Example 9-15](#) describes an s3-

broker application that will be deployed and create a service instance binding to a MySQL database named `s3-broker-db`. Before we can deploy the application we'll first need it to be compiled so that the artifact is present in the path described in the manifest. After the artifact is compiled, we can run the Cloud Foundry CLI command `cf push` from the broker's project root. The deployment will be expecting that a service instance called `s3-broker-db` is already available. To create the `s3-broker-db` service instance we need to create a MySQL service from the marketplace.

Create a MySQL Service Instance

From the command line, run the `cf marketplace` to view the available service brokers on PCF Dev.

Example 9-16. The list of available service brokers on PCF Dev

| service | plans | description |
|------------|------------|-----------------------------------------|
| p-mysql | 512mb, 1gb | MySQL databases on demand ① |
| p-rabbitmq | standard | RabbitMQ is a multi-protocol messaging |
| p-redis | shared-vm | Redis service to provide a key-value st |

①

The MySQL service broker we need for the S3 broker application

In [Example 9-16](#) we can see the available service broker plans available on PCF Dev. The service broker we'll need to create a service instance with is named `p-mysql`. Let's now create a service instance using `p-mysql` and a plan of `512mb`.

Example 9-17. Create a new service instance of p-mysql and name it s3-broker-db

```
$ cf create-service p-mysql 512mb s3-broker-db
```

```
Creating service instance s3-broker-db in org pcfdev-org / space
OK
```

In [Example 9-17](#) we run the `create-service` command to create a new service instance of `p-mysql`, which will create a MySQL database for our S3

broker application. Now that we have a MySQL database available named s3-broker-db, we can go ahead and deploy our service broker application to PCF Dev using the Cloud Foundry manifest file from [Example 9-15](#).

Pushing the S3 Broker App

We'll now run the command `cf push` from the S3 broker's project directory, after making sure to first compile the project using the command `mvn clean install`. We can now run the `cf push` command to push the application to PCF Dev. The output from [Example 9-18](#) captures the sequence of steps as the s3-broker artifact is uploaded, staged, and started on PCF Dev.

Example 9-18. Push the S3 broker application to PCF Dev

```
$ cf push

Using manifest file ./service-brokers/s3-service-broker/manifest.

Updating app s3-broker in org pcfdev-org / space pcfdev-space as
OK

Using route s3-broker.local.pcfdev.io
Uploading s3-broker...
Uploading app files from: /var/folders/unzipped-app819946907
Uploading 1019.2K, 163 files
Done uploading
OK

Binding service s3-broker-db to app s3-broker in org pcfdev-org /
space pcfdev-space as admin...
OK

Stopping app s3-broker in org pcfdev-org / space pcfdev-space as
OK

...
0 of 1 instances running, 1 starting
0 of 1 instances running, 1 starting
1 of 1 instances running

App started
```

```
OK

...
state      since                  cpu      memory      disk
#0  running  2016-05-29 09:17:47 AM  0.8%   442.7M of 600M  184.
```

Externalizing the AWS Credentials

Even though the `s3-broker` application has been deployed and is now running, there is another step we need to take care of before we can start creating Amazon S3 service instances. The S3 broker application contains an `application.yml` file that configures the broker to look for environment variables containing the AWS access credentials, as seen in [Example 9-19](#).

Example 9-19. The `application.yml` of the S3 broker that externalizes the AWS API keys

```
aws:
  access-key-id: ${AWS_ACCESS_KEY_ID:replace}
  secret-access-key: ${AWS_SECRET_ACCESS_KEY:replace}
```

If we attempted to create any new service instances on our S3 broker that is currently running, the operation would fail because we did not provide any AWS credentials to the environment. We'll need to login to the AWS console and retrieve valid account credentials of a user that has access to create new Amazon S3 buckets and IAM users. After retrieving these credentials, we'll need to set the access keys as environment variables where the S3 broker application is deployed to on PCF Dev.

Example 9-20. Sets the AWS credentials as environment variables

```
$ cf set-env s3-broker AWS_ACCESS_KEY_ID AKIAIOSFODNN7EXAMPLE
$ cf set-env s3-broker AWS_SECRET_ACCESS_KEY wJalrXUtnFEMI/K7MDEN
```

In [Example 9-20](#) we run two separate commands to set both the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` as environment variables. These environment variables will be sourced by the S3 broker's Spring Boot configuration properties, as we saw in its `application.yml` file from

Example 9-19.

Even though we have successfully set the environment variables on the `s3-broker` app, we will still need to *restage* the app before the environment variables are available to be used.

Note

The term “restage”, or *staging an app*, is a part of a deployment workflow that Cloud Foundry uses to build containers that are capable of running applications for different types of build artifacts. These containers are packaged for applications using task scripts that are executed on an artifact in order to stage the application in a runtime environment using a buildpack.

To restage the `s3-broker` app, run the following command.

```
$ cf restage s3-broker
```

The command will tell Cloud Foundry to restage the application’s container, and in the process, set the environment variables for the AWS credentials. We can now run the command `cf env s3-broker`, as seen in [Example 9-21](#), to confirm that the AWS credentials have been exported as user-provided environment variables on the `s3-broker` application.

Example 9-21. Display the environment variables for the S3 broker app

```
$ cf env s3-broker
```

```
...
```

User-Provided:

```
AWS_ACCESS_KEY_ID: AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY: wJalrXUtnFEMI/K7MDENG/bPxRfiCY
```

Our Amazon S3 service broker is now *fully operational* on the platform and accessible at <http://s3-broker.local.pcfdev.io>. We are now ready to make this service available for our PCF Dev apps by registering and listing the S3 broker on the Cloud Foundry marketplace.

Registering the Amazon S3 Service Broker

Now that our S3 broker application is deployed and running with the provided AWS credentials, we can begin to create new service instances using the CF CLI. Before PCF Dev can recognize the `s3-broker` app as a new service broker, we will need to perform a few steps to register and enable it for usage in the CF marketplace.

There are two steps we need to take care of before users can create service instances using our S3 broker.

- Register the service broker
- Enable service access for a target org

The first step is to use the CF CLI to create the service broker on the Cloud Foundry marketplace. In [Example 9-22](#) we can find the usage instruction for the CF CLI command `cf create-service-broker`.

Example 9-22. Instructions for using the `create-service-broker` command

```
$ cf help create-service-broker

NAME:
  create-service-broker - Create a service broker

USAGE:
  cf create-service-broker SERVICE_BROKER USERNAME PASSWORD URL
```

We'll use this command to issue a request to the Cloud Controller to register and start communicating with the service broker contract that was deployed with the `amazon-s3` app. The Cloud Controller will expect the parameters found in the usage instructions in [Example 9-22](#).

Table 9-2. Parameters to use for registering the S3 broker

| <i>Parameter name</i> | <i>Value</i> |
|-----------------------|--------------|
|-----------------------|--------------|

SERVICE_BROKER s3-broker

USERNAME admin

PASSWORD admin

URL <http://s3-broker.local.pcfdev.io>

The parameters we need to register the service broker using the `create-service-broker` command are listed in [Table 9-2](#). The Cloud Controller will use these parameter values to contact the S3 broker app and attempt to retrieve the service catalog information from <http://s3-broker.local.pcfdev.io/v2/catalog>.

If everything is configured correctly with the `amazon-s3` app that we deployed earlier, the output from [Example 9-23](#) will indicate that CF was able to contact the broker and download its service catalog information.

Example 9-23. Register the s3-broker app with the CF cloud controller

```
$ cf create-service-broker s3-broker admin admin http://s3-broker  
Creating service broker s3-broker as admin...  
OK
```

Tip

The CF CLI is a client application written in Go, and it implements a variety of workflows for each of command you run. One of the most helpful features to understand what the CLI is doing is to turn on its tracing feature. By exporting `CF_TRACE=true` as an environment variable, you'll be able to see the sequence of HTTP request/response interactions with the Cloud Controller.

Enabling Service Access

After successfully registering the S3 service broker with the Cloud Controller, it will not be available to any users of the platform until an administrator has enabled it using the `cf enable-service-access` command. We can run `cf help enable-service-access` command to retrieve the usage details for enabling access to the registered `amazon-s3` broker. The usage details are shown in [Example 9-24](#).

Example 9-24. The usage details for the enable-service-access command

```
$ cf help enable-service-access

NAME:
  enable-service-access - Enable access to a service or service

USAGE:
  cf enable-service-access SERVICE [-p PLAN] [-o ORG]

OPTIONS:
  -p    Enable access to a specified service plan
  -o    Enable access for a specified organization
```

As you can see in [Example 9-24](#), there are multiple options we can use to control how our service broker is accessed on the platform. This is a helpful feature for making your service broker plans only available to certain orgs. For example, you may have added a new plan to your broker release and it is not yet ready for general availability. You can choose to only enable its access to certain orgs and prevent apps in production from consuming service instances provided by the new plan.

In [Example 9-25](#) we are choosing to only enable service access to our `amazon-s3` broker for the `s3-basic` plan only in the org named `pcfdev-org`.

Example 9-25. Enable the S3 broker for access on the CF marketplace

```
$ cf enable-service-access amazon-s3 -p s3-basic -o pcfdev-org

Enabling access to plan s3-basic of service amazon-s3 for org pcf
OK
```

Now that marketplace access to the `s3-basic` plan on the `amazon-s3` service catalog has been enabled, we can confirm it is now visible in the marketplace in spaces of `pcfdev-org`.

Example 9-26. Confirm that the `amazon-s3` service is available on the CF marketplace

```
$ cf marketplace

Getting services from marketplace in org pcfdev-org / space pcfde
OK

service      plans          description
amazon-s3    s3-basic       Amazon S3 simple storage as a backing s
p-mysql      512mb, 1gb    MySQL databases on demand
p-rabbitmq   standard      RabbitMQ is a robust multi-protocol mes
p-redis      shared-vm     Redis service to provide a key-value st
```

①

The Amazon S3 broker is now ready to be used to create service instances

In [Example 9-26](#) we can now see that the `amazon-s3` service is listed with the `s3-basic` plan. We can also see that the `amazon-s3` service definition is visible, which we configured as a property in the `application.yml` file of the `s3-broker` app.

Note

Spring Boot actuator endpoints are available on the `s3-broker` app at <http://s3-broker.local.pcfdev.io/management>. To understand how the S3 broker application has been configured for the `cloud` profile, it's helpful to introspect the configuration properties listed at <http://s3-broker.local.pcfdev.io/management/env>.

Creating Amazon S3 Service Instances

Now that our S3 broker is running and is now registered as a service on PCF Dev's marketplace, we can begin to create service instances to create new S3

storage buckets.

Example 9-27. Create the first service instance using the new S3 broker

```
$ cf create-service amazon-s3 s3-basic s3-service  
Creating service instance s3-service in org pcfdev-org / space pc  
OK
```

If the AWS credentials that we set as environment variables on the `s3-broker` application are correct, the command shown in [Example 9-27](#) will succeed. If the command is completes without error, that means that a new Amazon S3 bucket and corresponding IAM user were created on the AWS account. Before we can see what these resources are, we'll need to create a service binding for a new application deployment that will consume the new Amazon S3 service instance.

Tip

If you see a failure after attempting to create a new `amazon-s3` service instance, it is because the provided AWS credentials were not valid. To fix the issue, make sure that the user with the credentials has the correct resource policies attached for administrating IAM users and managing Amazon S3 buckets. You can configure the policies on the AWS console by navigating to the *Identity and Access Management* tool.

Consuming Service Instances

When it comes to extending the platform with new services, there should also be a focus on providing developers with an easy way to consume those services in a cloud native application. While a major part of platform engineering is to build services, it is equally important to make those services as easy to consume as possible. To do this effectively we need to consider the different application runtimes of workloads that will be consuming our new service.

Earlier we talked about the dividing line between implementing functionality in an application versus offering that functionality as a service. The rule of thumb being: that if you need to implement the same functionality in every application, it should instead be provided a service. We need to take this same view when it comes to writing client libraries to consume new services. For a majority of scenarios for integrating with service instances, there will be an existing client library or SDK available to be used in applications. This will be the case for consuming our Amazon S3 service instances, where we'll be using the AWS Java SDK to interact with Amazon S3's storage APIs.

Let's also consider a scenario where a client library does not exist. One such example could be user authentication. User authentication is a functionality that needs to be supported in mostly all applications that need security. In this scenario, if we did not provide a client library for each application, then each application would need to create that functionality to consume a service.

It's also important to make sure that certain details like application security are implemented in the same way across multiple applications. This precaution prevents duplication of efforts that could lead to similar defects in separate applications. Divergences in different but similar implementations of client-side authorization may cause multiple but similar security vulnerabilities across an application architecture. If a vulnerability were to be exposed in one application, then depending on the risk imposed by that security flaw, there may be a need to extensively examine the multiple but similar authorization implementations of all applications. By providing a

client library to our cloud native applications, we gain the benefit of only having to patch a security vulnerability once. This practice also decreases the exposure window of security flaws by applying the patch as an automatic upgrade across all affected applications that are running on the platform.

When developing cloud native JVM applications to use custom services on Cloud Foundry, platform engineers can take an opinionated view on how application developers will consume these services. By taking an opinionated view on how platform services are to be consumed, all developers will automatically conform to certain implementation practices for service consumption by making integration an extended feature of the application framework.

There is an added benefit to being an opinionated platform engineer, which is that you prevent architecture review boards from having to police the source code of applications in order to determine the level of compliance with the prescribed standards. We can automate a majority of the compliance in applications by being opinionated about how an application framework integrates with the platform. With Spring Boot we can take the same opinionated approach as an application framework developer by providing Spring Boot starter projects that automatically consume the service instances that we provide by extending Cloud Foundry.

Extending Spring Boot

Spring Boot can be extended to automatically integrate with service instances on the Cloud Foundry platform. In this section we'll be creating a web-based file browser for uploading and viewing files in an Amazon S3 bucket. For this simple web application we'll bind to a service instance of our new `amazon-s3` broker. In order to make the Amazon S3 service instance as easy to consume as possible, we'll create a custom Spring Boot starter project that auto-configures an Amazon S3 template.

Creating an Amazon S3 Browser

The Spring Boot application we will create in this section is seen in [Figure 9-1](#). This single page application will allow users to upload and browse files that are stored in an Amazon S3 bucket.

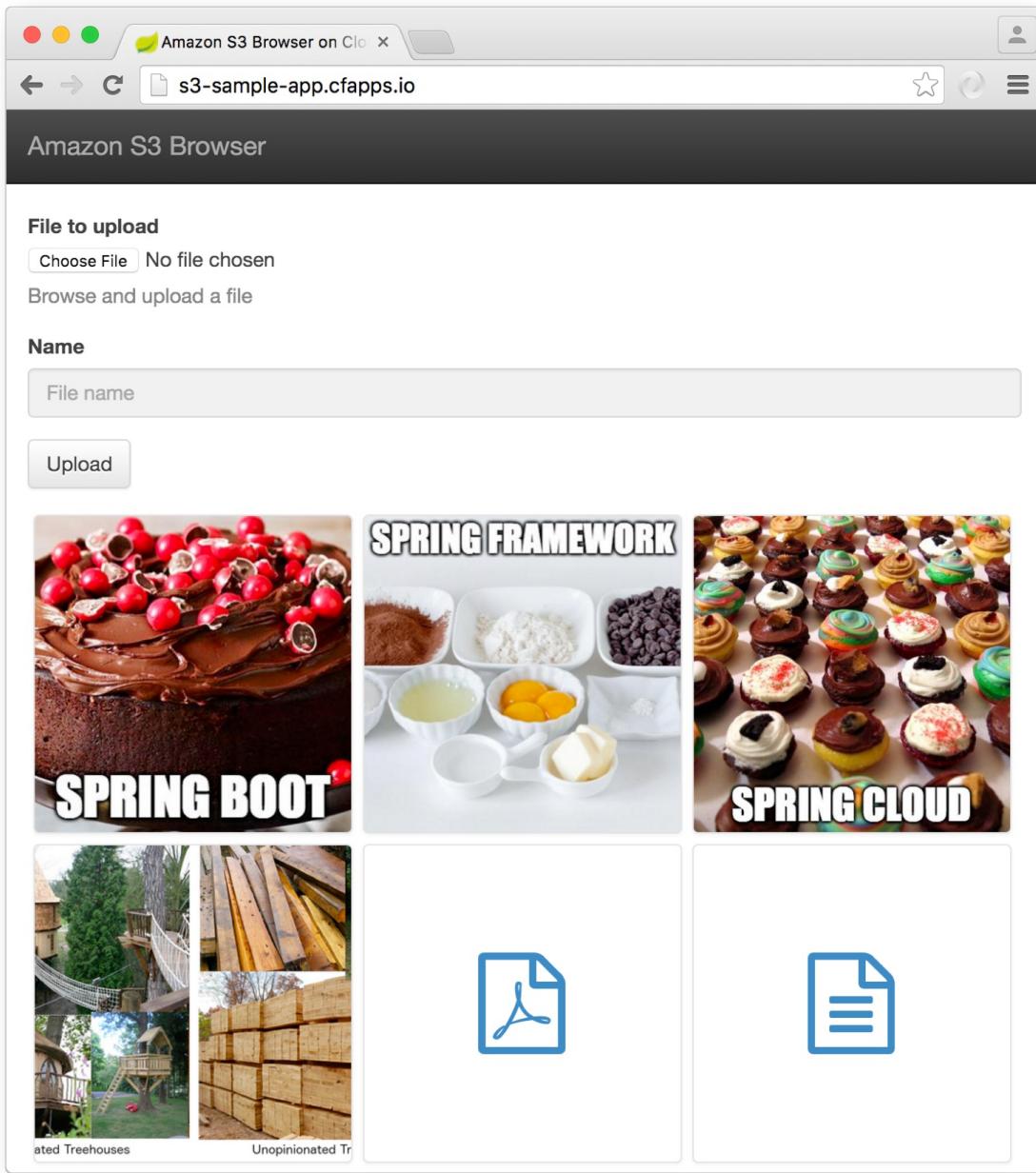


Figure 9-1. The web-based Amazon S3 browser we'll create using Spring Boot

The `spring-boot-amazon-s3-sample` is a Spring Boot application that allows users to upload files to an auto-configured Amazon S3 bucket.

Custom Starter Projects

Early on in this book we talked about how Spring Boot starter projects work. The Spring Boot starter projects that are available from Spring Initializr (<http://start.spring.io>) are a set of auto-configured dependencies that can be included in your Spring Boot project. While these starter projects are published and maintained by the Spring Engineering team for each Spring Boot release, there are many scenarios where creating your own custom starter projects will make it easier for developers to consume services offered through Cloud Foundry's marketplace.

In this section we're going to create a client library that integrates with our Amazon S3 service broker we created earlier. The client library will include a set of custom configuration properties for connecting to Amazon S3. Depending on whether or not these custom properties are set in the `application.properties` of a Spring Boot application that references our new client library, an `AmazonS3Template` bean will be auto-configured with the configuration values.

Example 9-28. The client library includes a configurable `AmazonS3Template` bean

```
/**
 * This class is a client for interacting with Amazon S3 bucket r
 *
 * @author kbastani
 */
@Component
public class AmazonS3Template {

    private String defaultBucket;
    private String accessKeyId;
    private String accessKeySecret;
    private Credentials sessionCredentials;

    /**
     * Create a new instance of the {@link AmazonS3Template} with
     * name and access credentials
     *
     * @param defaultBucket is the name of a default bucket fro
     * @param accessKeyId is the access key id credential for
     * @param accessKeySecret is the access key secret for the sp
     */
    public AmazonS3Template(String defaultBucket, String accessKe
                           String accessKeySecret) {
```

```

        this.defaultBucket = defaultBucket;
        this.accessKeyId = accessKeyId;
        this.accessKeySecret = accessKeySecret;
    }
    ...
}

```

In [Example 9-28](#) we can see the partial definition of an `AmazonS3Template` bean that we will distribute with our starter project. This is a simple client library that will wrap around the AWS Java SDK that is distributed by Amazon. We'll do this to auto-configure the `AmazonS3Template` and use the custom configuration properties that will contain the credential information to connect to an Amazon S3 bucket. Those credentials can be seen in [Example 9-28](#), in the constructor's required parameters.

Example 9-29. Custom configuration properties class for Amazon S3 credentials

```

/**
 * Configuration property group for Amazon S3 and AWS
 *
 * @author kbastani
 */
@Configuration
@ConfigurationProperties(prefix = "amazon")
public class AmazonProperties {

    @NestedConfigurationProperty
    private Aws aws;

    @NestedConfigurationProperty
    private S3 s3;

    ...
}

```

In [Example 9-29](#) we have a partial definition of a configuration properties class called `AmazonProperties`. This configuration class will include a set of nested configuration property classes that will be mapped to a namespace of properties in the `application.properties` files that we use to configure Spring Boot applications. In this example we see that there is a parameter to the `@ConfigurationProperties` annotation that sets `prefix` to the value `amazon`. By providing a prefix for our configuration properties, we are mapping a root namespace that will be used to group together nested properties of this class. These hierarchy of property classes will generate keys

in an `application.properties` file we can use to map external configuration properties to the configuration class used for auto-configuration.

Example 9-30. Auto-configuration class for an Amazon S3 template

```
/*
 * This class auto-configures a {@link AmazonS3Template} bean.
 *
 * @author kbastani
 */
@Configuration
@ConditionalOnMissingBean(AmazonS3Template.class)
@EnableConfigurationProperties(AmazonProperties.class)
public class S3AutoConfiguration {

    @Autowired
    private AmazonProperties amazonProperties;

    @Bean
    AmazonS3Template amazonS3Template() {
        return new AmazonS3Template(amazonProperties.getS3().getD(
            amazonProperties.getAws().getAccessKeyId(),
            amazonProperties.getAws().getAccessKeySecret()));
    }
}
```

The next step will be to include a configuration class that will be triggered if an instance of `AmazonS3Template` is not found in the current Spring context's container. In [Example 9-30](#) we see an `S3AutoConfiguration` class that does a simple function. This class will bind the custom configuration properties from `AmazonProperties` and use its property values to create a new instance of the `AmazonS3Template` and add it to the Spring container. This process is the essence of what we refer to when we talk about Spring Boot auto-configuration.

Summarizing auto-configuration

Auto-configuration in Spring Boot has three simple ingredients.

- Bean definitions

- Configuration properties
- Auto-configuration classes

Bean definitions

The first ingredient in this list is a collection of custom bean definitions. These beans are typically client APIs that can be auto-configured using a set of external configuration properties. An example of this for our Amazon S3 starter project is the `AmazonS3Template`. We created this bean definition to be auto-configured using a set of external properties.

Configuration properties

The second ingredient is a set of custom configuration property classes. These property classes will be used as the configurable inputs to create and register new instances of our beans for the Spring context. We'll register these configuration property classes and inject their values into an auto-configuration class that will be used to register new bean definitions. The values of these property classes will be sourced directly from the configuration profile that contains a key-value map that is attached to a Spring Boot application at runtime. These key-values can be found in the `application.properties` file or `application.yml`, for example.

Auto-configuration classes

The last ingredient we'll need is the actual auto-configuration class that is triggered using a condition. There are many different conditions that can be used to trigger the creation of a new bean for the Spring context's container. The most common auto-configuration condition is called

`@ConditionalOnMissingBean`. This annotation can be placed on a configuration class and will only be registered by the Spring context if a set of specified beans are not found within the context's container. This also allows us to override the auto-configuration by creating our own bean definitions in our Spring Boot application, and by doing so we avoid the

creation of multiple beans of the same class type.

Creating a starter project

Starter projects are simple dependency wrapper projects that are used to include auto-configuration classes on the classpath of a Spring Boot application.

Example 9-31. The Amazon S3 starter project's pom.xml

```
<artifactId>spring-boot-starter-amazon-s3</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
...
<dependencies>
    <dependency>
        <groupId>com.example</groupId>
        <artifactId>spring-boot-autoconfigure-amazon-s3</artifact
        <version>0.0.1-SNAPSHOT</version>
    </dependency>
    ...
</dependencies>
```

In [Example 9-31](#) we have a simple Maven build definition that will be used to distribute the auto-configuration classes for the `AmazonS3Template`. While it is possible to simply include the auto-configuration dependency directly on a Spring Boot application, it may become the case that you'll include multiple auto-configuration projects for a starter project. A starter project is just a parent project that includes a set of auto-configuration libraries that are bundled into a clean dependency reference in your Spring Boot applications.

Creating an Amazon S3 Browser

Now that we have created a custom starter project for interacting with Amazon S3 storage APIs, we can start using it in a new Spring Boot application.

Example 9-32. Include the Amazon S3 starter project in a new Spring Boot

application

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>spring-boot-starter-amazon-s3</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

In [Example 9-32](#) we see the dependency information that we'll include in a new Spring Boot application's `pom.xml`. By including this starter project, we'll be able to use Spring Boot configuration properties to automatically configure a new instance of the `AmazonS3Template` bean.

The only thing we need to do is to map configuration keys to values in the `application.yml` file in the `resources` directory of our new Spring Boot application.

Example 9-33. Configuring a Spring Boot application to use the Amazon S3 service instance binding

```
spring.profiles.active: development
---
spring:
  profiles: cloud
  application:
    name: ${vcap.application.name:s3-sample-app}
amazon:
  aws:
    access-key-id: ${vcap.services.s3-service.credentials.accessKey}
    access-key-secret: ${vcap.services.s3-service.credentials.secretKey}
    s3:
      default-bucket: ${vcap.services.s3-service.credentials.bucketName}
  multipart.maxFileSize: 5MB
  multipart.maxRequestSize: 5MB
---
spring:
  profiles: development
  application:
    name: s3-sample-app
amazon:
  aws:
    access-key-id: replace
    access-key-secret: replace
```

```
s3:  
  default-bucket: replace  
  multipart.maxFileSize: 5MB  
  multipart.maxRequestSize: 5MB
```

In [Example 9-33](#) we see two configuration profiles in the `application.yml` file for the new Spring Boot application. The first of the two profiles is the `cloud` profile. By default, when we push a Spring Boot application to Cloud Foundry the active profile will be set to `cloud`. In this `cloud` profile we see the configuration keys that are mapped by the configuration properties class `AmazonS3Properties` seen in [Example 9-29](#). The values that are mapped to this configuration properties are being sourced from the environment in Cloud Foundry. The namespace for environment properties always starts with `vcap`, as seen in the example.

This new application will be bound to a service instance called `s3-service`, which we created earlier from our new Amazon S3 broker on Cloud Foundry.

Example 9-34. The manifest file for deploying the S3 browser to Cloud Foundry

```
---  
applications:  
- name: s3-sample-app  
  path: ./target/spring-boot-amazon-s3-sample.jar  
  host: s3-sample-app  
  services:  
    - s3-service ①
```

①

The service instance that provides credentials to use Amazon S3

In [Example 9-34](#) we have a `manifest.yml` file that describes the Cloud Foundry deployment for the Amazon S3 browser application we're building. Notice that the `s3-service` is specified as a service binding for the application deployment. Because we created a new S3 service instance called `s3-service` earlier in the chapter, this application will automatically bind to it when it is deployed.

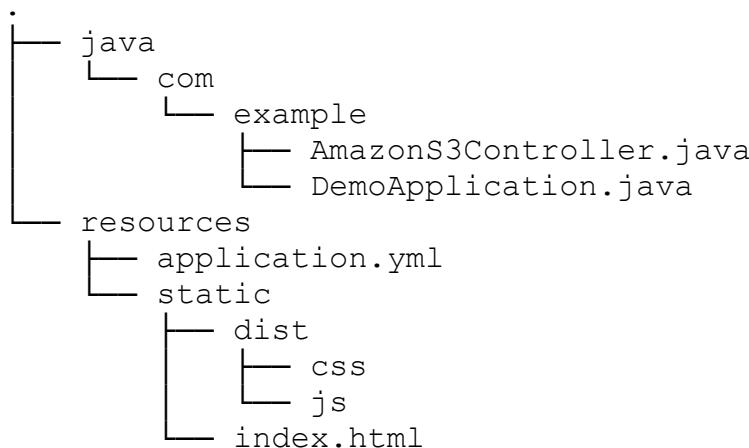
When a Spring Boot application is bound to a service instance, it is then able

to retrieve configuration property values by referencing the credentials values from the key `vcap.services.service-name.credentials`. This is what we've done in the `cloud` profile of our new Spring Boot application.

Building the S3 Browser

Included in the source code of this book for this chapter, you'll find the `spring-boot-amazon-s3-sample` project which includes the source code for the S3 browser. This project is a simple web application that includes a set of static web resources in the `/src/main/resources/static` directory. These static files include the client-side HTML and JavaScript files that interact with a REST API with request mappings defined in an `AmazonS3Controller`.

Example 9-35. The Amazon S3 sample's directory structure



In the directory tree found in [Example 9-35](#), you'll see the basic structure of the sample application that we'll use to browse and upload files to an Amazon S3 bucket.

Example 9-36. Creates a controller to interact with Amazon S3 resources

```
@RestController
@RequestMapping("/s3")
public class AmazonS3Controller {

    private AmazonS3Template amazonS3Template;
```

```

@Value("${amazon.s3.default-bucket}")
private String bucketName;

@Autowired
public AmazonS3Controller(AmazonS3Template amazonS3Template)
    // Get a reference to the S3 template bean
    this.amazonS3Template = amazonS3Template;
}

@RequestMapping(method = RequestMethod.GET, path="/resources")
public List<Resource<S3ObjectSummary>> getBucketResources() {
    // List objects using the S3 template
    ObjectListing objectListing = amazonS3Template.getAmazonS
        .listObjects(new ListObjectsRequest()
            .withBucketName(bucketName));
    ...
}

@RequestMapping(method = RequestMethod.POST, value = "/resour
public @ResponseBody Object handleFileUpload(
    @RequestParam("name") String name,
    @RequestParam("file") MultipartFile f
    // Upload the file using the S3 template
    amazonS3Template.getAmazonS3Client().putObject(...);
    ...
}
}

```

In [Example 9-36](#) we see the partial definition of the `AmazonS3Controller` class that is a part of the `spring-boot-amazon-s3-sample` project. Here we will be able to use the `AmazonS3Template` bean that was auto-configured using the configuration properties provided by the Cloud Foundry service instance we provisioned using the Amazon S3 broker. The controller has two request mappings, one for uploading files and one for browsing files in the S3 bucket at the endpoint `/s3/resources`.

Deploying the S3 Browser to Cloud Foundry

Using the same Cloud Foundry instance you have installed the Amazon S3 broker to, compile and then push the S3 sample application. To compile the application, run the command `mvn clean install`. This will compile and place the build artifact in the `/target` directory of the project. Since the

`manifest.yml` has been included with the project, all that is left to do is to run `cf push` from the root of the `spring-boot-amazon-s3-sample` project.

The application will then be deployed and started on your Cloud Foundry instance. After binding to the `s3-service` instance that we created earlier, the new application will be staged with the credentials that were created to access an S3 bucket.

Navigate to the S3 browser application at <http://s3-sample-app.local.pcfdev.io>, which is the URL if you're using PCF Dev. You'll be able to upload new files to your S3 bucket. After uploading files, you can navigate to `/s3/resources` to see a collection of S3 objects that have been uploaded to the default bucket.

Example 9-37. Retrieving the list of objects that are in the application's S3 bucket

```
[  
  {  
    bucketName:"457f6a37-cbbc-4958-86dc-0821d01deb92",  
    key:"profile-photo.jpg",  
    size:148527,  
    lastModified:1455963592000,  
    storageClass:"STANDARD",  
    owner:null,  
    etag:"76e545dbfaa2b05754f21007d76a6f1a",  
    links:[  
      {  
        rel:"url",  
        href:"https://s3.amazonaws.com/.../profile-photo.jpg"  
      }  
    ]  
  },  
  {  
    bucketName:"457f6a37-cbbc-4958-86dc-0821d01deb92",  
    key:"twitter-discovery.png",  
    size:398233,  
    lastModified:1455963182000,  
    storageClass:"STANDARD",  
    owner:null,  
    etag:"a51d7a07b7aa959f664e5f0b8d53a1ea",  
    links:[  
      {  
        rel:"url",  
        href:"https://s3.amazonaws.com/.../twitter-discovery.png"  
      }  
    ]  
  }]
```

```
        href:"https://s3.amazonaws.com/.../twitter-discovery.  
    }  
]  
}  
]
```

Summary

In this chapter we've explored how to extend Cloud Foundry and Spring Boot to deliver a fully managed service, one that has an auto-configured starter project that can be used by development teams that are building cloud-native applications.

Chapter 10. The Forklifted Application

So you've got that shiny new distributed runtime, infinite greenfield potential and lots of existing applications, now what?

The Contract

Cloud Foundry aims to improve velocity by reducing or at least making consistent the operational concerns associated with deploying and managing applications. Cloud Foundry is an ideal place to run online web-based services and applications, service integrations and back-office type processing.

[Cloud Foundry](#) optimizes for the continuous delivery of web applications and services by making assumptions about the shape of the applications it runs. The *inputs* into Cloud Foundry are applications - Java .jar binaries, Ruby on Rails applications, Node.js applications, etc. - Cloud Foundry provides well-known operational benefits (log aggregation, routing, self-healing, dynamic scale-up and scale-down, security, etc.) to the applications it runs. There is an implied contract between the platform and the applications that it runs. This contract allows the platform to keep promises to the applications it runs.

Some applications may never be able to meet that contract. Other applications might be able to, albeit with some soft-touch adjustments. In this chapter, we'll look at possible soft-touch refactorings to coerce legacy applications to run on Cloud Foundry.

The goal isn't, in this case, to build an application that's *native* to the cloud. It's to move existing workloads to the cloud to reduce the operational surface area, to increase uniformity. Once an application is deployed on Cloud Foundry it is at least as well off as it was before and now you have one less snowflake deployment to worry about. Less is more.

I distinguish this type of workload migration - *application forklifting* - from building a *cloud native application*. Much of what [we talk](#) about these days is about building *cloud native applications* - applications that live and breathe in the cloud (they inhale and exhale as demand and capacity require) and that fully exploit the platform. That journey, while ideal and worth taking on assuming the reward on investment is tangible, is a much longer larger discussion and not the focus of this chapter (but it *definitely* is the focus of

the *other* chapters!)

Application behavior is, broadly speaking, the sum of its environment and code. In this chapter, we'll look at strategies for moving a legacy Java application from some of the environments that legacy Java applications typically live in. We'll look at patterns typical of applications developed before the arrival of cloud-computing and then we'll look at some specific solutions and accompanying code.

Migrating Application Environments

There are some qualities that are common to all applications, and those qualities - like RAM and DNS routing - are configurable directly through the Cloud Foundry `cf` CLI tool, various dashboards, or in an application's `manifest.yml` file. If your application is a compliant application that just needs more RAM or a custom DNS route, then you'll have everything you need in the basic tools

the Out-of-the-Box Buildpacks

Things sometimes just aren't that simple, though. Your application may run in any number of snowflake environments, whereas Cloud Foundry makes very explicit assumptions about the environments its applications run in. These assumptions are encoded to some extent in the platform itself and in *buildpacks*. Buildpacks were adopted from Heroku. Cloud Foundry and Heroku don't really care what kind of application they are running. They care about Linux containers, which are ultimately operating system processes. Buildpacks tell Cloud Foundry what to do given a Java `.jar`, a Rails application, a Java `.war`, a Node.js application, etc. A buildpack is a set of callbacks - shell scripts that respond to well-known calls - that the runtime will use to ultimately create a Linux container to be run. This process is called *staging*.

Cloud Foundry provides [many out-of-the-box system buildpacks](#). Those buildpacks can be customized or even completely replaced. Indeed, if you want to run an application for which there is no existing buildpack provided out of the box ([by the Cloud Foundry community, Heroku or Pivotal](#)) then at least it's easy enough [to develop and deploy your own](#). There are buildpacks for all manner of environments and applications out there, including one called [*Sourcey*](#) that simply compiles native code for you!

Customizing Buildpacks

These buildpacks are meant to provide sensible defaults while remaining adaptable. As an example, [the default Java/JVM buildpack] (<https://github.com/cloudfoundry/java-buildpack>) supports `war`'s (which will run inside of an up-to-date version of Apache Tomcat), Spring Boot-style executable `jar`'s, Play web framework applications, Grails applications, and much more.

If the system buildpacks don't work for you and you want to use something different, you only need to tell Cloud Foundry where to find the code for the buildpack using the `-b` argument to `cf push`:

```
cf push -b https://github.com/a/custom-buildpack.git#my-branch c
```

Alternatively, you can specify the buildpack in the `manifest.yml` file that accompanies your application. As an example, suppose we have a Java EE application that has historically been deployed using Websphere. IBM maintains a very capable WebSphere Liberty buildpack. To demonstrate this, let's look at a basic Java EE-only Servlet.

```
package demo;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/hi")
public class DemoApplication extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws Serv

        response.setContentType("text/html");

        response.getWriter().print(
            "<html><body><h3>Hello Cloud</h3>
```

```
    }
}
```

To run this, I've specified the WebSphere Liberty buildpack in the application's manifest.

```
---
applications:
- name: wsl-demo
  memory: 1024M
  buildpack: https://github.com/cloudfoundry/ibm-websphere-liberty
  instances: 1
  host: wsl-demo-$(random-word)
  path: target/buildpacks.war
  env:
    SPRING_PROFILES_ACTIVE: cloud
    DEBUG: "true"
    debug: "true"
    IBM_JVM_LICENSE: L-JWOD-9SYNCP
    IBM_LIBERTY_LICENSE: L-MCAO-9SYMVC
```

Some buildpacks lend themselves to customization. The [Java buildpack](#) - which was originally developed by the folks at Heroku and which people working on Cloud Foundry have since expanded - supports configuration through environment variables. The Java buildpack provides default configuration in the `config` directory for various aspects of the buildpack's behavior. You can override the behavior described in a given configuration file by providing an environment variable (prefixed with `JBP_CONFIG_`) of the same name as the configuration file, sans the `.yml` extension. Thus, borrowing an example from the excellent [documentation](#), if I wanted to override the JRE version and the memory configuration (which lives in the `config/open_jdk_jre.yml` file in the buildpack), I might do the following:

```
cf set-env custom-app JBP_CONFIG_OPEN_JDK_JRE \
' [jre: {version: 1.7.0_+}, memory_calculator: {memory_heuristics:
```

Containerized Applications

Applications in the Java world that were developed for a J2EE / Java EE application server tend to be very *sticky* and hostile to migration outside of that application server. Java EE applications - for all their vaunted portability - use class loaders that behave inconsistently, offer different subsystems that themselves often require proprietary configuration files and - to fill in the many gaps - they often offer application server-specific APIs. If your application is completely intractable and these various knobs and levers we've looked at so far don't afford you enough runway to make the jump, there may still be hope yet! Be sure to look through the community buildpacks. There are buildpacks that stand up IBM's WebSphere (with contributions from IBM, since they have a PaaS based on Cloud Foundry!) and RedHat's WildFly, as well, for example.

Cloud Foundry "Diego" also supports running containerized (Docker, with other containers to come) applications. This might be an alternative if you've already got an application containerized and just want to deploy and manage it with the same toolchain as any other application. We've extracted some of the interesting scheduling and container-aware features of the forthcoming Cloud Foundry into a separate technology called Lattice. Lattice is Cloud Foundry by subtraction. If nothing else, you can use it to containerize and validate your existing application. We've even put together some nice guides [on containerizing your Spring applications](#) and [then running them on Lattice!](#)

We've run the gamut from common-place configuration, to application- and runtime-specific buildpack overrides to opaque containerized applications. I start any attempts to forklift an application in this order, with simpler tweaks first. The goal is to do as little as possible and let Cloud Foundry do as much as possible.

Soft-Touch Refactoring to get your application into the cloud

In the last section we looked at things that you can do to wholesale move an application from it's existing environment into a new one without modifying the code. We looked at techniques for moving simple applications that have fairly common requirements all the way to very exotic requirements. We saw that there are ways to all but virtualize applications and move them to Cloud Foundry, but we didn't look at how to point applications to the backing services (databases, message queues, etc.) that they consume. We also ignored, for simplicity, that there are some classes of applications that could be made to work cleanly on Cloud Foundry with some minor, tedious, and feasible changes.

It always pays off to have a comprehensive test suite in place to act as a harness against regressions when refactoring code. I understand that - due to their very nature - some legacy applications won't have such a test suite in place.

We'll look mostly at *soft-touch* adjustments that you could make to get your application working, hopefully with a minimum of risk. It goes without saying, however, that - absent a test suite - more modular code will isolate and absorb change more readily. It's a bitter irony then that the applications most in need of a comprehensive test-suite are the ones that probably don't have it: large, monolithic, legacy applications. If you *do* have a test suite in place, you may not have smoke tests that validate connectivity and deployment of the application and its associated services. Such a suite of tests is necessarily harder to write but would be helpful precisely when undertaking something like forklifting a legacy application into a new environment.

Talking to Backing Services

A backing service is a service (databases, message queues, email services, etc.) that an application consumes. Cloud Foundry applications consume backing services by looking for their locators and credentials in an environment variable called `VCAP_SERVICES`. The simplicity of this approach is a feature: any language can pluck the environment variable out of the environment and parse the embedded JSON to extract things like service hosts, ports, and credentials.

Applications that depend on Cloud Foundry-managed backing services can tell Cloud Foundry to create that service on-demand. Service creation could also be called *provisioning*. Its exact meaning varies depending on context; for an email service it might mean provisioning a new email username and password. For a MongoDB backing service it might mean creating a new Mongo database and assigning access to that MongoDB instance. The backing service's lifecycle is modeled by a Cloud Foundry service broker instance. Cloud Foundry service brokers are REST APIs that Cloud Foundry cooperates with to manage backing services.

Once the broker is registered with Cloud Foundry, it is available through the `cf marketplace` command and can be provisioned on demand using the `cf create-service` command. This service is ready to be consumed by one or more applications. At this point the service is a logical construct with a logical name that can be used to refer to it.

Here's a hypothetical service creation example. The first parameter, `mongo`, is the name of the service. I'm using something generic here but it could as easily have been New Relic, or MongoHub, or ElephantSQL, or SendGrid, etc. The second parameter is the plan name - the level and quality of service expected from the service provider. Sometimes higher levels of service imply higher prices. The third parameter is the aforementioned logical name.

```
cf create-service mongo free my-mongo
```

It's not hard to create a service broker, but it might be more work than you

need. If your application wants to talk to an existing, static service that isn't likely to move and you just want to point your application to it, then you can use [user provided services](#). A user-provided service is a fancy way of saying "take this connection information and assign a logical name to it and make it something I can treat like any other managed backing service."

A backing service - created using the `cf create-service` command or as a user-provided service - is invisible to any consuming applications until it is *bound* to an application; this adds the relevant connectivity information to that application's `VCAP_SERVICES`.

If Cloud Foundry supports the backing service that you need - like MySQL or MongoDB - and if your code has been written in such a way that it centralizes the initialization or acquisition of these backing services - ideally using something like dependency injection (which Spring makes dead simple!) - then switching is a matter of rewiring that isolated dependency. If your application has been written to support 12 Factor-style configuration where things like credentials, hosts, and ports are maintained in the environment or at least external to the application build then you may be able to readily point your application to its new services without even so much as a rebuild. For a deeper look at this topic, check out this [blog on 12 Factor app style service configuration](#).

Often, however, it's not this simple. Classic J2EE / Java EE applications often resolve services by looking them up in a well-known context like JNDI. If your code was written to use dependency injection then it'll be fairly simple to simply to rewire the application to resolve its connection information from the Cloud Foundry environment. If not, then you'll need to rework your code and - ideally - do so by introducing dependency injection to insulate your application from further code duplication.

Achieving Service Parity with Spring

In this section, we'll look at some things that people tend to struggle with when moving applications to lighter weight containers and - by extension - the cloud. This is by no means an exhaustive list.

Remote Procedure Calls

Cloud Foundry (and indeed the majority of clouds) are HTTP-first. It supports individually addressable nodes, and it even now has support for non-routable custom ports, but these features work against the grain and aren't supported in every environment. If you're doing RPC with RMI/EJB, for example, then you'll need to tunnel it through HTTP. Ignoring for now the wisdom of using RPC, it's easier if you do RPC through HTTP. There are many ways to do this including XML-RPC, SOAP (bleargh!), and even [Spring's HTTP Invoker service exporters](#) and service clients which funnels RMI payloads through HTTP. This last option is convenient.

`DemoApplication.java` demonstrates how to export `SimpleMessageService` through its interface using HTTP Invoker.

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.remoting.httpinvoker.HttpInvokerService

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args
    }

    @Bean
    MessageService messageService() { ❶
        return new SimpleMessageService();
    }
}
```

```

    @Bean(name = "/messageService")
②
    HttpInvokerServiceExporter httpMessageService() {
        HttpInvokerServiceExporter http = new HttpInvoker
        http.setServiceInterface(MessageService.class);
        http.setService(this.messageService());
        return http;
    }
}

```

①

the implementation itself needs to implement, at a minimum, the service interface specified in the `HttpInvokerServiceExporter`

②

the `HttpInvokerServiceExporter` maps the given bean to an HTTP endpoint (`/messageService`) under the Spring DispatcherServlet.

The `Message` itself, of course, needs to implement `java.io.Serializable` to be serialized, just as with straight RMI serialization. Spring provides mirror image beans to create clients to these remote services based on an agreed upon service interface. In the example below, we'll use the `HttpInvokerProxyFactoryBean` to create a client side proxy to the remote service, bound to the same service contract. This shared contract, by the way, is the quality of RPC that's so limiting: it couples the client to the types of the service, and frustrates the service's ability to evolve without breaking the client.

```

package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.SpringApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.httpinvoker.HttpInvokerProxyF

```

```

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class DemoApplicationTests {

    private Log log = LogFactory.getLog(getClass());
    private ConfigurableApplicationContext serviceApplication

    @Before
    public void before() throws Exception {
        this.serviceApplicationContext = SpringApplication
            .run(DemoApplication.class); ❶
    }

    @After
    public void tearDown() throws Exception {
        this.serviceApplicationContext.close();
    }

    @Test
    public void contextLoads() throws Exception {

        ❷
        AnnotationConfigApplicationContext clientContext
            DemoApplicationClientConfiguration

        ❸
        MessageService messageService = clientContext
            .getBean(MessageService.class);
        Message result = messageService.greet("Josh");
        assertNotNull("the result must not be null", result);
        assertEquals(result.getMessage(), "Hello, Josh!");
        log.info("result: " + result.toString());
    }

    @Configuration
    public static class DemoApplicationClientConfiguration {

        @Bean
        HttpInvokerProxyFactoryBean client() { ❹
            HttpInvokerProxyFactoryBean client = new
            client.setServiceUrl("http://localhost:80");
            client.setServiceInterface(MessageService);
            return client;
        }
    }
}

```

```
}
```

①

the test first stands up the HTTP Invoker service

②

..then stands up the configuration for the client

③

..then interacts with the service through the shared interface

④

the `HttpInvokerProxyFactoryBean` is the mirror image of the `HttpInvokerServiceExporter`

HTTP Sessions with Spring Session

Cloud Foundry (and most cloud environments in general) don't do well with multicast networking. One use case commonly associated with multicast networking is HTTP session replication. You can get HTTP session replication, foregoing multicast networking, by using [Spring Session](#). Spring Session is a drop in replacement for the Servlet HTTP Session API that relies on an SPI to handle synchronization. The default implementation of this SPI uses Redis for distribution, instead of multicast. You just install Spring Session, you don't have to do anything else to your HTTP session code. The HTTP Servlet specification provides for replacing the implementation in this manner, so this works in a consistent manner across Servlet implementations. Spring Session in turn writes session state through the SPI. Redis is available as a backing service on Cloud Foundry. As multiple nodes spin up, they all talk to the same Redis cluster, and benefit from Redis' world-class state replication. Spring Session gives you a few other features too, for free. [You might consult this blog - *The Portable, Cloud Ready, Session - for more details.*](#)

```
package demo;
```

```
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.View;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

@SpringBootApplication
public class DemoApplication {

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

@RestController
class SessionController {

    @Value("${CF_INSTANCE_IP:127.0.0.1}")
    private String ip;

    @RequestMapping("/hi")
    Map<String, String> uid(HttpSession session) {
        ①
        UUID uid = (UUID) session.getAttribute("uid");
        if (uid == null) {
            uid = UUID.randomUUID();
        }
        session.setAttribute("uid", uid);

        Map<String, String> m = new HashMap<>();
        m.put("instance_ip", this.ip);
        m.put("uuid", uid.toString());
        return m;
    }
}
```

```
    }  
  
}  
  
❶
```

this example stores an attribute in the HTTP session if it's not already present. Subsequent calls to the `/hi` endpoint should return the same, cached value in the HTTP session.

To see the example at work, bring up Redis with the Redis CLI, then clear it using the `FLUSHALL` command. My Redis shell looks like this:

Then, bring up the web application at `http://localhost:8080/hi`. This will trigger a new HTTP session which Spring Session will persist in Redis.

Confirm this by using the `KEYS *` command in the Redis CLI.

the Java Message Service

I don't know of a good JMS solution for Cloud Foundry. It's invasive, but straightforward, to rework most JMS code to use the AMQP protocol, which RabbitMQ speaks. If you're using Spring, then the primitives for dealing with JMS or RabbitMQ (or indeed, Redis' publish-subscribe support) look and work similarly. RabbitMQ and Redis are available on Cloud Foundry.

Distributed Transactions using the X/Open XA Protocol and JTA

If your application requires distributed transactions, using the XA/Open protocol and JTA, it's possible to [configure standalone XA providers using Spring](#) and it's downright easy to [do so using Spring Boot](#). You don't need a Java EE-container hosted XA transaction manager. The following example defines a JMS message listener and a JPA-based service.

```
package demo;  
  
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.transaction.Transactional;

// TODO map the transactional log to the filesystem. Use FUSE on

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args
    }
}

interface AccountRepository extends JpaRepository<Account, Long>
}

@Entity
class Account {

    @Id
    @GeneratedValue
    private Long id;

    private String username;

    Account() {
    }

    public Account(String username) {
        this.username = username;
    }

    public String getUsername() {
        return this.username;
    }
}
```

```

        }
    }

@Component
class Messages {

    private Log log = LogFactory.getLog(getClass());

    @JmsListener(destination = "accounts")
    public void onMessage(String content) {
        log.info("----> " + content);
    }
}

@Service
@Transactional
class AccountService {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Autowired
    private AccountRepository accountRepository;

    public void createAccountAndNotify(String username) {
        this.jmsTemplate.convertAndSend("accounts", username);
        this.accountRepository.save(new Account(username));
        if ("error".equals(username)) {
            throw new RuntimeException("Simulated error");
        }
    }
}

```

Spring Boot automatically enlists JDBC XADataSource and JMS XAConnectionFactory resources in a global transaction. A unit test demonstrates this by triggering a rollback and then confirming that there are no side-effects in either the JDBC DataSource or the JMS ConnectionFactory.

```

package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.springframework.boot.test.SpringApplicationConfigurati
import org.springframework.test.context.junit4.SpringJUnit4ClassR

import static org.junit.Assert.assertEquals;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = DemoApplication.class)
public class DemoApplicationTests {

    private Log log = LogFactory.getLog(getClass());

    @Autowired
    private AccountService service;

    @Autowired
    private AccountRepository repository;

    @Test
    public void contextLoads() {
        service.createAccountAndNotify("josh");
        log.info("count is " + repository.count());
        try {
            service.createAccountAndNotify("error");
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
        log.info("count is " + repository.count());
        assertEquals(repository.count(), 1);
    }
}

```

There are several properties that you can specify to configure where the underlying JTA implementation (Bitronix, Atomikos) store their transaction logs. Ideally, this transaction log should be someplace durable. Applications on Cloud Foundry do not have a guaranteed file system. You'll need to use something more permanent.

Cloud File Systems

Cloud Foundry doesn't provide a durable file system. You can use FUSE-based filesystems like SSHFS on Cloud Foundry. FUSE is a C/C++-level API for building filesystem implementations in userspace. There are all manner of FUSE-based filesystems that expose HTTP APIs, SSH

connections, MongoDB file systems, and much more as file systems to UNIX-style operating systems. This lets you mount, in userspace, a remote file system using SSH, for example. Naturally, you're going to need a remote machine on which you have SSH access for this to work, but it's one valid option and it's particularly convenient in the case of JTA which requires an actual honest-to-goodness `java.io.File` in order to keep promises about integrity of data. This is *also* slower.

If you need to read and write bytes, and don't care if the IO happens with a `java.io.File` or with an alternative, filesystem-like backing service, then there are many suitable alternatives worth consideration [like a MongoDB GridFS-based solution](#) or an Amazon Web services-based S3 solution. These backing services offer a filesystem-like API; you will read, write and query bytes by a logical name. Spring Data MongoDB provides the very convenient `GridFsTemplate` that makes short work of reading and writing data.

```
package demo;

import com.mongodb.gridfs.GridFSDBFile;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.gridfs.GridFsCriteria;
import org.springframework.data.mongodb.gridfs.GridFsTemplate;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayOutputStream;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Controller
@SpringBootApplication
public class DemoApplication {

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

```

}

@Controller
@RequestMapping(value = "/files")
class FileController {

    @Autowired
    private GridFsTemplate gridFsTemplate;

    ①
    @RequestMapping(method = RequestMethod.POST)
    String createOrUpdate(@RequestParam MultipartFile file) {
        String name = file.getOriginalFilename();
        maybeLoadFile(name).ifPresent(
            p -> gridFsTemplate.delete(getFile(
                gridFsTemplate
                    .store(file.getInputStream(), name)
                    .save()));
        return "redirect:/";
    }

    ②
    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    List<String> list() {
        return getFiles().stream().map(GridFSDBFile::getName)
            .collect(Collectors.toList());
    }

    ③
    @RequestMapping(value = "/{name:.+}", method = RequestMethod.GET)
    ResponseEntity<byte[]> get(@PathVariable String name) throws IOException {
        Optional<GridFSDBFile> optionalCreated = maybeLoadFile(name);
        if (optionalCreated.isPresent()) {
            GridFSDBFile created = optionalCreated.get();
            try (ByteArrayOutputStream os = new ByteArrayOutputStream()) {
                created.writeTo(os);

                HttpHeaders headers = new HttpHeaders();
                headers.add(HttpHeaders.CONTENT_TYPE, "application/octet-stream");
                return new ResponseEntity<byte[]>(os.toByteArray(),
                    HttpStatus.OK);
            }
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}

```

```

private List<GridFSDBFile> getFiles() {
    return gridFsTemplate.find(null);
}

private Optional<GridFSDBFile> maybeLoadFile(String name)
    GridFSDBFile file = gridFsTemplate.findOne(getFil
    return Optional.ofNullable(file);
}

private static Query getFilenameQuery(String name) {
    return Query.query(GridFsCriteria.whereFilename()
}
}

```

①

the `/files` endpoint accepts multipart file uploaded data and writes it to MongoDB's GridFS

②

the `/files` endpoint simply returns a listing of the files in GridFS

③

the `/files/{name}` endpoint reads the bytes from GridFS and sends them back to the client.

Alternatively, if your application's use of the file system is ephemeral - staging file uploads or something - then you can use your Cloud Foundry application's temporary directory but keep in mind that Cloud Foundry makes no guarantees about how long that data will survive. You don't need to worry about things like where your database lives and where the application logs live, though; Cloud Foundry will handle all of that for you.

HTTPS

Cloud Foundry terminates HTTPS requests at the highly available proxy that guards all applications. Any call that you route to your application will respond to HTTPS as well. If you're using on-premise Cloud Foundry, you can provide your own certificates centrally.

E-Mail

Does your application use SMTP/POP3 or IMAP? If you are using email from within a Java application, you're likely using JavaMail. JavaMail is a client Java API to handle SMTP/POP3/IMAP based email communication. There are many email providers-as-a-service. [SendGrid](#) - which is supported out of the box with Spring Boot 1.3 - is a cloud-powered email provider with a simple API.

```
package demo;

import com.sendgrid.SendGrid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class DemoApplication {

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

@RestController
class EmailRestController {

    @Autowired
    private SendGrid sendGrid;

    ①
    @RequestMapping("/email")
    SendGrid.Response email(@RequestParam String message) throws
        SendGrid.Email email = new SendGrid.Email();
        email.setHtml("<hi>" + message + "</h1>");
        email.setText(message);
        email.setTo(new String[]{"user1@host.io"});
        email.setToName(new String[]{"Josh"});
        email.setFrom("user2@host.io");
        email.setFromName("Josh (sender)");
        email.setSubject("I just called.. to say.. I (mes
    return sendGrid.send(email);
}
```

```
    }  
}  
❶
```

This REST API takes a message as a request parameter and sends an email using the auto-configured SendGrid Java client.

Identity Management

Identity management, authentication and authorization is a very important capacity in a distributed system. The ability to centrally describe users, roles and permissions is critical. Cloud Foundry ships with a powerful authentication and authorization service called UAA that you can talk to using Spring Security. Alternatively, you might [find that Stormpath](#) is a worthy third-party hosted service that can act as a facade in front of other identity providers, or be the identity provider itself. There's even a very simple Spring Boot and [Spring Security integration!](#)

Next Steps

Hopefully, there aren't any! The goal of this post was to address common concerns typical of efforts to move existing legacy applications to the cloud. Usually that migration involves some combination of the advice in this post. Once you've made the migration, have a strong cup of water! You've earned it. That's one less thing to manage and worry about.

If you have time and interest, there is quite a lot to do in making the move to cloud-native applications. This very blog is routinely full of such posts.

Chapter 11. The Observable System

If we’re to truly iterate in an agile fashion then software must be shippable and *released* to the customer after every iteration. The “customer” may be a client, it may be a non-profit, it may be an open-source project, or yourself. The “customer” describes anyone that draws value from the software. The sooner that software is released to the customer, the sooner it releases value. Released software derisks continued development by capturing business value. Released and *working* software derisks continued development by capturing business value.

How do we know if it’s *working*? Software is silent. It runs just as quietly when it’s working as when it’s dead. There are no telltale sounds or smells that signal its malfunction. When we build software, we need to build in a definition of *working*. This helps us set expectations for normal operating behavior. A baseline. A baseline serves as a basis for measured improvements in behavior, and helps better capture business value.

In this chapter, we’ll look at the *non-functional* or *cross-functional* capabilities that all applications need if we’re to have a hope of operationalizing it.

The rub is that these capabilities are not business differentiators; they’re *not* what your organization went into business to address! Yet, they’re critical to the continued and safe operation and evolution of an application. Michael Nygard details in his epic tome *Release It!* these sorts of capabilities and concerns in dizzying detail. The punchline is that code complete is *not* the same as production ready. Software can not be released if it is not production ready. Done *must* mean done.

An operationalized application is one that is *built* for production. Such an application will, as we’ll see through the balance of this chapter, contain a *lot* of code to work with a diverse ecosystem of tools (centralized log processing, health and process managers, job schedulers, distributed tracing systems, and so on) beyond the business differentiating functionality that is the essence of

the application. We will, by building our application with key tenants from the 12-factor manifesto in mind, and by working with the conventions of Spring Boot and Spring Cloud, benefit from this infrastructure if it is available. Make no mistake, however, that *something* needs to standup that infrastructure. Deploying into production blind, with no supporting infrastructure, is not an option. If the 12-factor manifesto describes a set of good-clean cloud-hygiene principles for building production-ready applications, then *something* needs to satisfy the other side of the contract and support what Andrew Clay Shafer refers to as *12-factor Operations*. Cloud Foundry, naturally, does a very good job supporting the operational requirements.

In this chapter we'll look at how to surface node-by-node information and how to centralize that information to support the single-pane-of-glass experience required for quick comprehension of a system's behavior. It is critical that we capture the behavior of the *system*, not just the applications in the system.

The map is not the terrain. Just as looking at a map of Manhattan has far less fidelity than actually walking through Manhattan, a system has emergent behavior that cannot be captured in an architecture diagram of the system. It can only be captured through effective, system-wide monitoring.

The New Deal

The requirements to successfully deploy applications into production have not changed drastically. What *has* changed is how divorced developers can afford to be from operational concerns, if the organization is to prosper, and how apathetic operations can be to application requirements that may risk system stability. The hand off between developers and operations used to be an opaque deliverable, something like a Servlet container `.war`. The application deployed in this black box benefitted from some container-provided services, like a built-in start and stop script and a central log spout. Operators needed to further customize such a container in order for any of those container promises to be meaningful. Operators would then need to ensure a whole world of supporting infrastructure was in place so that the application would enjoy stability in production:

- process scheduling and management: what component will start the application and gracefully shut it down? How do we ensure that it isn't running twice on the same host?
- application health and remediation: how do operators know if the application is running well? What happens if the application process dies? What happens if the host itself dies?
- log management: how do operators see the logs spooling from application instances? How is collection and analysis handled?
- application visibility and Transparency: how do operators capture application state, or quantify application state as metrics, and analyze them and visualize them?
- route management: is the application exposed to the internet? Load-balanced? Do the routes update correctly when the application is restarted on another host?
- distributed tracing: who is accessing the system? Which services are

involved in processing a transaction, and what is the latency of those requests? What does the average call graph look like?

- application performance management: how do operators diagnose and address complex application performance problems?

This infrastructure is hard to get right and expensive to develop. It is also, very importantly, not business differentiating functionality. We believe that opinionated platforms, like Heroku, or distributions of the open-source Cloud Foundry, provide the best mix of supporting infrastructure and ease-of-use. These platforms make certain assumptions about the applications - that they are transactional, online web applications written to the 12-factor manifesto - and applications benefit from these assumptions in automation and velocity.

Visibility and Transparency

Operational visibility is a tricky thing. If you have a monolithic application, things are in some ways easier. If something goes wrong with the system, there can be no question at least of *where* the error occurred: *the error is coming from inside the building!* Things become markedly more complicated in a distributed systems world as interactions between components make failure isolation critical. You wouldn't drive a single passenger car without instrumentation, gauges and windows supporting visibility; how could you hope to operate air-traffic control for hundreds of airplanes without visibility?

Improved visibility supports business' continued investment and changing priorities for a system. Operations uses *good* visibility to at least connect eyes to potential system problems (*alerting*); they use *great* monitoring infrastructure supports automatic response to system state changes.

Ideally, operational and business visibility can be correlated and used to drive a *single pane of glass* experience - a dashboard. In this chapter, we'll look at how to collect and understand **historical** and **present** status and how to support forward-looking **predictions**. Predictions, in particular, are driven by models based on historical data.

Historical data is data stored *somewhere* for a period of time. Historical data may drive long-term business insight, where fresher, more recent data may support operational telemetry, error analysis and debugging.

Your system is a complex machine, with lots of moving parts. It is difficult to know which information to collect and which to ignore, so err on the side of caution and collect as much as possible. Operational information collection can be tedious work! It is undifferentiated heavy-lifting; it will not land you a promotion or further your organization's standing in the market yet it is *required* in order to ensure the continued and safe operation of the application.

It is critical that effective collection of this information be as friction-free as

possible so that it is a no-brainer to introduce it consistently across services and projects. Organizations large and small know the dreaded corporate Wiki page (“500 Easy Steps to Production!”) full of boilerplate and manual work to be done before a service may be deployed. Here, Spring Boot stands particularly strong: use auto-configuration described in [WHAT CHAPTER DO WE TALK ABOUT AUTO CONFIG]() to codify the customizations and combinations of the ideas we discuss in this chapter. Get it right once, and reuse from there. Undifferentiated heavy lifting is the enemy of velocity.

Chapter 12. Push vs. Pull Observability and Resolution

Some monitoring and observability tools take a pull-based approach where centralized infrastructure pulls data from services at an interval, and some monitoring infrastructure expects events about the status of different nodes to *push* that information to it. Many of the tools that we'll look in this chapter can work in one fashion or the other or sometimes both. It's up to you to decide upon which approach you'd like. For a lot of organizations, the discussion is one of resolution. How often do you update monitoring infrastructure? In a dynamic environment things may come and go as they need to. Indeed, the lifespan of a service might be only seconds. If a system employs pull-based monitoring then the interval between pulls may be longer than the entire span of a running application! The monitoring infrastructure is effectively blind to entire running components and could possibly miss out on major peaks and valleys in the data. This is one strong reason to embrace push-based monitoring for these kinds of components.

Here, we benefit considerably from Spring's flexibility: it often provides events that we can use to trigger monitoring events. As you read the following chapter, ask yourself whether a given approach is pull or push-based, and ask how you could conceivably turn something pull to push-based if needed.

Capturing an Application's Present Status with Actuator

The present status of an application is the kind of information that you would project onto a dashboard perhaps then visualized on a giant screen in the office where people can see it. You may or not keep all of this information for later use. Present-state status is like the speedometer in a car: it should tell you as concisely and quickly as possible whether there's trouble or not.

If you had to distill your system's state into a visualization (red to indicate danger, green to indicate that everything is alright, or yellow to indicate that something is perhaps amiss but within tolerable ranges), what information would you choose? That's present-state information.

This might include information like memory, worker threads, thread pools, database connections, total requests processed and statistics like requests per second (for all parts of the system that have requests, including HTTP endpoints and message queues), errors encountered, and the state of circuit breakers.

The *Spring Boot Actuator* framework provides out-of-the-box support for surfacing information about the application through endpoints. Endpoints collect information and sometimes interact with other subsystems. These endpoints may be viewed a number of different ways (using JMX, or the CrashHub Remote Shell, for example). We'll focus on REST endpoints. Endpoints are pluggable, and various Spring Boot-based subsystems often contribute custom, additional endpoints where appropriate. To use Spring Boot Actuator, add `org.springframework.boot : spring-boot-starter-actuator` to your project's build. Add `org.springframework.boot : spring-boot-starter-web` to have the custom endpoints exposed as REST endpoints.

Tip

from the documentation: “an actuator is a manufacturing term, referring to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.”

Table 12-1. a few of the Actuator endpoints

| Endpoint | Usage |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /info | exposes information about the current service |
| /metrics | expose quantifiable values about the service |
| /beans | expose a graph of all the objects that Spring Boot has created for you |
| /configprops | exposes information about all the properties available to configure the current Spring Boot application |
| /mappings | exposes all the HTTP endpoints that Spring Boot is aware of in this application as well as any other metadata (such as specified content-types or HTTP verbs in the Spring MVC mapping) |
| /health | a description of the state of components in the system: <i>UP</i> , <i>DOWN</i> , etc. Also returns HTTP status codes. |
| /env | returns all of the known environment properties such as those in the operating system’s environment variables or the results of <code>System.getProperties()</code> |

Metrics

Metrics are numbers. In the Spring Boot Actuator framework, there are gauges and counters. A **gauge** records a single value. A **counter** records a delta (an increment or decrement). Metrics are numbers and so are easy to store, graph, and query. There are some metrics that operations, and operations alone, will care about: host-specific information like RAM use, disk space, and requests per second. Everybody in the organization will care about semantic metrics: how many orders were made in the last hour, how many orders were placed, how many new account sign-ups have occurred, which products were sold and how many, etc. By default, Spring Boot exposes these metrics at `/metrics`.

Example 12-1. metrics from a nondescript application's `/metrics` endpoint

```
{  
    "classes" : 9731,  
    "heap.committed" : 570368,  
    "nonheap.used" : 72430,  
    "systemload.average" : 3.328125,  
    "gauge.response.customers.id" : 7,  
    "gc.ps_marksweep.count" : 2,  
    "nonheap" : 0,  
    "counter.status.200.customers" : 1, ❶  
    "counter.status.200.customers.id" : 2,  
    "mem.free" : 390762,  
    "heap.used" : 179605,  
    "classes.unloaded" : 0,  
    "gauge.response.star-star.favicon.ico" : 4,  
    "instance.uptime" : 47231,  
    "counter.status.200.star-star.favicon.ico" : 2,  
    "threads.peak" : 21,  
    "nonheap.init" : 2496,  
    "threads.totalStarted" : 27,  
    "mem" : 642797,  
    "httpsessions.max" : -1,  
    "counter.customers.read.found" : 2,  
    "gc.ps_marksweep.time" : 96,  
    "uptime" : 52379,  
    "threads" : 21,  
}
```

```

"customers.count" : 6,
"gc.ps_scavenge.count" : 6,
"heap.init" : 262144,
"httpsessions.active" : 0,
"nonheap.committed" : 74112,
"gc.ps_scavenge.time" : 87,
"counter.status.200.admin.metrics" : 2,
"datasource.primary.usage" : 0,
processors" : 8, ②
"gauge.response.customers" : 9,
"heap" : 3728384,
"gauge.response.admin.metrics" : 4,
"threads.daemon" : 19,
"datasource.primary.active" : 0,
"classes.loaded" : 9731
}

```

①

the metrics include counts of requests, their paths and their HTTP status codes already. Here, 200 is the HTTP status code.

②

the Spring Boot Actuator *also* captures other salient information about the system, like how many processors are available.

The metrics *already* records a lot of useful information for us: it records all requests made (and the corresponding HTTP status code), information about the environment (like the JVM's threads, loaded classes, and information about any configured `DataSource` instances). Spring Boot conditionally registers metrics based on the subsystems in play.

You can contribute your own metrics using the `org.springframework.boot.actuate.metrics.CounterService` to record deltas (“one more request has been made”) or the `org.springframework.boot.actuate.metrics.GaugeService` to capture absolute values (“there are 140 users connected to the chat room”).

Example 12-2. collecting customer metrics with the `CounterService`

```
package demo.metrics;
```

```
import demo.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponen

import java.net.URI;

@RestController
@RequestMapping("/customers")
public class CustomerRestController {

    private final CounterService counterService; ❶
    private final CustomerRepository customerRepository;

    @Autowired
    CustomerRestController(CustomerRepository repository,
                          CounterService counterService) {
        this.customerRepository = repository;
        this.counterService = counterService;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/{id}")
    ResponseEntity<?> get(@PathVariable Long id) {
        return this.customerRepository
            .findById(id)
            .map(customer -> {
                String metricName = metri
                this.counterService.incre
                return ResponseEntity.ok(
                    ))
            .orElseGet(
                () -> {
                    String me
                    this.coun
                    return Re
                })
    }

    @RequestMapping(method = RequestMethod.POST)
    ResponseEntity<?> add(@RequestBody Customer newCustomer) {
        this.customerRepository.save(newCustomer);
        ServletUriComponentsBuilder url = ServletUriCompo
            .fromCurrentRequest();
    }
}
```

```

        URI location = url.path("/{id}").buildAndExpand(n
            .toUri());
        return ResponseEntity.created(location).build();
    }

    @RequestMapping(method = RequestMethod.DELETE)
    ResponseEntity<?> delete(@PathVariable Long id) {
        this.customerRepository.delete(id);
        return ResponseEntity.notFound().build();
    }

    @RequestMapping(method = RequestMethod.GET)
    ResponseEntity<?> get() {
        return ResponseEntity.ok(this.customerRepository.
    }

❸
protected String metricPrefix(String k) {
    return k;
}

}

```

❶ the CounterService is auto-configured for you. If you're using Java 8, you'll get a better performing implementation than on earlier Java versions

❷

..record how many requests resulted in a successful match

❸

..how many were a miss

❹

we'll override this method in the next example to change the key used for the metric.

The CounterService and GaugeService capture metrics as they're updated, in transaction. It's not always easy to insert instrumentation into the request path of some components, or perhaps it's easier to collect interesting metrics

in a single place. The Spring Boot Actuator

org.springframework.boot.actuate.endpoint.PublicMetrics interface supports centralizing metrics collection. Spring Boot provides implementations of this interface internally to surface information about the JVM environment, Apache Tomcat, the configured DataSource, etc.

Example 12-3. a custom PublicMetrics implementation exposing how many customers exist in the system

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.endpoint.PublicMetrics;
import org.springframework.boot.actuate.metrics.Metric;
import org.springframework.stereotype.Component;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

@Component
class CustomerPublicMetrics implements PublicMetrics {

    private final CustomerRepository customerRepository;

    @Autowired
    public CustomerPublicMetrics(CustomerRepository customerR
        this.customerRepository = customerRepository;
    }

    @Override
    public Collection<Metric<?>> metrics() {

        Set<Metric<?>> metrics = new HashSet<>();

        long count = this.customerRepository.count();

        Metric<Number> customersCountMetric = new Metric<
            count);
        metrics.add(customersCountMetric);
        return metrics;
    }
}
```

①

this metric reports the aggregate count of `Customer` records in the database.

So far we've looked at metrics as fixed point-in-time quantities. They represent a value as of the moment you review it. These are useful, but they don't have context. For some values, a fixed point-in-time value is pointless. Absent history - the perspective of time - it's hard to know whether a value represents an improvement or a regression. Given the axis of time, we can take a value and derive statistics: averages, medians, means, percentiles, etc.

The Spring Boot Actuator seamlessly integrates with the Dropwizard Metrics library. [Code Hale](#) developed the Dropwizard Metrics library while at Yammer to capture gauges, counters and a handful of other types of metrics, including a meter. A **meter** measures the rate of events over time (e.g., "orders per second"). If the Dropwizard Metrics library is on the CLASSPATH, and you prefix any metric captured through the `CounterService` or `GaugeService` with `meter.`, the Spring Boot Actuator framework will delegate to the Dropwizard Metrics `Meter` implementation.

From the [Dropwizard Metrics documentation](#): "Meters measure the rate of the events in a few different ways. The mean rate is the average rate of events. It's generally useful for trivia, but as it represents the total rate for your application's entire lifetime (e.g., the total number of requests handled, divided by the number of seconds the process has been running), it doesn't offer a sense of recency. Luckily, meters also record three different exponentially-weighted moving average rates: the 1-, 5-, and 15-minute moving averages."

The `Meter` doesn't need to retain *all* values it records as it uses an exponentially-weighted moving average; it retains a set of samples of values, over time. This makes it very efficient even over great periods of time.

Let's revise our example to use a `Meter` and then review the effect on the recorded metrics. We'll simply extend the previous `CustomerRestController`, overriding the `metricPrefix` method to prefix all `CounterService` metrics with `meter..`.

Example 12-4. collecting metered metrics with the CounterService and the Dropwizard Metrics library

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/metered/customers")
public class MeterCustomerRestController extends CustomerRestController {

    @Autowired
    MeterCustomerRestController(CustomerRepository repository,
                                CounterService counterService) {
        super(repository, counterService);
    }

    @Override
    protected String metricPrefix(String k) {
        return "meter." + k; ❶
    }
}
```

❶

prefix all recorded metrics with meter.

Example 12-5. metrics powered by the Dropwizard Meter

```
{
    "meter.customers.read.fifteenMinuteRate" : 0.102683423806518,
    "meter.customers.read.meanRate" : 0.00164167411117908,
    "meter.customers.read.fiveMinuteRate" : 0.0270670566473226,
    "meter.customers.read.oneMinuteRate" : 9.07998595249702e-06
    ...
}
```

A counter and gauge capture a single value. A meter captures the number of values over a time period. A meter does *not* tell us anything about the frequencies of values in a data set. A **histogram** is a statistical distribution of values in a stream of data: it shows how many times a certain value occurs. It

lets you answer questions like “what percent of orders have more than one item in the cart?” The Dropwizard Metrics `Histogram` measures the minimum, maximum, mean, and median as well as percentiles: 75th, 90th, 95th, 98th, 99th, and 99.9th percentiles.

If you’ve taken a basic statistics class then you know that we need all data points in order to derive these values. This is an *overwhelming* amount of data, even over a small period of time. Suppose your application saw a 1,000 requests a second and that there were ten actions per request. Over a day, that’s 864,000,000 values ($24 * 60 * 60 * 1000 * 10$)! If we’re using Java, that’s 8 bytes per `long`, and more than *six gigabytes* of data per day! Many applications will have more than ten actions per request, too.

The Dropwizard Metrics library uses *reservoir sampling* to keep a statistically representative sample of measurements as they happen. As time progresses the Dropwizard Histogram creates samples of older values and uses those samples to derive new samples going forward. The result isn’t perfect - it’s lossy - but it’s very efficient. The Spring Boot Actuator framework will automatically convert any value submitted using the `GaugeService` with a metric key prefix of `histogram.` into a Dropwizard `Histogram` if the Dropwizard Metrics library is on the CLASSPATH.

Example 12-6. collecting a distribution of file upload sizes using the Dropwizard Metrics histogram implementation

```
package demo.metrics;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

@RestController
@RequestMapping("/histogram/uploads")
public class HistogramFileUploadRestController {
```

```

private Log log = LogFactory.getLog(getClass());
private final GaugeService gaugeService;
@.Autowired
HistogramFileUploadRestController(GaugeService gaugeService)
    this.gaugeService = gaugeService;
}

@RequestMapping(method = RequestMethod.POST)
void upload(@RequestParam MultipartFile file) {
    long size = file.getSize();
    this.log.info(String.format("received %s with file
                                %s", file.getOriginalFilename(), size));
    this.gaugeService.submit("histogram.file-uploads.");
}
}

```

❶

prefix all recorded metrics with `histogram`. to have them converted into DropWizard `Histogram` instances behind the scenes.

This example maintains a histogram for file upload sizes. I used `curl` to upload three different files of varying size, randomly.

Table 12-2. the sample files and their sizes

| File Size | File Name | Frequency |
|------------------|------------------|------------------|
|------------------|------------------|------------------|

| | | |
|------|--------------------------------------|---|
| 8.0K | <code> \${HOME}/Desktop/1.png</code> | 2 |
|------|--------------------------------------|---|

| | | |
|-----|--------------------------------------|---|
| 32K | <code> \${HOME}/Desktop/2.png</code> | 5 |
|-----|--------------------------------------|---|

| | | |
|-----|--------------------------------------|---|
| 40K | <code> \${HOME}/Desktop/3.png</code> | 3 |
|-----|--------------------------------------|---|

The `/metrics` confirm what the data table tells us.

Example 12-7. metrics powered by the Dropwizard Histogram

```

{
    ...
    "histogram.file-uploads.size.snapshot.98thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.999thPercentile" : 38803
    "histogram.file-uploads.size.snapshot.median" : 29929,
    "histogram.file-uploads.size.snapshot.mean" : 27154.1998413605
    "histogram.file-uploads.size.snapshot.75thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.min" : 6347,
    "histogram.file-uploads.size.snapshot.max" : 38803,
    "histogram.file-uploads.size.count" : 10,
    "histogram.file-uploads.size.snapshot.95thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.99thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.stdDev" : 11639.41039254
    ...
}

```

The Dropwizard Metrics library also supports timers. A **timer** measures the rate that a particular piece of code is called and the distribution of its duration. It answers the question: how long does it *usually* take for a given type of request to be run and what are atypical durations? The Spring Boot Actuator framework will automatically convert any metric starting with `timer.` that's submitted using the `GaugeService` into a `Timer`.

You can time requests using a variety of mechanisms. Spring itself ships with the venerable `StopWatch` class.

Example 12-8. capturing timings using the Spring `StopWatch`

```

package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.http.ResponseEntity;
import org.springframework.util.StopWatch;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TimedRestController {

    private final GaugeService gaugeService;

    @Autowired

```

```

public TimedRestController(GaugeService gaugeService) {
    this.gaugeService = gaugeService;
}

@RequestMapping(method = RequestMethod.GET, value = "/tim
ResponseEntity<?> hello() throws Exception {
    StopWatch sw = new StopWatch(); ❶
    sw.start();
    try {
        Thread.sleep((long) (Math.random() * 60)
            return ResponseEntity.ok("Hi, " + System.
    } finally {
        sw.stop();
        this.gaugeService.submit("timer.hello", s
    }
}

}

```

❶

this is the Spring framework `StopWatch` which we use here to count how long a request took

Example 12-9. a timer

```
{
...
"timer.hello.snapshot.stdDev" : 11804,
"counter.status.200.timer.hello" : 7,
"timer.hello.snapshot.75thPercentile" : 35004,
"timer.hello.meanRate" : 0.0561559793104086,
"timer.hello.snapshot.mean" : 27639,
"timer.hello.snapshot.min" : 2007,
"timer.hello.snapshot.max" : 42003,
"timer.hello.snapshot.median" : 35004,
"timer.hello.snapshot.98thPercentile" : 42003,
"timer.hello.fifteenMinuteRate" : 0.182311662062598,
"timer.hello.snapshot.99thPercentile" : 42003,
"timer.hello.snapshot.999thPercentile" : 42003,
"timer.hello.oneMinuteRate" : 0.0741487724647533,
"timer.hello.fiveMinuteRate" : 0.153231174431025,
"timer.hello.count" : 7,
"gauge.response.timer.hello" : 28008,
"timer.hello.snapshot.95thPercentile" : 42003
...
}
```

}

Joined Up Views of Metrics

Thus far our examples have all run on a single node or host; one instance, one host. It will become critical to centralize the metrics from across all services and instances as we scale out. A time-series database is a database that's optimized for the collection, analysis and visualization of metrics over time. There are many popular time-series databases like [Ganglia](#), [Graphite](#), [OpenTSDB](#), [InfluxDB](#) and [Prometheus](#). A time-series database stores values for a given key over a period of time. Usually, they work in tandem with something that supports graphing of the data in the time-series database. There are many fine technologies for graphing time-series data, the most popular of which seems to be Grafana[<http://grafana.org/>]. Alternative visualization technologies abound; many companies have open-sourced their tools: [Vimeo's released Graph Explorer](#), [TicketMaster's released Metrilyx](#) and [Square's released Cubism.js](#).

Spring Boot supports writing metrics to a time-series database using implementations of the `MetricsWriter` interface. Out of the box, Spring Boot can publish metrics to a Redis instance, JMX (possibly more useful for development), out over a Spring framework `MessageChannel` or to any service [that speaks the StatsD protocol](#). StatsD was a proxy originally written in Node.js by folks at Etsy to act as a proxy for Graphite/Carbon, but the protocol itself has become so popular that many client and services speak the protocol and StatsD itself supports multiple backend implementations besides Graphite, including [InfluxDB](#), [OpenTSDB](#), and [Ganglia](#). To take advantage of the various `MetricWriter` implementations, simply define a bean of the appropriate type and annotate it with `@ExportMetricWriter`. Dropwizard also provides support for publishing metrics to downstream systems through its `*Reporter` implementations.

Metric Data Dimensions

The data you create in a time-series database is still data, even if the dimensions of that data would seem to be very limited: a metric has a key and

a value, at least. Therein lies the rub; there's very little in the way of *scheme*. Different time-series databases provide improvements, and offer other dimensions of data. Some offer the notion of *labels* or *tags*. The only dimension common to all time-series databases, however, is a key. Most implementations we've seen use hierarchical keys, e.g.: `a.b.c`. Many time-series databases support glob queries, e.g.: `a.b.*` that will return all metrics that match the prefix, `a.b`. Each path component in a key should have a clear, well-defined purpose and volatile path components should be kept as deep into the hierarchy as possible. Design your keys and metrics to support what will certainly be a growing number of metrics. Think carefully about what you capture in your metrics, be it through hierarchical keys or other dimensions like labels and tags.

How will you encode requests across different products in the same time-series database? Capture the component name, e.g.: `order-service`.

How will you encode different types of activities or processes: HTTP requests vs messaging-based back-office requests vs. back-office batch jobs, etc.: `order-service.tasks.fullfillment.validate-shipping` or `order-service.requests.new-order`.

How will you encode the information so that it could ultimately be correlated to product management-facing systems like HP's Vertica or Facebook's Scuba? While we'd like to claim that *all* operational telemetry directly translates into business metrics, it's just not true. It can be useful, though, to have a way of capturing this information and connecting it. You can do this upfront in the metrics' keys themselves or perhaps using labels and tags.

How will you correlate requests to A/B tests (or experiments) where a sample of the population runs through a code-path for a given feature that behaves differently than the majority of requests. This helps to gauge whether a feature works, and is well-received. The implication here is that you may have the same metric in two different code-paths, one a different *experimental* alternative to the other. Many organizations have a system for experiments, and surface experiment numbers that should be incorporated into the metrics.

As always, schema is a subjective matter and new time-series databases

differentiate on the richness of the data collected *and* the scale! Some time-series databases are lossy where others scale horizontally, almost infinitely.

Shipping Metrics from a Spring Boot Application

You can readily find hosted versions of many of these services. One is [Hosted Graphite](#), which is easy to integrate with. It pre-configures Graphite, Graphite Composer, and Grafana. Grafana and Graphite Composer both let you build graphs based on collected metrics. You can of course run your own Graphite instance, but keep in mind the goal is, as always, to get to production as quickly as possible so we tend to prefer cloud-based services; we don't like to run software and things unless you can sell them. There are a few worthy options for connecting to Graphite available to the Spring Boot developer. You can use Spring Boot's `StatsdMetricWriter`, which speaks the StatsD protocol and works with many of the aforementioned backends. You can also use one of the myriad native Dropwizard Metrics reporter implementations if, for example, you'd prefer to use a native protocol besides StatsD. We'll do that here to communicate natively with Graphite.

Example 12-10. configuring a Dropwizard Metrics `GraphiteReporter` instance

```
package demo.metrics;

import com.codahale.metrics.MetricRegistry;
import com.codahale.metrics.graphite.Graphite;
import com.codahale.metrics.graphite.GraphiteReporter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.actuate.autoconfigure.ExportMetricsFilter;
import org.springframework.boot.actuate.metrics.statsd.StatsdMetricWriter;
import org.springframework.boot.actuate.metrics.writer.MetricWriter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import java.util.concurrent.TimeUnit;

@Configuration
class ActuatorConfiguration {

    @PostConstruct
    public void begin() throws Exception {
```

```

        java.security.Security.setProperty("networkaddress.cache.ttl", "0");
    }

    @Bean
    GraphiteReporter graphiteWriter(
        @Value("${hostedGraphite.apiKey}") String apiKey,
        @Value("${hostedGraphite.url}") String host,
        @Value("${hostedGraphite.port}") int port
    ) {
        GraphiteReporter reporter = GraphiteReporter.forRegistry(graphiteWriter)
            .prefixedWith(apiKey) ❷
            .build(new Graphite(host, port));
        reporter.start(1, TimeUnit.SECONDS);
        return reporter;
    }
}

```

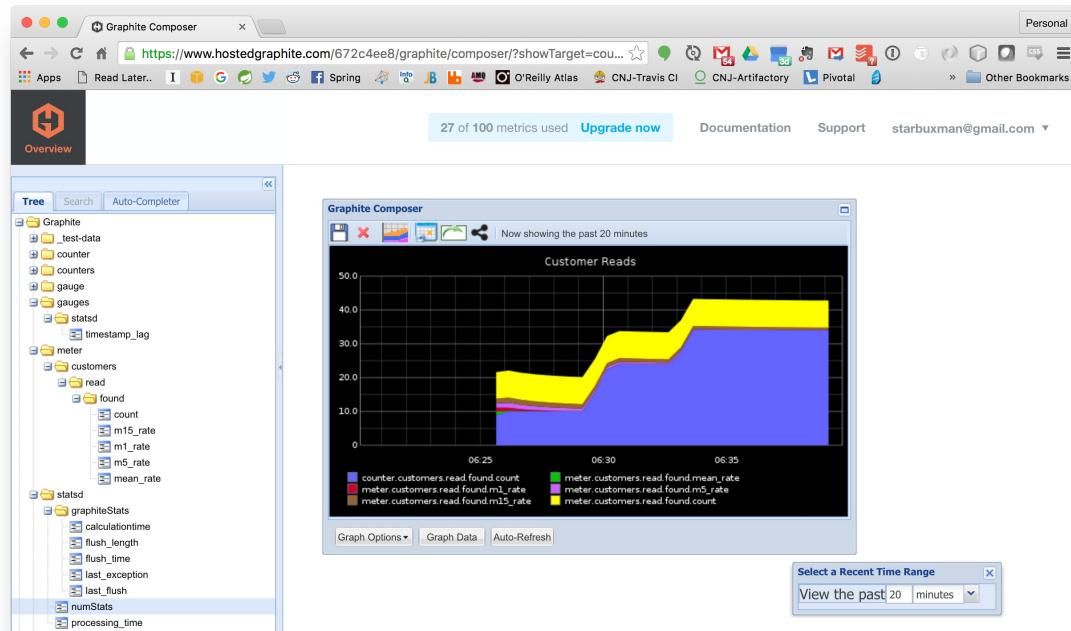
❶

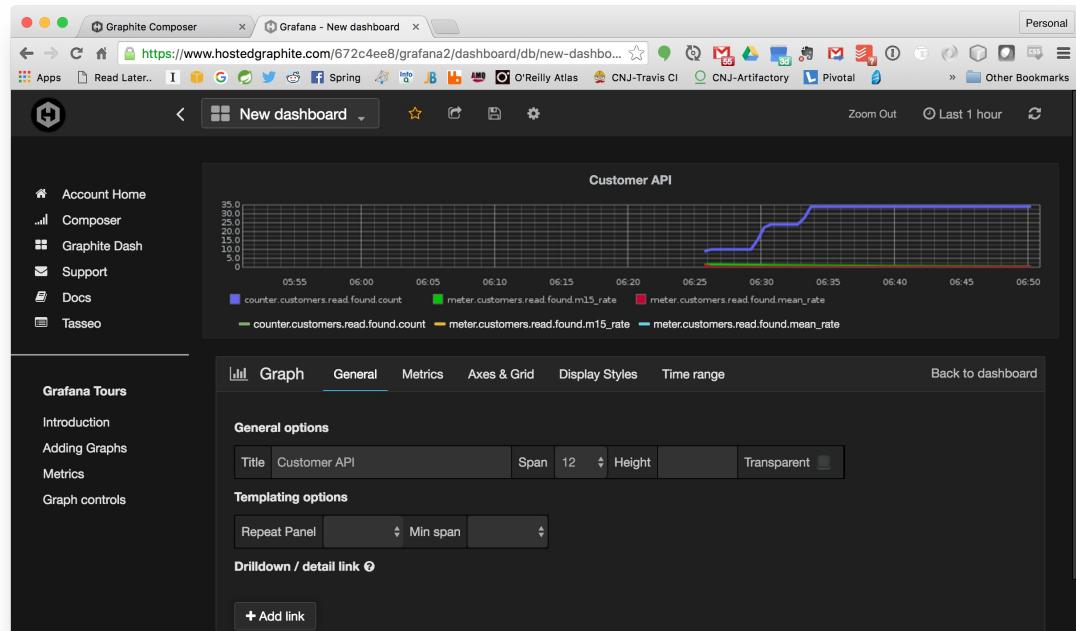
prevent DNS caching because HostedGraphite.com nodes may move and be mapped to a new DNS route

❷

[HostedGraphite.com's service](#) establishes authentication using an API key and expects that you transmit it as part of the `prefix` field. This is, admittedly, a bit of an ugly hack, but there is otherwise no obvious place to put the authentication information!

Now, you need only drive traffic to the `CustomerRestController` or `MeterCustomerRestController` and you'll see the traffic reflected in graphs that you can create on HostedGraphite.com in either the Graphana interface or the Graphite composer interface.





Identifying Your Service with the /info Endpoint

Ideally, you're developing your code in a continuous delivery pipeline. In a continuous delivery pipeline, *every commit could* result in a push to production. Ideally, this happens many times a day! If something goes wrong, the first thing people will want to know is which version of the code is running. Give your service the ability to identify itself using the /info endpoint.

The /info endpoint is intentionally left blank. The /info endpoint is a natural place to put information about the service itself. What's the service name? Which git commit triggered the build that ultimately resulted in the push to production? What is the service version?

You can contribute custom properties by prefixing properties with info. in the environment through the normal channels (application.properties, application.yml, etc.). You can also surface information about the state of the git source code repository when the project was built by adding in the pl.project13.maven : git-commit-id-plugin Maven plugin. This plugin is pre-configured for Maven users in the Spring Boot parent Maven build. It generates a file, git.properties containing the git.branch and git.commit properties. The /info endpoint will know to look for it if it's available.

Example 12-11. the Git branch, and commit id and time exposed from /info

```
{  
  git: {  
    branch: "master",  
    commit: {  
      id: "407359e",  
      time: "2016-03-23T00:47:09+0100"  
    }  
  }  
  ...  
}
```

It's very simple to add custom properties, as well. Any environment property prefixed with `info.` will be added to the output of this endpoint. Spring Boot's default Maven plugin configuration, for example, is already setup to handle Maven resource filtering. You can take advantage of Maven resource filtering to emit custom properties captured at build time, like the Maven `project.artifactId` and `project.version`.

Example 12-12. capturing custom build-time information like the project's `artifactId` and `version` with the `/info` endpoint by contributing properties during the build with Maven resource filtering.

```
info.project.version=@project.version@  
info.project.artifactId=@project.artifactId@
```

Once that's done, bring up your `/info` endpoint and identify what's happening with ease.

Example 12-13. capturing custom build-time information like the project's `artifactId` and `version` with the `/info` endpoint

```
{  
    ...  
    project: {  
        artifactId: "actuator",  
        version: "1.0.0-SNAPSHOT"  
    }  
}
```

Health Checks

An application needs a way of volunteering its health to infrastructure. A good health check should provide an aggregate status that sums up the reported statuses for individual components in play. Health checks are often used by load-balancers to determine the viability of a node. Load-balancers may evict nodes based on the HTTP status code returned. The `org.springframework.boot.actuate.endpoint.HealthEndpoint` collects all `org.springframework.boot.actuate.health.HealthIndicator` implementations in the application context and exposes them. Here's the output of the default `/health` endpoint in our sample application:

Example 12-14. the output of the default `/health` endpoint for our sample application

```
{  
    status: "UP",  
    diskSpace: {  
        status: "UP",  
        total: 999334871040,  
        free: 735556071424,  
        threshold: 10485760  
    },  
    redis: {  
        status: "UP",  
        version: "3.0.7"  
    },  
    db: {  
        status: "UP",  
        database: "H2",  
        hello: 1  
    }  
}
```

Spring Boot automatically registers common `HealthIndicator` implementations based on various auto-configurations for JavaMail, MongoDB, Cassandra, JDBC, SOLR, Redis, ElasticSearch, the file system, etc. Here we can see that Redis, the file system and our JDBC `DataSource` are all automatically instrumented.

Let's contribute a custom `HealthIndicator`. The contract for a `HealthIndicator` is simple: when asked, return a `Health` instance with the appropriate status. Other components in the system need to be able to influence the returned `Health` object. You *could* directly inject the required `HealthIndicator` and manipulate its state in every component that cares, but this couples a *lot* of application code to a secondary concern, the health. An alternative approach is to use Spring's `ApplicationContext` event-bus to publish events within components and manipulate the `HealthIndicator` based on acknowledged events.

In the following example we'll establish an emotional health indicator that is happy (`UP`) or sad (`DOWN`) when it receives a `HappyEvent` or a `SadEvent` accordingly.

Example 12-15. an *emotional* `HealthIndicator`

```
package demo.health;

import org.springframework.boot.actuate.health.AbstractHealthIndicator;
import org.springframework.boot.actuate.health.Health;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.Optional;

@Component
class EmotionalHealthIndicator extends AbstractHealthIndicator {

    private EmotionalEvent event;
    private Date when;

    ①
    @EventListener
    public void onHealthEvent(EmotionalEvent event) {
        this.event = event;
        this.when = new Date();
    }

    ②
    @Override
    protected void doHealthCheck(Health.Builder builder) thro
```

```

        Optional.ofNullable(this.event)
                    .ifPresent(
                        evt -> {
                            Class<? extends EmotionalEvent> e
                                = evt.getClass();
                            Health.Bu
                                .withName(e.getSimpleName())
                                .withValue("OK")
                                .build();
                            String ev
                                = e.getSimpleName();
                            healthBuilder
                                .addEvent(ev);
                        }
                    );
    }
}

```

① we'll connect this listener method to application context events using the `@EventListener` annotation.

② the `doHealthCheck` method uses the `Health.Builder` to toggle the state of the health indicator based on the last known, recorded `EmotionalEvent`.

Example 12-16. the `sadEvent`

```

package demo.health;

public class SadEvent extends EmotionalEvent {
}

```

Example 12-17. the `HappyEvent`

```

package demo.health;

public class HappyEvent extends EmotionalEvent {
}

```

Now, any component in the `ApplicationContext` needs only publish an appropriate event to trigger an according status change:

Example 12-18. the emotional REST endpoint

```
package demo.health;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
class EmotionalRestController {

    private final ApplicationEventPublisher publisher; ❶

    @Autowired
    EmotionalRestController(ApplicationEventPublisher publish
                           this.publisher = publisher;
    }

    @RequestMapping("/event/happy")
    void eventHappy() {
        this.publisher.publishEvent(new HappyEvent()); ❷
    }

    @RequestMapping("/event/sad")
    void eventSad() {
        this.publisher.publishEvent(new SadEvent());
    }
}
```

❶

Spring automatically exposes an implementation of the `ApplicationEventPublisher` interface for use in component code. Indeed, the `ApplicationContext` Spring is using to run your application probably already *is* an `ApplicationEventPublisher`!

❷

from there it's trivial to dispatch an event between components.

Application Logging

It's 2016 and one of the best ways we have to understand a given node or system's behavior remains the venerable log file. Logs reflect the activity's rhythm, its activity. Logging requires intrusive statements be added to the code but the resulting log files themselves are one of the most decoupled tools out there! There are entire ecosystems, tools, languages, and big-data platforms that have developed entirely around usefully mining data from logs.

Logs have become such a natural extension of an application's state that it can be dizzying and difficult to even begin to choose which logging technology to use! So, let's start with that: if you're using Spring Boot, then you're probably fine just using the defaults. Spring Boot uses Commons Logging for all internal logging, but leaves the underlying log implementation open. Default configurations are provided for the JDK's logging support, Log4J, Log4J2 and Logback. In each case loggers are pre-configured to use console output with optional file output also available. By default, Spring Boot will use Logback, which in turn can capture and forward logs produced from other logging technologies like Apache Commons Logging, Log4j, Log4J2, etc. What's this mean? It means that all the likely log producing dependencies on the CLASSPATH will work just fine out of the box, and arrive on the console in a well-known configured format that includes the date-and-time, log level, process ID, the thread name, the logger name, and the actual log message. It'll even include color codes if you're viewing the logs on a console! You can even use Spring Boot to manage log-levels in a generic fashion by specifying the levels in the Spring environment.

Example 12-19. tuning logging levels

```
logging.level.demo=WARN
```

If you run this application, you'll see one message emitted to the console.

Example 12-20. a Java application that logs three messages at different log levels on

startup

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
public class LoggingApplication implements CommandLineRunner {

    public static void main(String args[]) {
        SpringApplication.run(LoggingApplication.class, a
    }

    private Log log = LogFactory.getLog(getClass());

    @Override
    public void run(String... args) throws Exception {

        String greeting = "Hello, world!";

        this.log.info("INFO: " + greeting);
        this.log.warn("WARN: " + greeting); ❶
        this.log.debug("DEBUG: " + greeting);
    }
}
```

❶

only this message will appear on the console.

Logs appear, by default, on the console. You can optionally configure writing out to a file or some other log appender but the console is a particularly sensible default. If you're doing development you'll want to see the logs as they arrive, and if you're running your application in a cloud environment then you shouldn't need to worry about where the logs get routed to. This is one of the tenants of the [12-factor manifesto](#):

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to

stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app's behavior.

Log collectors or log multiplexers take the resulting logs from disparate processes and unify them into a single stream, possibly forwarding that stream onward to some place where it may be analyzed. Log data should be as structured as possible - use consistent delimiters, define groups, and support something like a log *schema* - to support analysis. Logs should be treated as event streams; they tell a story about the behavior of the system. Log information might be the output of one process and the *input* to another downstream analytical process. One very popular log multiplexer is called Logstash. Logstash provides numerous plugins that let you pipeline logs from multiple input sources and connect those logs to a central analytics system like Elasticsearch, a full-text search engine powered by Lucene.

If you're using Cloud Foundry, it provides a log multiplexer, Loggregator, that will aggregate and forward logs to your console, using `cf logs $YOUR_APP_NAME`, or to any SyslogD protocol-compliant service, including on-premise or hosted services like ElasticSearch (via Logstash), Papertrail, Splunk, Splunk Storm, SumoLogic or of [course SyslogD itself](#). Configure a log drain as you would any user-provided service, specifying `-l` to signal that it's to be a log drain:

Example 12-21.

```
cf cups my-logs -l syslog://logs.papertrailapp.com:PORT
```

Then, it's just a service that's available for any application to bind to:

Example 12-22.

```
cf bind-service my-app my-logs  
cf restart my-app
```

Loggregator also publishes log messages on the websocket protocol and so it's very simple to programmatically *listen* to logs coming off of any Cloud Foundry applications. We're using Java, naturally, and so benefit from the Cloud Foundry client API's easy integration with this websocket feed. We'll

revisit this possibility later when we look at “[Remediation](#)”.

Distributed Tracing

Understanding an application's behavior and performance characteristics is critical. There are agent-based instrumentation, things like New Relic and App Dynamics, which use Java agents and automatic instrumentation to give you a low-level perspective of an application's performance behavior. These tools are worth investigation as they can give you runtime perspective visibility into an application's performance. APM tools like New Relic can give you a cross-language and technology dashboard of an application's end-to-end behavior from HTTP request down to low-level datasource access.

Advances in technology and cloud computing have made it easier to stand up and deploy services with ease. Cloud computing enables us to automate away the pain (from days or weeks (gasp!) to minutes!) associated with standing up new services. This increase in velocity in turn enables us to be more agile, to think about smaller batches of independently deployable services. The proliferation of new services complicates reasoning about system-wide and request-specific performance characteristics.

When all of an application's functionality lives in a *monolith* - what we call applications written as one, large, unbroken deployable like a `.war` or `.ear` - it's much easier to reason about where things have gone wrong. Is there a memory leak? It's in the monolith. Is a component not handling requests correctly? It's in the monolith. Messages getting dropped? Also, probably in the monolith. Distribution changes everything.

Systems behave differently under load and at scale. The specification of a system's behavior often diverges from the actual behavior of the system, and the actual behavior may itself vary in different contexts. It is important to contextualize requests as they transit through a system. It's also important to be able to talk about the nature of a specific request and to be able to understand that specific request's behavior relative to the general behavior of similar requests in the past minute, hour, day (or whatever!) other useful interval provides a statistically significant sampling. Context helps us establish whether a request was abnormal and whether it merits attention.

You can't trace bugs in a system until you've established a baseline for what *normal* is. How long is is *long*? For some systems it might be microseconds, for others it might be seconds or minutes!

In this section, we'll look at how Spring Cloud Sleuth, which supports distributed tracing, can help us establish this context and helps us better understand a system's actual behavior, not just its specified behavior.

Finding Clues with Spring Cloud Sleuth

Tracing is simple, in theory. As a request flows from one component to another in a system, through ingress and egress points, **tracers** add logic - instrumentation - where possible to perpetuate a unique **trace ID** that's generated when the first request is made. As a request arrives at a component along its journey, a new **span ID** is assigned for that component and added to the trace. A trace represents the whole journey of a request, and a span is each individual hop, or request, along the way. Spans may contain **tags**, or metadata, that can be used to later contextualize the request and perhaps correlate a request to a specific transaction. Spans typically contain common tags like start timestamps and stop timestamp, though it's easy to associate semantically relevant tags like a business entity ID with a span.

[Spring Cloud Sleuth](#) (`org.springframework.cloud:spring-cloud-starter-sleuth`) automatically instruments common communication channels:

- requests over messaging technologies like [Apache Kafka](#) or RabbitMQ (or any other [Spring Cloud Stream](#) binder)
- HTTP headers received at Spring MVC controllers
- requests that pass through a Netflix Zuul microproxy
- requests made with the `RestTemplate`, etc.
- requests made through the Netflix Feign REST client
- requests made through websockets

Spring Cloud Sleuth sets up useful log formatting for you that prints the trace ID and the span ID. Assuming you're running Spring Cloud Sleuth-enabled code in a microservice whose `spring.application.name` is `my-service-id`, you will see something like this in the logging for your microservice:

Example 12-23. Logs coming off a Spring Cloud Sleuth-instrumented application

```
2016-02-11 17:12:45.404  INFO [my-service-id,73b62c0f90d11e06,73b  
85184 --- [nio-8080-exec-1] com.example.MySimpleComponentMaking
```

In that example, `my-service-id` is the `spring.application.name`, `73b62c0f90d11e06` is the trace ID and `73b62c0f90d11e06` is the span ID. This information is very useful. You can publish your logs to log analysis and manipulation tools like Elasticsearch and Splunk. There are various ways to get that data there. Logstash, for example, is a log publisher that will write to ElasticSearch. Cloud Foundry automatically aggregates logs from all instances of a service into a multiplexed log through a tool called the [Loggregator](#) that can then be forwarded to any [Syslog](#)-compatible *drain*. Drains include popular services like [Splunk](#) or [PaperTrail](#). Whatever approach you take, you can do interesting queries if you have all the logs, and the trace information, in a single place available for query and analysis.

Spring Cloud Sleuth instrumentation usually consists of two components: an object that does the *tracing* of some subsystem, and the specific `SpanInjector<T>` instance for that subsystem. The tracer is usually some sort of interceptor, listener, filter, etc., that you can insert into the request flow for the component under trace. For example, Spring Cloud Sleuth automatically instruments all requests flowing through `HttpServletRequests` using a `javax.servlet.Filter`. In that code, it uses a `SpanInjector<HttpServletResponse>` instance that injects spans into the carrier object, a `HttpServletResponse`.

How Much Data is Enough?

Which requests should be traced? Ideally, you'll want enough data to see trends reflective of live, operational traffic. You don't want to overwhelm your logging and analysis infrastructure, though. Some organizations may only keep requests for every thousand requests, or every ten, or every million! By default, the threshold is 10%, or .1, though you may override it by specifying a sampling percentage:

Example 12-24. changing the sampling threshold percentage

```
properties
spring.sleuth.sampler.percentage = 0.2
```

Alternatively, you may register your own `Sampler` bean definition and programmatically make the decision which requests should be sampled. You can make more intelligent choices about which things to trace, for example, by ignoring successful requests, perhaps checking whether some component is in an error state, or really anything else. The `Span` given as an argument represents the span for the current in-flight request in the larger trace. You can do interesting and request-specific types of sampling if you'd like. You might decide to only sample requests that have a 500 HTTP status code, for example. The following `Sampler`, for example, would trace roughly half of all requests:

Example 12-25. the Spring Cloud `sampler` interface

```
package org.springframework.cloud.sleuth;

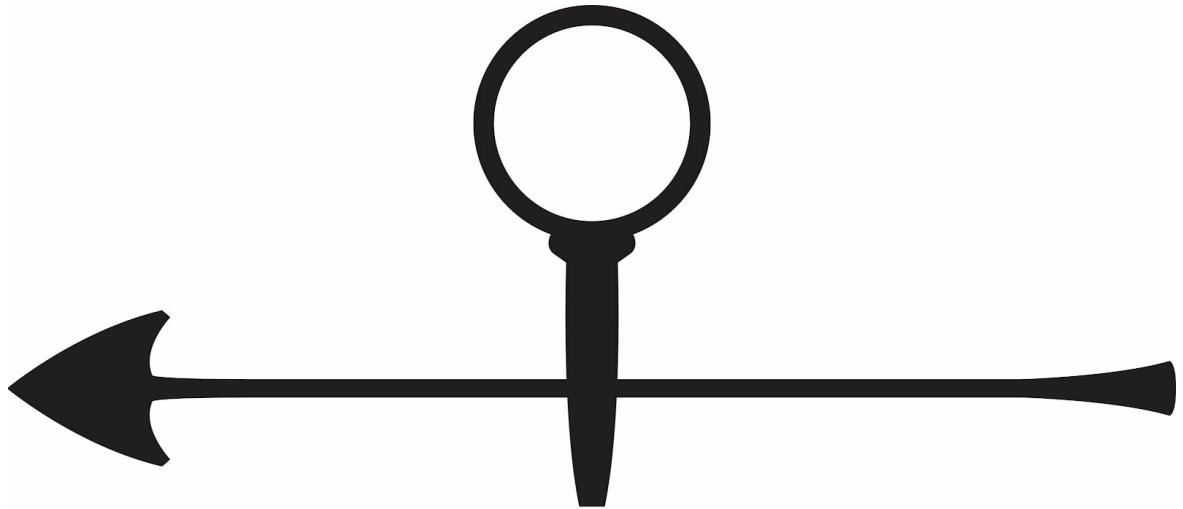
import org.springframework.cloud.sleuth.Span;

public interface Sampler {
    boolean isSampled(Span s);
}
```

Make sure to set realistic expectations for your application and infrastructure. It may well be that the usage patterns for your applications require something

more sensitive or less sensitive to detect trends and patterns. This is meant to be operational data; most organizations don't warehouse this data more than a few days or, at the upper bound, a week.

OpenZipkin: a Picture is worth a Thousand Traces



Z I P K I N

Data collection is a start but the goal is to *understand* the data, not just collect it. In order to appreciate the big picture, we need to get beyond individual events.

For this we'll use [the OpenZipkin project](#). OpenZipkin is the fully open-source version of Zipkin, a project that originated at Twitter in 2010, and is based [on the Google Dapper papers](#).

Previously, the open-source version of Zipkin evolved at a different pace than the version used internally at Twitter. OpenZipkin represents the synchronization of those efforts: [OpenZipkin](#) is Zipkin and when we refer to Zipkin in this post, we're referring to the version reflected in OpenZipkin.

Zipkin provides a REST API that clients talk to directly. Zipkin even supports a Spring Boot-based implementation of this REST API. Using that is as simple as using Zipkin's `@EnableZipkinServer` directly. The Zipkin Server delegates writes to the persistence tier via a `SpanStore`. Presently, there is support for using MySQL or an in-memory `SpanStore` out-of-the-box.

As an alternative to REST, we can *also* publish messages to the Zipkin server over a Spring Cloud Stream binder like RabbitMQ or Apache Kafka. We'll use this option. Add `org.springframework.cloud:spring-cloud-sleuth-zipkin-stream` to the CLASSPATH and then add

`@EnableZipkinStreamServer` to a Spring Boot application to accept and adapt incoming Spring Cloud Stream-based Sleuth Span's into Zipkin's `Span's and then persist them using the `SpanStore. You may use whatever Spring Cloud Stream binding you like, but in this case we'll use Spring Cloud Stream RabbitMQ (`org.springframework.cloud:spring-cloud-starter-stream-rabbitmq`).

Example 12-26. the Zipkin Server code

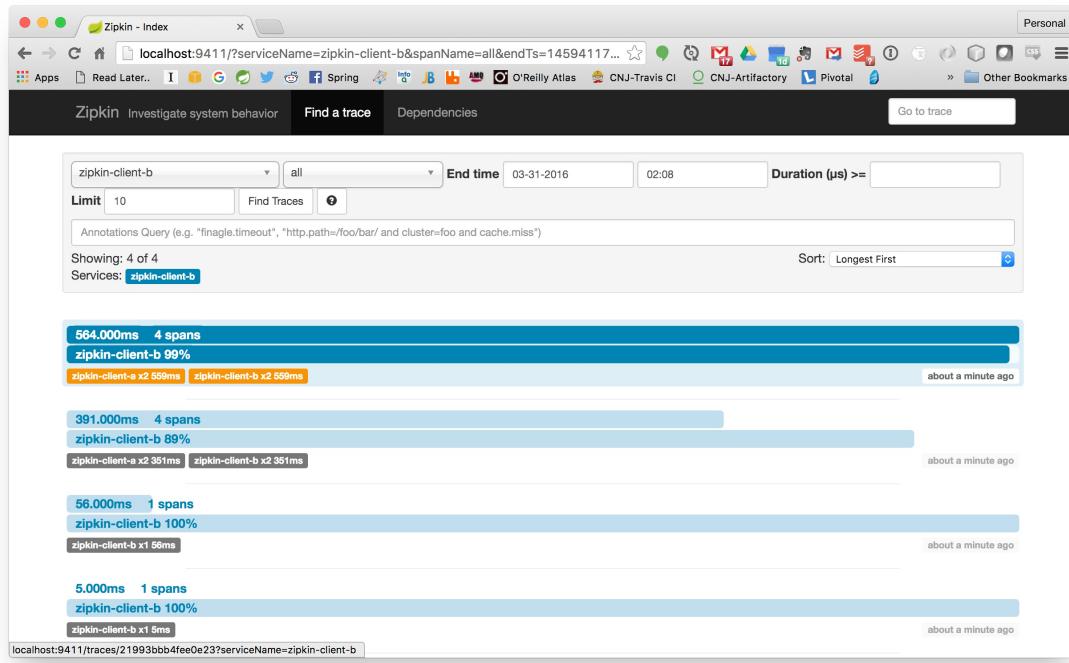
```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.sleuth.zipkin.stream.EnableZipki
①
@EnableZipkinStreamServer
②
@SpringBootApplication
public class ZipkinApplication {

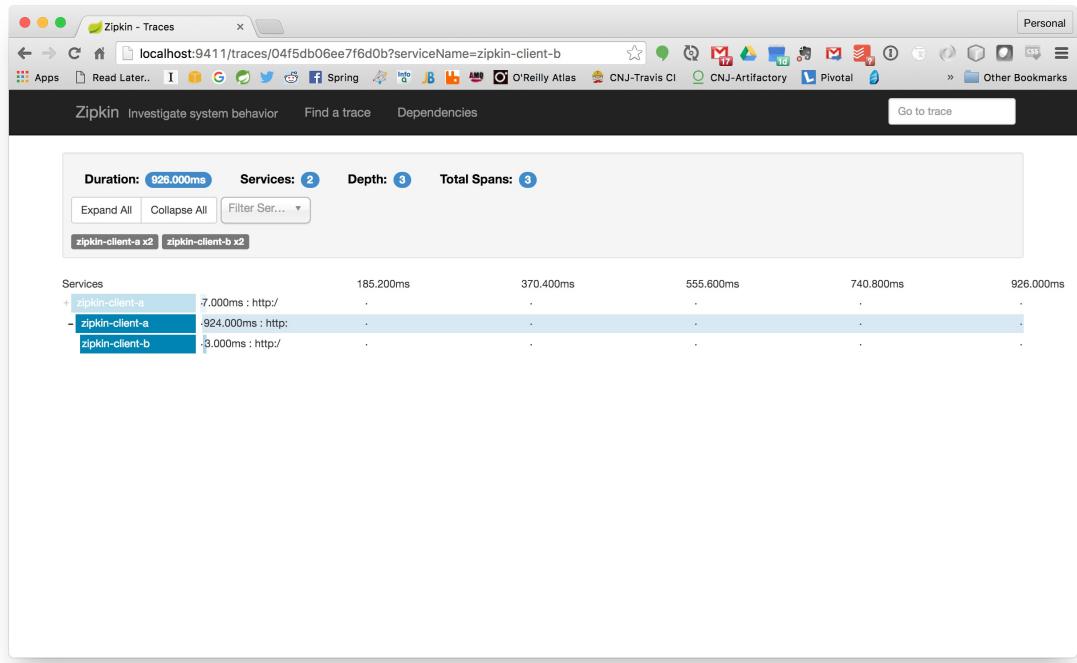
    public static void main(String[] args) {
        SpringApplication.run(ZipkinApplication.class, ar
    }
}
③
```

this tells the Zipkin server to listen for incoming spans on a stream

This server also includes the Zipkin UI (`io.zipkin:zipkin-ui`) on the CLASSPATH in order to visualize the information. Bring up the UI (also on port 9411, where the stream server lives) and then find all the recent traces. You can sort by most recent, longest, etc., for finer-grained control over which results you see.



You can inspect details about the trace.



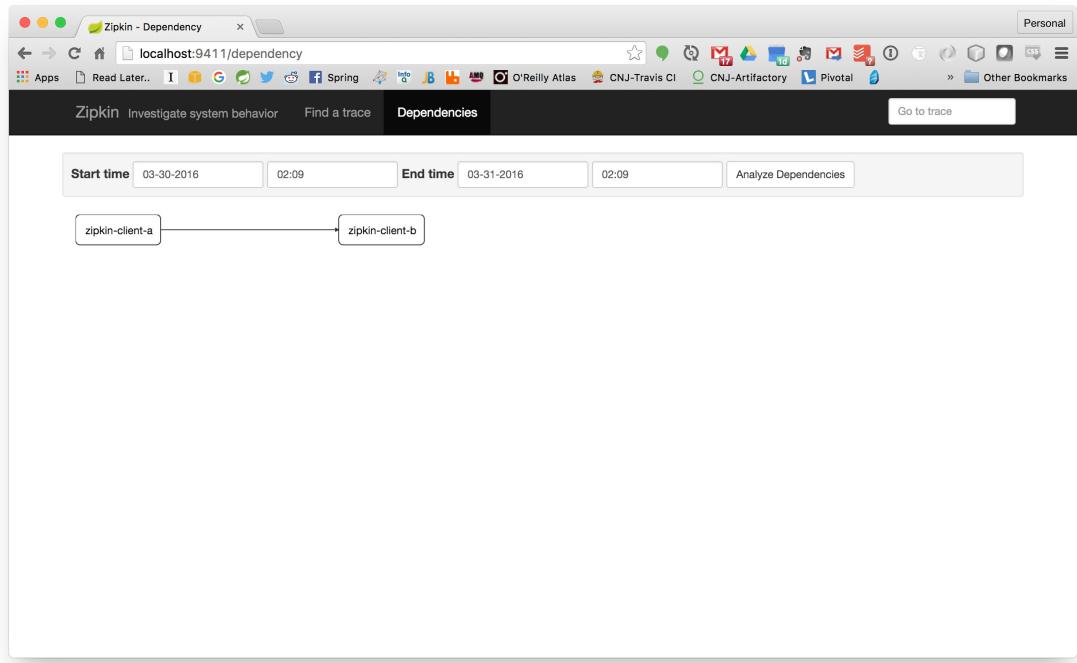
Each individual span also carries with it information (**tags**) about the particular request with which its associated. You can view this detail by clicking on an individual span:

The screenshot shows a web browser window titled "Zipkin - Traces" with the URL "localhost:9411/traces/04f5db06ee7f6d0b?serviceName=zipkin-client-b". The main content area displays a trace for "zipkin-client-a.http: 924.000ms". The trace table has columns: Date Time, Relative Time, Service, Annotation, and Host. The data shows four events: Client Send at 2.000ms from zipkin-client-a to host 192.168.99.1:8082; Server Receive at 3.000ms from zipkin-client-b to host 192.168.99.1:8081; Client Receive at 6.000ms from zipkin-client-a to host 192.168.99.1:8082; and Server Send at 6.000ms from zipkin-client-b to host 192.168.99.1:8081. A sidebar on the left lists services: "zipkin-client-a", "zipkin-client-b", and "zipkin-client". A "Dependencies" tab is visible at the top right. A large "926.000ms" timestamp is displayed on the right side of the trace table.

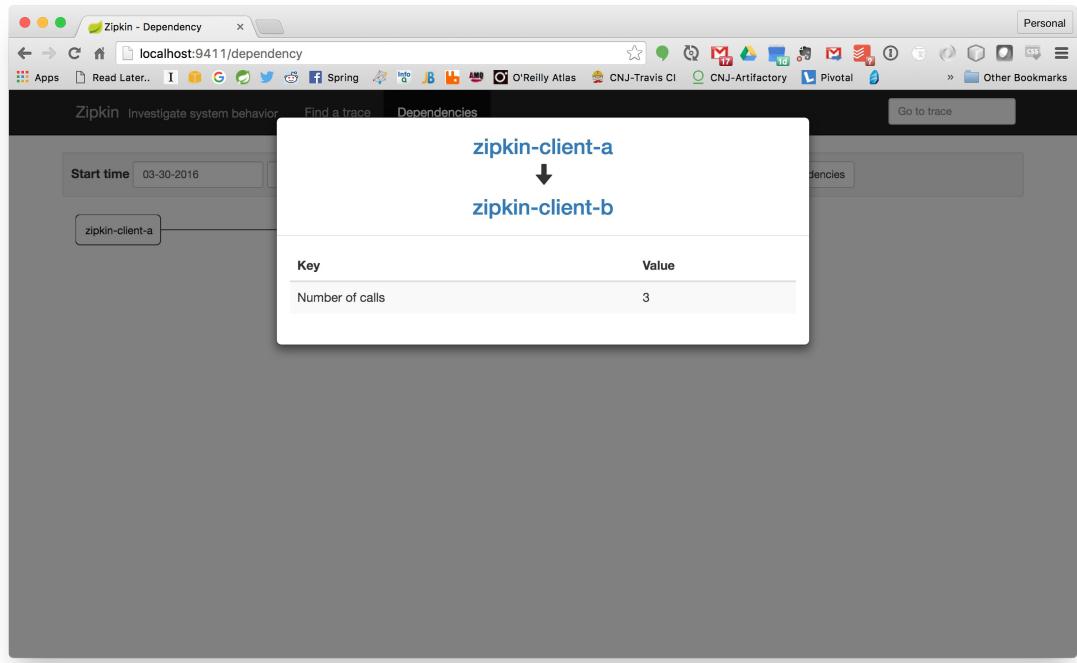
| Date Time | Relative Time | Service | Annotation | Host |
|-----------------------|---------------|-----------------|----------------|-------------------|
| 3/31/2016, 2:09:21 AM | 2.000ms | zipkin-client-a | Client Send | 192.168.99.1:8082 |
| 3/31/2016, 2:09:21 AM | 3.000ms | zipkin-client-b | Server Receive | 192.168.99.1:8081 |
| 3/31/2016, 2:09:21 AM | 6.000ms | zipkin-client-a | Client Receive | 192.168.99.1:8082 |
| 3/31/2016, 2:09:21 AM | 6.000ms | zipkin-client-b | Server Send | 192.168.99.1:8081 |

| Key | Value |
|----------------|-------------------|
| Server Address | 192.168.99.1:8082 |

Zipkin is an enviable position: it knows how services interact with each other. It knows the topology of your system. It'll even generate a handy visualization of that topology if you click on the *Dependencies* tab.



Each element in the visualization can give you further information still, including which components use it and how many (traced) calls have been made.



The OpenTracing Initiative

For Spring-based workloads, distributed tracing couldn't be easier! However, tracing, by its very nature, is a cross-cutting concern for all services no matter which technology stack they're implemented in. [The OpenTracing initiative](#) is an effort to standardize the vocabulary and concepts of modern tracing for multiple languages and platforms. The OpenTracing API has support from multiple *very large* organizations as its lead one of the original authors on the original Google Dapper paper. The effort defines language bindings; there are already implementations for JavaScript, Python, Go, etc. We will keep Spring Cloud Sleuth conceptually compatible with this effort and will track it. It is expected, but not implied, that the bindings will as often as not have Zipkin as their backend.

Dashboards

Thus far we've mostly looked at ways to surface information per-node and how to customize that information. This information is only useful in so far as we can connect it to a bigger picture about the larger system. The Actuator, for example, publishes information about the node but assumes that there's some sort of infrastructure to soak up this information and consolidate it, in a similar way to how Google manages services with their Borg Monitoring ("Borgmon") approach. Borgmon is a centralized management and monitoring solution used inside of Google, but it relies on each node exposing service information. Borgmon-aware services publish information over HTTP endpoints, even if the services they monitor aren't themselves HTTP. We'll see several options in this chapter on how to centralize and visualize the system itself, beyond the node-by-node endpoints provided by Actuator.

In this section we'll look at a few handy tools that support the ever important dashboard experience that both operations and business will appreciate. These dashboards often build on the tools we've looked at so far, presenting the relevant information in a single, at-a-glance experience. These tools rely on service registration and discovery to discover services in a system and then surface information about them. See [the section on routing](#) for details on working with a service registry like Netflix's Eureka. In this section, we rely on a Netflix Eureka registry being available so that our dashboards can discover and monitor the deployed services in our system. We might alternatively use Hashicorp Consul or Apache Zookeeper, or any other registry for which there's a Spring Cloud `DiscoveryClient` abstraction implementation available.

Monitoring Potentially Risky Service Calls with the Hystrix Dashboard

Spring Cloud supports easy integration with the Netflix Hystrix circuit breaker. Add `org.springframework.cloud:spring-cloud-starter-hystrix` to the CLASSPATH and add `@EnableCircuitBreaker` to a `@Configuration` class of any potentially shaky service-to-service call. Decorate any method that may result in an exception or hang because of a fallen downstream service with the `@HystrixCommand` annotation. Here's a trivial example that randomly inserts failure when issuing calls to either `http://google.com` or `http://yahoo.com`.

Example 12-27. use the Hystrix circuit breaker

```
package com.example;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.context.annotation.Bean;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.util.Random;

@SpringBootApplication
@EnableCircuitBreaker
❶
public class HystrixCircuitBreakerApplication {

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

```

public static void main(String[] args) {
    SpringApplication.run(HystrixCircuitBreakerApplic
}
}

@RestController
class ShakyRestController {

    @Autowired
    private RestTemplate restTemplate;

    ②
    public ResponseEntity<String> fallback() {
        return ResponseEntity.ok("ONOTES");
    }

    ③
    @HystrixCommand(fallbackMethod = "fallback")
    @RequestMapping(method = RequestMethod.GET, value = "/goo
    public ResponseEntity<String> google() {
        return this.proxy(URI.create("http://www.google.c
    }

    @HystrixCommand(fallbackMethod = "fallback")
    @RequestMapping(method = RequestMethod.GET, value = "/yah
    public ResponseEntity<String> yahoo() {
        return this.proxy(URI.create("http://www.yahoo.co
    }

    private ResponseEntity<String> proxy(URI url) {
        if (new Random().nextInt(100) > 50) {
            throw new RuntimeException("tripping circ
        }

        ResponseEntity<String> responseEntity = this.rest
            url, String.class);

        return ResponseEntity.ok()
            .contentType(responseEntity.getHe
            .body(responseEntity.getBody()));
    }

}

```

①

activate the circuit breaker (in this case, it's Netflix's Hystrix circuit breaker)

②

provide a fallback behavior that gets called when a circuit throws an exception. In this case we return a silly String

③

we decorate our various REST calls with a circuit breaker so that downstream service calls that may fail are safely handled.

Each node that uses the Hystrix circuit breaker also emits a server-sent event (SSE) heartbeat stream for every node that contains a circuit breaker. The SSE stream is accessible from `http://localhost:8000/hystrix.stream`, assuming the default configuration and that the above code was listening on port 8000. That stream is constantly updated. This stream contains information about the flow of traffic through the circuit, including how many requests were made, whether the circuit is open (and thus flowing directly through to the fallback behavior) or closed (and thus attempting to let traffic through to the potentially downed downstream service), and statistics about the traffic itself. It's not much to look at but you can take that stream and give it to another component, the Hystrix Dashboard, which can then visualize the transit of messages through the circuits in that specific node.

Example 12-28. monitor the flow of traffic through the circuit with the dashboard

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard

❶
@SpringBootApplication
public class HystrixDashboardApplication {

    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication
```

```
    }  
}
```

❶

trigger the creation of a Hystrix Dashboard UI, available at
`/hystrix.html`

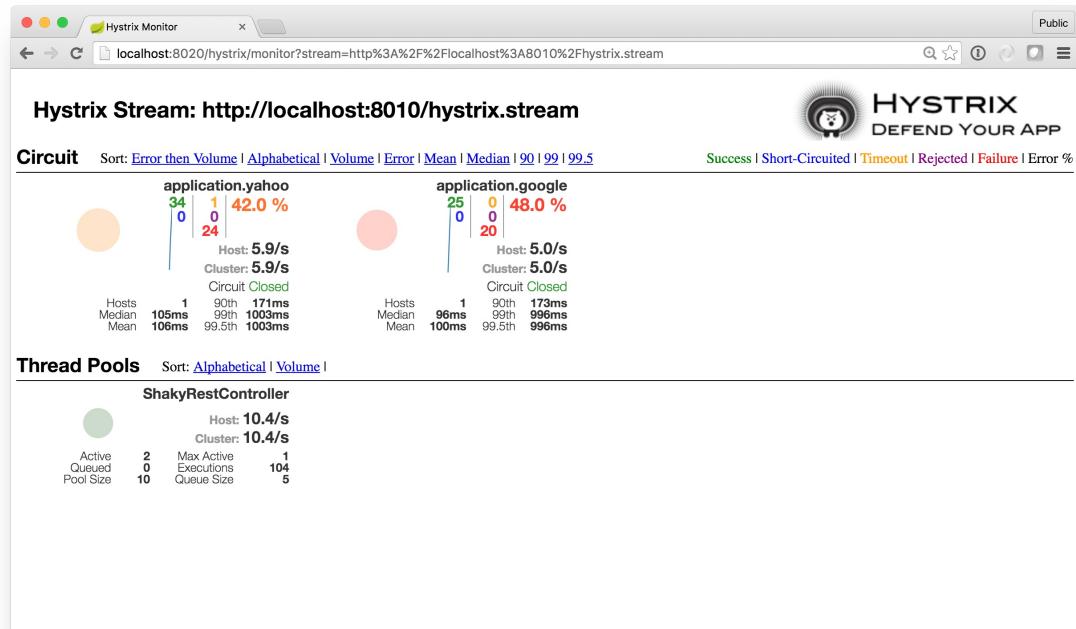
Naturally, this becomes untenable at scale where you have more than one instance of the same service or multiple services besides. We'll use Spring Cloud Turbine to multiplex all the streams from all the nodes into one Stream and then feed the resulting stream to our Hystrix Dashboard. Spring Cloud Turbine can aggregate services using service registration and discovery (via the Spring Cloud `DiscoveryClient` service registry abstraction) or through messaging brokers like RabbitMQ and Apache Kafka exposed (via the Spring Cloud Stream messaging abstraction). We'll power a Turbine-based circuit breaker using Spring Cloud Stream and RabbitMQ (with `org.springframework.cloud:spring-cloud-starter-stream-rabbitmq`).

Example 12-29. using Spring Cloud Turbine to aggregate the SSE streams from multiple circuit brokers across multiple nodes.

```
package com.example;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.turbine.stream.EnableTur  
  
@SpringBootApplication  
@EnableTurbineStream  
❶  
public class TurbineApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(TurbineApplication.class, a  
    }  
}
```

❶

stand up the Spring Cloud Stream-based Turbine aggregation stream



Codecentric's Spring Boot Admin

[Codecentric's Spring Boot Admin](#) is a project from the folks over at Codecentric. It provides an aggregated view of services and makes it dead simple to drop down into a Spring Boot-based service's logs, JMX environment, request logs, etc. The Spring Boot Admin is a slick way to get an enumeration of registered services in the system and to drill down on Actuator-exposed information.

To use it, setup a regular Spring Boot application instance and then add the following dependencies to the classpath: `de.codecentric:spring-boot-admin-server` and `de.codecentric:spring-boot-admin-server-ui`. The version itself will vary, of course, so check out the Git repository and your favorite Maven repository. In this example, we're using 1.3.5.

Example 12-30. Standing up an instance of the Spring Boot Admin

```
package com.example;

import de.codecentric.boot.admin.config.EnableAdminServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient

@EnableDiscoveryClient
@EnableAdminServer
@SpringBootApplication
public class SpringBootAdminApplication {

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

We'll connect the clients to the server with Spring Cloud's `DiscoveryClient` support. Add `org.springframework.cloud:spring-cloud-starter-eureka` and the `@EnableDiscoveryClient` annotation to both the service and all the clients. We avoid the need for our clients to be explicitly aware of the Spring

Boot Admin with service registration and discovery. You may alternatively use the Spring Boot Admin client dependency, `de.codecentric:spring-boot-admin-starter-client`. If you use the Spring Boot Admin client dependency, then you'll need to specify a `spring.boot.admin.url` property, pointing the clients to the Spring Boot Admin server instance.

Spring Boot Admin

192.168.99.1:8080/#/overview

Public

Apps dumb checkbox fix gnoming Docker Zipkin Spring Initializr Spring Initializr RabbitMQ Spring & Protocol Bu me as ASCII

spring

APPLICATIONS JOURNAL ABOUT

Spring-Boot applications

Here you'll find all Spring-Boot applications that registered themselves at this admin application.

Filter

| Application ▲ / URL | Version | Info | Status |
|------------------------------------------------------|---------|------|--------|
| spring-boot-admin-client http://192.168.99.1:8081 | | | UP |

Code licensed under Apache Open Source

Details Environment Logging JMX Threads Trace

The screenshot shows the Spring Boot Admin interface. At the top, there's a navigation bar with links to various services like Apps, Docker Zipkin, and RabbitMQ. Below that is a main header with the Spring logo and tabs for APPLICATIONS, JOURNAL, and ABOUT. The APPLICATIONS tab is selected. The main content area is titled "Spring-Boot applications" and contains a message about finding registered applications. A search bar labeled "Filter" is present. A table lists one application: "spring-boot-admin-client" with the URL "http://192.168.99.1:8081". The status column shows "UP". To the right of the table, a context menu is open, offering detailed monitoring and management options such as Environment, Logging, JMX, Threads, and Trace. At the bottom of the page, it states "Code licensed under Apache Open Source".

Spring Boot Admin

192.168.99.1:8080/#/apps/fe1f6d86/trace

Public

Apps dumb checkbox fix gnoming Docker Zipkin Spring Initializr Spring Initializr RabbitMQ Spring & Protocol Bu me as ASCII

spring-boot-admin-client up

APPLICATIONS JOURNAL ABOUT

DETAILS ENVIRONMENT LOGGING JMX THREADS TRACE

http://192.168.99.1:8081/health http://192.168.99.1:8081 http://192.168.99.1:8081

5 sec

19:53:18.349 GET /health
26.04.2016

19:53:17.698 GET /trace
26.04.2016

```
{  
    "method": "GET",  
    "path": "/trace",  
    "headers": {  
        "request": {  
            "accept": "application/json, text/plain, */*",  
            "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.266  
            "referer": "http://192.168.99.1:8080/",  
            "accept-language": "en-US,en;q=0.8",  
            "x-forwarded-host": "192.168.99.1:8080",  
            "x-forwarded-proto": "http",  
            "x-forwarded-prefix": "/api/applications/fe1f6d86/trace",  
            "accept-encoding": "gzip",  
            "host": "192.168.99.1:8081",  
            "connection": "Keep-Alive"  
        }  
    }  
}
```

Spring Boot Admin

192.168.99.1:8080/#/apps/fe1f6d86/details/metrics

Public

Apps dumb checkbox fix gnoming Docker Zipkin Spring Initializr Spring Initializr RabbitMQ Spring & Protocol Bu me as ASCII

spring-boot-admin-client up

APPLICATIONS JOURNAL ABOUT

DETAILS ENVIRONMENT LOGGING JMX THREADS TRACE

Application raw JSON

Health Checks raw JSON

| Application | UP |
|-------------|--------|
| DiskSpace | UP |
| Free | 684.2G |
| Threshold | 10M |

Memory

| Total Memory (93M / 482.3M) | 19.28% |
|-----------------------------|--------|
| Heap Memory (38.7M / 428M) | 9.03% |
| Initial Heap (-Xms) | 256M |
| Maximum Heap (-Xmx) | 3.6G |

JVM

| Systemload | 3.08 (last min. @ runq-sz) |
|------------------------|----------------------------|
| Uptime | 00:00:07:33 [d:h:m:s] |
| Available Processors | 8 |
| Current loaded Classes | 6405 |
| Total loaded Classes | 6405 |

The screenshot shows the Spring Boot Admin web interface. At the top, there's a navigation bar with links to various Spring-related tools like Docker Zipkin, Spring Initializr, and RabbitMQ. Below that is a main header with the Spring logo and tabs for APPLICATIONS, JOURNAL, and ABOUT. The APPLICATIONS tab is selected, showing a single application named 'spring-boot-admin-client' with a status of 'up'. Below this, there are several cards: one for 'Health Checks' showing disk space is up and within thresholds; two for 'Memory' showing total memory usage (19.28% of 482.3M) and heap memory usage (9.03% of 428M); and one for 'JVM' showing system load, uptime, available processors, and class loading statistics.

Ordina Microservices Dashboard

Ordina's JWorks division created another dashboard which provides a very handy *visual* enumeration of the registered services in a system. It discovers services through Spring Cloud's `DiscoveryClient` support.

Example 12-31. Standing up an instance of the Microservices Dashboard

```
package com.example;

import be.ordina.msdashboard.EnableMicroservicesDashboardServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient

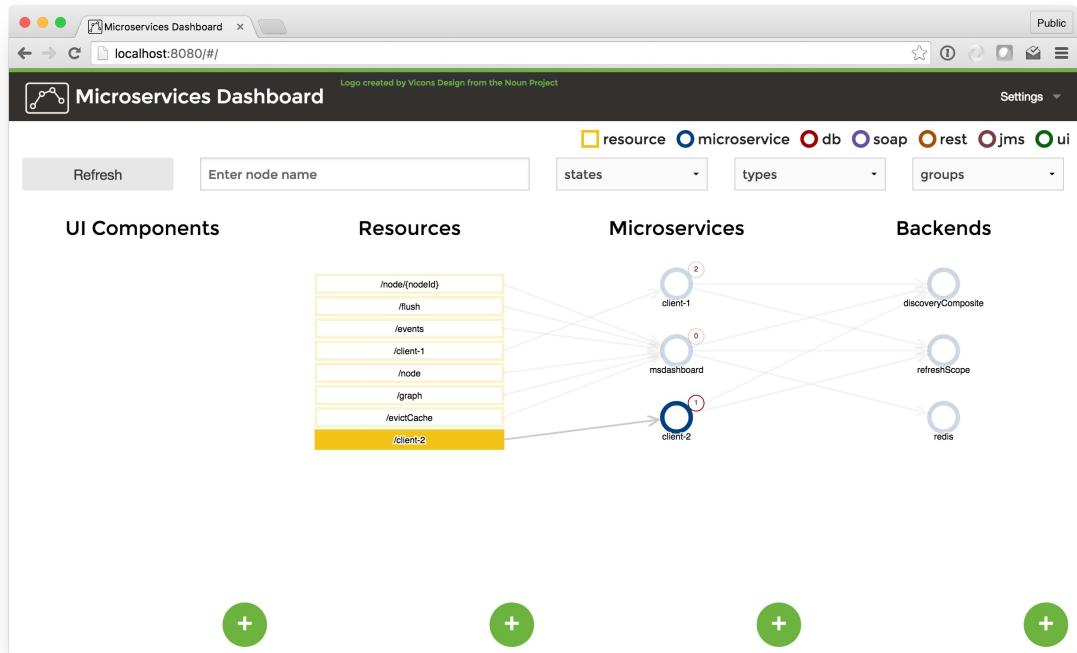
@SpringBootApplication
@EnableDiscoveryClient
@EnableMicroservicesDashboardServer
public class MicroservicesDashboardServerApplication {

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

The Microservices Dashboard provides a visualization of how services are connected together. It features four different lanes meant to reflect layers of components in a system:

- UI components are just that: user interface components, like Angular directives, for example
- Resources might be information drawn from the Spring Boot Actuator `/mappings` endpoint with the default spring mappings excluded or hypermedia links exposed on an index resource through an index controller
- Microservices are services (Spring Boot or not) discovered services (using Spring Cloud's `DiscoveryClient` abstraction)

- Backends are any `HealthIndicators` found on the discovered microservices



The Microservices Dashboard supports drilling down by node states, name, types and groups. You can add *virtual* nodes - nodes that aren't automatically discovered but about which you'd like to make the Microservices Dashboard aware. If nothing else, they could be placeholders for things that should be there, eventually, for planning purposes.

Remediation

Thus far we've focused on surfacing information about the state of the system, on improving visibility. What do we do with this knowledge? In a static, pre-cloud environment, improved visibility can be used to trigger alerting which then, hopefully, results in a pager going off or somebody getting an email. What happens then is anybody's guess, but chances are that - by that point - it's already too late and somebody's already had a bad experience on the system. Cloud computing changes this dynamic: we can solve software problems with.. software. If the system needs more capacity, we don't need to file an ITIL ticket, just make an API request! This is called *automatic remediation*.

Distributed computing changes the way we should think about application instance availability. In a sufficiently distributed system, having a single instance that is available 100% of the time becomes near impossible. Instead, the focus must be on building a system where service is somehow restored; we must optimize time-to-remediation. If time-to-remediation is zero seconds, then we are *effectively* 100% highly available, but this change in approach has profound implications in how we architect our system. We can only achieve this effect if we can program the platform itself.

The platform can even do basic remediation for you, automatically. Most cloud platforms provide health monitoring. If you ask Cloud Foundry to stand up ten instances of an application, it'll ensure that there are at least ten instances. Even if an instance should die, Cloud Foundry will restart it.

Most cloud platforms, including Pivotal Web Services and Pivotal Cloud Foundry, support *auto-scaling*. An auto-scaler monitors container-level information like RAM and CPU and, if necessary, adds capacity by launching new application instances. On Pivotal Web Services, you need only create a service instance of the type `app-autoscaler` and then bind it to the application. You'll need to configure it in the management panel on the Pivotal Web Services Console.

Example 12-32. Creating an auto-scaler service on Pivotal Web Services

```
cf marketplace
Getting services from marketplace in org platform-eng / space jos
OK

service          plans          description
..
app-autoscaler  bronze, gold  Scales bound applications in respon
..
```

There is still room for yet more advanced remediation. There are some inspiring examples out there, [like Netflix's Winston](#), [LinkedIn's Nurse](#) and [Facebook's FBAR](#), and the open-source [StackStorm](#). These tools make it easy to define pipelines composed of atoms of functionality that, when combined, solve a problem for you. These tools work in terms of well known input events, triggered by monitoring agents or sensors or other indicators deployed in the system. In a traditional architecture these events would trigger alerting, which is useful, but for some classifications of problems its possible to also trigger automatic remediation flows.

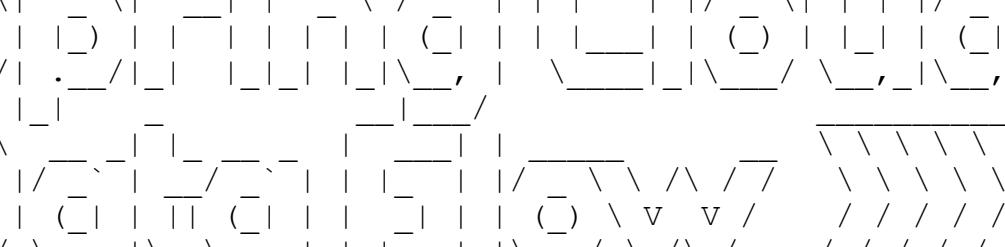
We encourage you to investigate some of the aforementioned approaches. Most of those solutions are rooted in environments that don't benefit from the layers of guarantees made by a platform like Cloud Foundry, though. With Cloud Foundry, it's easy to get away with less and use higher level building blocks for higher level problems. Consider integration technologies like Spring Integration. Spring Integration makes it dead simple to process events from different event sources with different components connected over event buses. Spring Cloud Data Flow builds upon Spring Integration, providing a Bash shell-like DSL that lets you compose arbitrary components and then orchestrate them. See the Data Integration chapter for details on Spring Cloud Data Flow.

In the source code for this chapter we've included a source - a Spring Cloud Data Flow component that can produce events - to monitor Cloud Foundry application metrics like CPU, RAM and hard disk usage. We've also included a sink - a Spring Cloud Data Flow component that accepts input and responds to it - to automatically scale an application's instance count up or down based on whether the incoming value is smaller or larger than a

configured threshold. So, for example, if the average CPU consumption on a deployed application should dip below a certain threshold, we could scale down. Once you've built these components, it's easy enough to then orchestrate them using the Spring Cloud Data Flow DSL.

In the following example, we'll first register the two types of components and then create a stream that takes the payload of the `cloudfoundry-metrics` component, a map with three keys (`CPU`, `RAM`, and `DISK`), passes it to a transformer that extracts the `CPU` value and then publishes it to the `cloudfoundry-autoscaler` component. These components require information about how to authenticate with Cloud Foundry, and those values can be passed in as -- properties or using any of the conventional Spring Boot configuration mechanisms like environment variables and the Spring Cloud Config Server. Note that the DSL statements are wrapped for the purposes of this book but they should be entered in one line.

Example 12-33. output from interacting with Spring Cloud Data Flow to automatically scale a service when a metric of interest meets a certain threshold.

```
→ remediation git:(master) ✘ ./dataflow-shell/target/dataflow-s  
  
1.0.0.RELEASE  
  
dataflow:>  
  
app import --uri http://bit.ly/stream-applications-rabbit-maven  
Successfully registered applications: [source.tcp, sink.jdbc, sou  
...  
  
dataflow:>app register --type sink --name cloudfoundry-autoscaler  
...
```

```
dataflow:>app register --type source --name cloudfoundry-metrics
...
dataflow:>stream create --name metrics-to-log --definition "cloud
scriptable-transform --scriptable-transformer.language=
cloudfoundry-autoscaler --cloudfoundry.autoscaler.sink.
Created and deployed new stream 'metrics-to-log'
...  
...
```

Next Steps

We've only begun to look at the possibilities in this chapter and if you're feeling overwhelmed, *good!* This subject is of a *critical* importance in a distributed system and failure to architect with observability in mind only tempts disaster. That said, Spring Boot and Spring Cloud are *meant* to address these concerns and come fully-loaded with support for these production-centric scenarios. You should consider extracting all these production-centric requirements into a separate Spring Boot auto-configuration that all new microservices benefit from. This way, you need only configure how you handle logs, metrics, tracing, etc., once and then simply add the appropriate auto-configuration to the classpath.

Chapter 13. The Application Centric Cloud

These days, the idea of a *PaaS* (platform-as-a-service) is a bit, pardon the pun, *cloudy*. Conceptually, a *cloud* is something that delivers services, on-demand. This definition is (ack..choke! *groan!*) a bit *nebulous*.

It's easiest to define clouds in terms of what they let you, the operator, create.

We all know what *SaaS* (software-as-a-service) is. It refers to end-user software deployed in a managed environment (like [SalesForce.com](#) or Google Apps) that is administered (if at all) *within* the application itself. Thus, a SaaS might make it easy to create more accounts on a hosted CRM, or more e-mail accounts on a hosted cloud-provider.

Until recently, this meant that a developer would describe in some declarative fashion via a CLI or a configuration file the resources an application needs. This configuration typically included parameters like how much RAM should be assigned, the HTTP routes to expose the application under, how many instances of an application should be run, and what backing services (databases, message queues) an application needs.

In order for the PaaS to run applications with these parameters, the PaaS had to understand the nature of the applications it runs. After all, most modern PaaSes treat everything as processes to be initialized, started, scaled, and stopped. They don't care that a Java application might need an Apache Tomcat, or that a PHP application might need an Apache HTTPD or Nginx installation.

Hopefully, this intuition about applications is an extension plane for developers, and not baked into the platform. Heroku and Cloud Foundry, for example, both use *buildpacks* to adapt given application artifacts to the lifecycle callbacks of the platform. Most PaaS take application artifacts as inputs. A PaaS like Cloud Foundry expects a `.jar` or `.war`, for example. The

default (overridable) Java buildpack does things like install the right supporting libraries, setup the right `PATH`, and start and stop the right infrastructure (like Apache Tomcat) and do so on the network port provided by the platform.

This approach works well: the rigidity creates a consistency in development and operations. Developers need only make their application behave appropriately with the buildpack (or override it!) and then reuse the recipe. As long as the PaaS is happy with the input applications, then almost everything required to move an application from a developer's machine to a production, or production-like, environment can be done with almost no administration. This makes operations happy because it frees them to focus on more important things, and it makes developers happy because they're not waiting for operations to catch up. (No more waiting days or weeks for operations to provision a machine and setup security!)

Portable Applications

Spring has always tried to make application portability as easy as possible. It supports good design patterns such as those prescribed by the [12 Factor App](#) manifesto.

As developers, we're used to being able to test applications in isolation, to validate the inputs into an application and validate the resulting behavior. We're used to reproducible builds; we're used to being able to throw away the build and - with the disciplined use of tags and so on - reproduce the same build. This isn't news.

Cattle

It took us developers a while to get to this place, but we did. In the last 8 years we've seen this rigor come to operations teams. Operations want reproducible infrastructure as much as we developers want reproducible builds. Consistency and reproducibility are even more important with disposable, ephemeral, cloud infrastructure. As Adrian Cockcroft (formerly at Netflix) reminds us: treat servers like cattle, not pets.

Things have come a long way. Developers and operations are - ideally at least - two highly integrated teams. Developers want consistency in the way their applications run, operations want consistency in their infrastructure.

Containerized Workloads

The next pattern we will look at is containerized workloads. Container deployments are an emerging pattern that have seen rapid adoption in the last few years. During Netflix's transition to become a cloud native company, they admitted to using Amazon Machine Images (AMI).

Note

Amazon Machine Images (AMI) are virtual server instances that have been customized and configured for re-use on EC2 (Elastic Cloud Compute) on AWS. Information from these customized instances are saved and used to launch new instances.

When deploying a new version of an application to AWS, Netflix would bake it into an AMI. The new AMI would then be used to spin up a cluster of the new version of the application. This is a common usage pattern with large cluster deployments on AWS. In this sense, AMIs are a container for our applications to be deployed to a cloud environment, allowing us to bundle our twelve-factor apps as a virtual appliance.

When most people think about containers today, they think Docker. Docker has largely provided us with a standard pattern for understanding what cloud native applications look like. I think it's important though to understand that Docker is not the end result for cloud native applications.

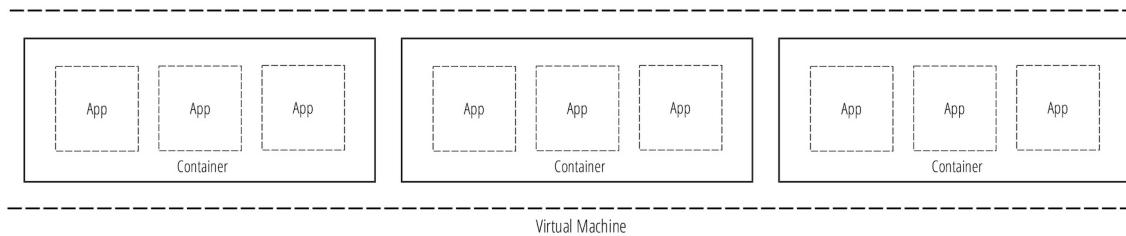


Figure 13-1. A virtual machine hosting multiple containerized applications

Containers give developers the freedom to build applications using the tools that they believe are best suited for solving a specific problem. Containers can take many forms. Core to the idea of containers is the idea of bundling applications with its dependencies and scheduling it to run on a virtual machine where it will only consume an allotted amount of system resources.

Scheduler

Schedulers are services that manage the logistics of container scheduling. Schedulers act as the operator that decide when, where, and how a container is provisioned with a cloud provider, such as Amazon EC2. The responsibility of a scheduler is to manage the logistics of moving containers around in the cloud and can be configured to elastically scale the number of virtual machines that host containers based on usage requirements. Schedulers help to give software operators a dial to control the cost of operating cloud native applications.

Composition and scheduling are important concepts when using containers or developing microservices. Composition and scheduling are how cloud native applications take advantage of [elastic scaling](#), a strategy that maximizes the benefits of using a cloud provider. We are able to package applications into a container that can be composed and scaled to increase operational efficiency.

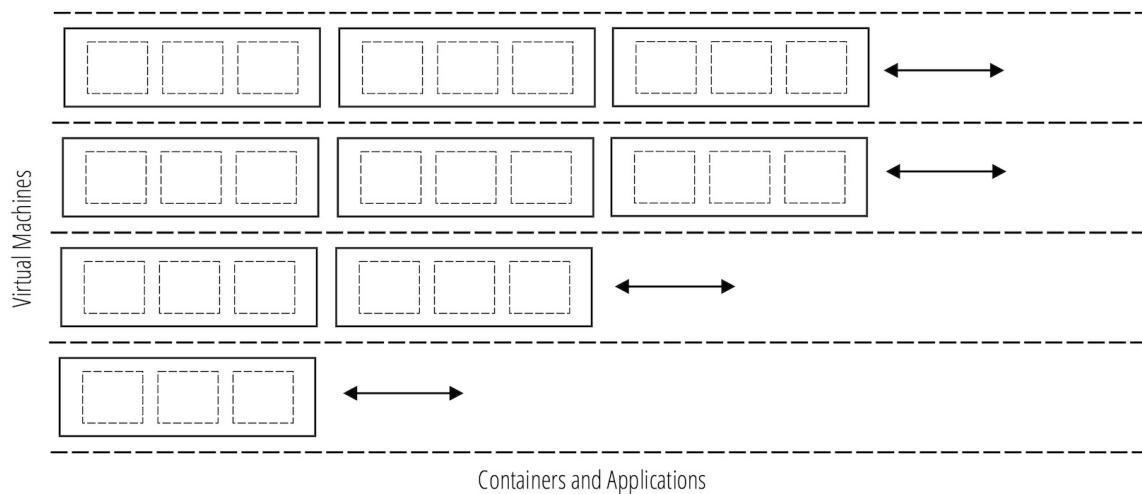


Figure 13-2. Containers are scheduled on virtual machines

The graphic above illustrates rows of 4 virtual machines that host a set of containers. Each container hosts a set of unique applications, isolated from other containers on the machine. The resources available to containers can be described by demarcating the resource priority for the container. Assigning a

priority to a container prevents resource contention between applications in separate containers on the same VM. One container may host a long-living web application in production, this container will be given higher priority over a non-production short-living application in another container.

One of the greatest benefits of cloud native application development is that it helps mitigate the cost of operating applications. With a cloud native platform, developers and operators shouldn't need to be concerned with manually addressing capacity concerns for applications. A cloud native application platform will optimize the cost of operation against a set of policies that a business can define.

Service Discovery

The language of how containers talk to each other on a cloud platform is called composition. The composition of cloud native applications that are scheduled on a cloud provider can be thought of as an automatic process for applications inside containers integrating with applications inside other containers. Composition focuses on how containers find each other after a scheduler has provisioned a new virtual machine and where it will run.

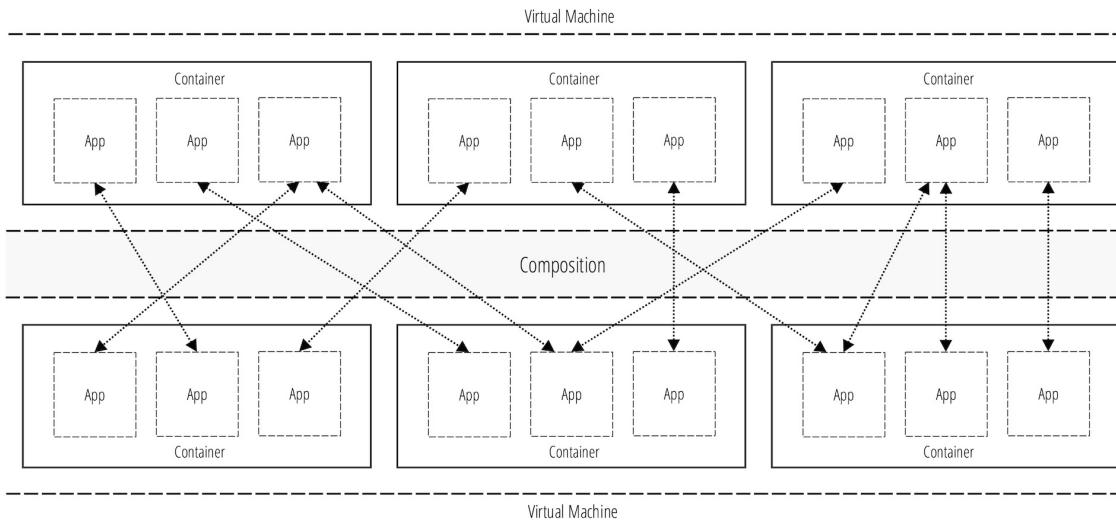


Figure 13-3. Composition of Cloud Native Applications

Container schedulers make it hard for applications to know in advance where to look for an application that is running in another container. Due to the logistics of the scheduler needing to provision virtual machines on-demand, when an application inside a container needs to talk to an application in another container, composition is the mechanism that enables it.

Tip

In the context of microservices, management APIs, such as a discovery service, provides applications a way to automatically compose themselves for integration.

A monolithic application's performance is dependent upon the computer it runs on. Because of this, the capacity of workloads by a monolithic application will depend on the chip-design of the computer. When scaling an application on a virtual machine, the point of scale depends on a single factor: the time it takes for an instruction cycle to be processed by the machine. This is largely dependent on the hardware of the computer. There is only so much a programmer can do to optimize the speed of a workload. Eventually you will run out of capacity and have to scale up the machine's virtualized hardware resources.

When we talk about availability for web applications, we are talking about the percentage of time that an application is available to its users. Increasing availability depends on the number machines that can balance the load of users actively requesting resources from a web application. The cost to operate a web application is then multiplied by the number of machines that are available to serve traffic to users.

If we were to decompose a monolithic web application into a set of smaller distributed web services, we can schedule and scale these applications much more efficiently on a set of virtual machines. In order to do this effectively, we need to address a set of concerns about the degree of complexity added by the communication between our distributed services.

When we scale out applications, the unit of scale is predicated on two factors: the speed of the network and the number of virtual machines that an application instance runs on. This approach provides opportunities to scale computing resources elastically using a scheduler, delivering cost savings for resource-hungry computing workflows that would otherwise be too expensive to be scaled up.

The adoption of cloud native application development is providing us opportunities to improve application performance, resiliency, availability, and many other factors when it comes to operating web applications at scale.

In this chapter we've talked about some of the common patterns that are requisite for optimizing how we operate cloud native applications. Now we will talk about the abstractions in today's popular open source tooling that we will use to build cloud native JVM applications in this book. At the center of today's Java ecosystem we have a set of popular tools from Spring that companies like Netflix are using to build their cloud native applications.

The Application Framework

Building cloud native applications requires a framework that is capable of responding to a wide-range of concerns for building ultra-scalable applications that do not sacrifice performance or availability.

When it comes to building cloud native applications using Java, the open source projects *Spring Boot* and *Spring Cloud* are leading the way.

Spring Boot

Spring Boot aims to provide developers with the quickest way to get started with building production-ready cloud native applications. Spring Boot applications can be customized to automatically configure desired components of the Spring Framework as a set of starter projects defined in your application's dependencies.

For those of you who are new to Spring Boot, the name of the project means exactly what it says. You get all of the best things of the Spring Framework and ecosystem of projects, tuned to perfection, with minimal configuration, all ready for production.

As we start building multiple cloud native Spring Boot applications that are dependent on one another, we quickly find that we will have entered into the realm of building *microservices*. We will take a deeper look at building microservices as Spring Boot applications later on in this book.

Tip

Martin Fowler popularized the term microservice on [his website](#) as “*an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.*”

As we start building many connected microservices, we enter into the world of building and operating distributed systems. As Netflix continued to build out a cloud native application architecture, they too found that they were building and operating distributed systems. The Netflix team would have to break apart its monolith using a newer practice of software development called microservices. To be successful at building this kind of new architecture, the team at Netflix would need to also develop new tooling to make applications fault tolerant and resilient to failures. As Netflix fashioned this tooling to build resilient distributed systems, it would eventually find its way into the Spring ecosystem as a project named *Spring Cloud*.

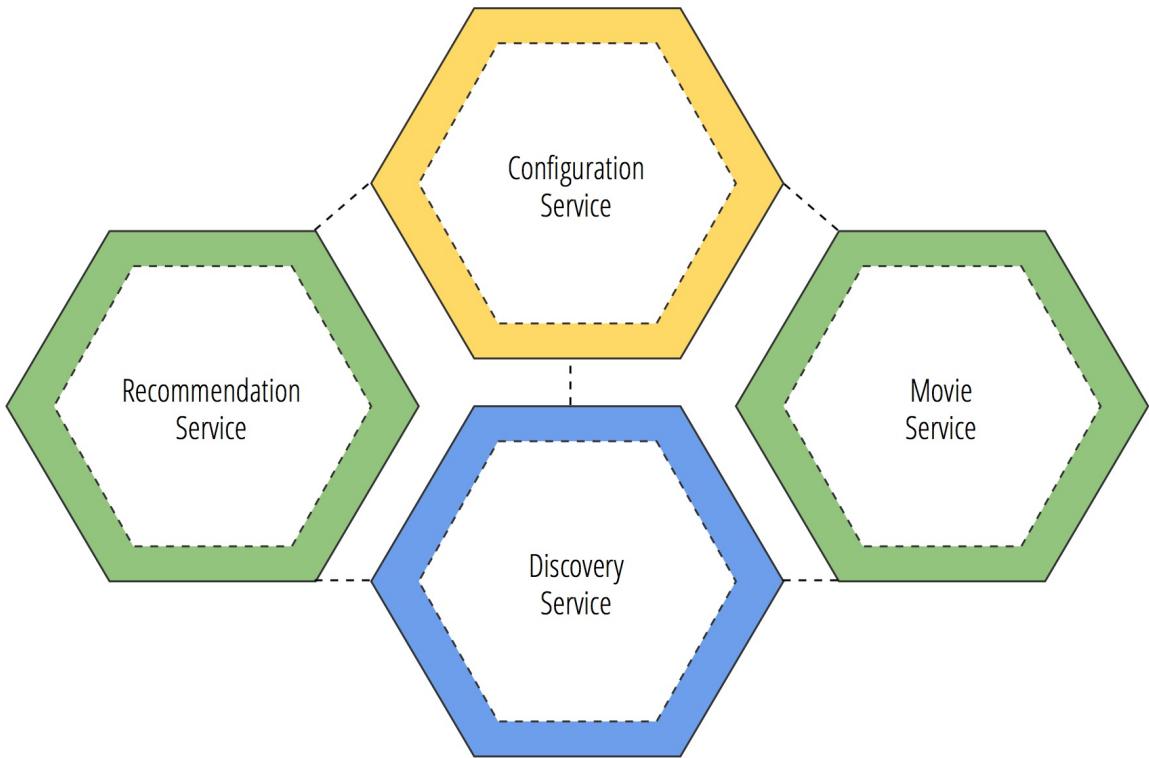
Spring Cloud

Spring Cloud is a collection of tools that provides solutions to some of the commonly encountered patterns when building distributed systems. If you're familiar with building applications with Spring Framework, Spring Cloud builds upon some of its common building blocks.

Among the solutions provided by Spring Cloud, you will find tools for the following problems:

- Configuration management
- Service discovery
- Circuit breakers
- Distributed tracing

Each Spring Cloud application in an architecture has a dedicated purpose and serves a specific role. When building Spring Cloud applications using Spring Boot, there are a few primary concerns to deal with first. The two Spring Cloud applications you will likely want to create first are a *Configuration Service* and a *Discovery Service*.



The graphic above illustrates four Spring Boot applications depicted as microservices. Each Spring Boot application plays a specific role in this architecture, with the connections between them indicating a service dependency.

Note

A Spring Cloud application extends a Spring Boot application and will provide a set of minimal features for applications to find each other, as well as preventing cascading failures using resiliency patterns such as circuit breakers.

The configuration service sits at the top, in yellow, and is depended on by the other Spring Boot microservices. The discovery service sits at the bottom, in blue, and also is depended upon by the other Spring Boot microservices.

In green, we have two Spring Boot microservices that deal with a part of the domain data of the distributed cloud native application.

Configuration Service

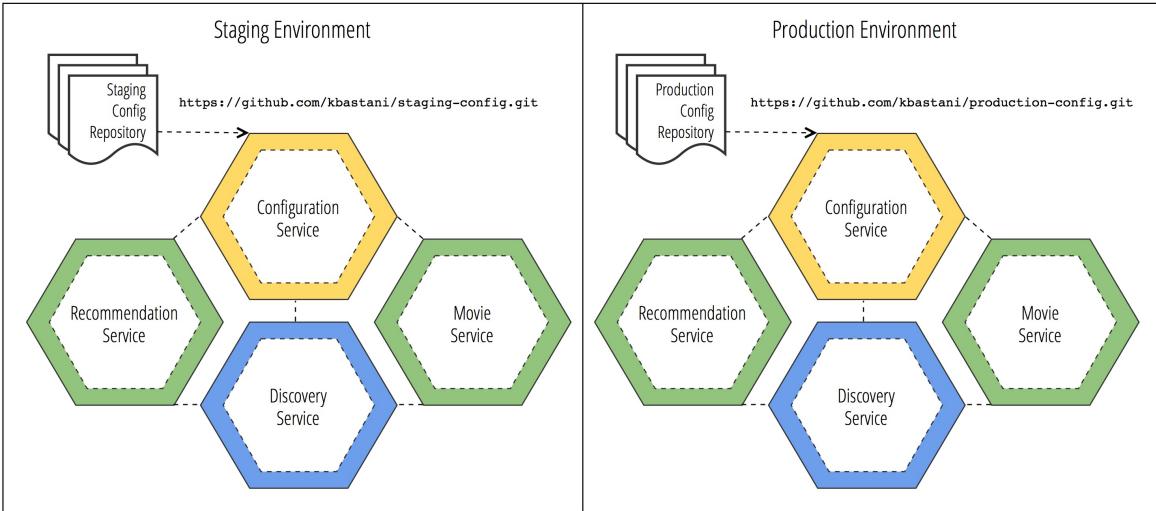
The configuration service is a vital component of any microservices architecture. Based on the principles of twelve-factor applications, the configurations for your microservice should be stored in the environment and not in the project.

Tip

When we use the word *microservice* in this book, we are talking about a type of Spring Boot application that has been customized with a web starter project. The web starter project gives a Spring Boot application the minimal functionality to be considered a microservice.

The configuration service is essential because it is responsible for handling the customized application properties for each microservice in an environment. The configuration service will provide these properties to other services through a simple point-to-point service call. The advantages of this are multi-purpose.

Let's assume that we have multiple deployment environments. If we have a staging environment and a production environment, configurations for those environments will be different. A configuration service might have a dedicated Git repository for the configurations of that environment. None of the other environments will be able to access this configuration, it is available only to the configuration service running in that environment.

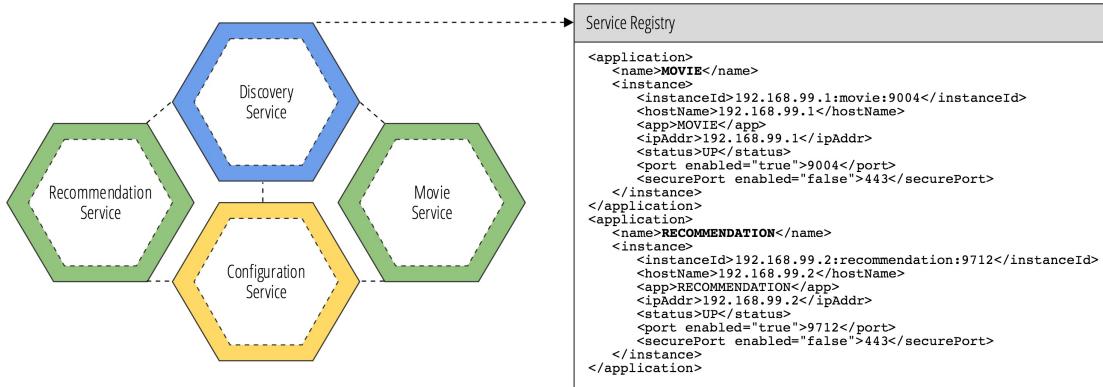


When the configuration service starts up, it will reference the path to those configuration files and begin to serve them up to the microservices that request those configurations. Each microservice can have their configuration file customized to the specifics of the environment that it is running in. In doing this, the configuration is both externalized and centralized, existing in one place that is version-controlled. By doing this, changes to configurations can be made without having to restart the microservice that a change is intended for.

With management endpoints available from Spring Cloud, you can make a configuration change in the environment and signal a refresh to the discovery service. This refresh signal will force all consumers to fetch the new configurations and to start using them.

Discovery Service

Each Spring Boot application has the ability to take on a certain role in a microservices architecture. One of the central themes behind cloud native applications and microservices is *service discovery*. This takes on the form of a service registry where applications are able to *check-in* and register information about how they can be contacted.



Spring Boot applications that register with a discovery service are able to download a registry that contains information about other service instances. This service registry will contain details of each healthy application instance that has previously registered with the discovery service.

Note

A registry that is managed by a discovery service is often compared to a *phone book*. Long ago, before paper phone books were made obsolete by smart phones and the internet, people needed to find each other's telephone numbers using a telephone directory. Subscribers would provide their full name, location, and telephone numbers, so that they could be reached by other subscribers in their neighborhood.

The information in the service registry can not only be used to communicate between services, but the registry can also be used to perform client-side load balancing. Client-side load balancing is a technique where consuming services can load balance their requests to a set of service instances within a cluster.

The service instances are grouped together in a service registry, indicating that they are members of a cluster of identical services. The consuming service can then use a load balancing strategy to choose which service in the cluster that it should contact for each request.

Circuit Breakers

The *Circuit Breaker* is a resiliency pattern popularized by Michael Nygard in the book *Release It!*. Nygard introduced this pattern as a systems metaphor into an electric circuit breaker that is installed in a home to prevent electrical fires from an overloaded circuit.

Nygard stated in his book that “the circuit breaker exists to allow one subsystem (an electrical circuit) to fail (excessive current draw, possibly from a short-circuit) without destroying the entire system (the house)”.

[Figure 13-4](#) shows each possible state of a circuit breaker, depicted as a rounded box. The directed relationships between the rounded boxes indicate the result of a transition from one state to another state.

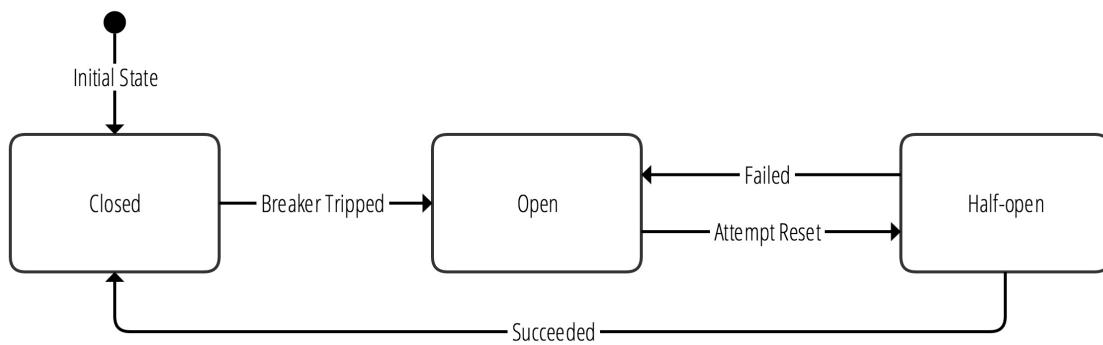


Figure 13-4. state machine diagram of a circuit breaker

Starting from the initial state of the system, we can see that the circuit breaker starts in a *closed* state. When a circuit breaker is in a closed state, traffic to a web service is functioning as normal, with responses being returned at a manageable rate by the service.

Let's now assume that the web service has started to return errors to consumers, indicating that the service is either overloaded or a downstream dependency of the service has failed. The web service will then transition the state of the circuit breaker from a *closed* state to an *open* state. When the breaker has tripped into an *open* state, it will fall back to the designated functionality for this condition. In most cases when using a circuit breaker pattern, the fallback response will closely resemble the response that a client is expecting, with perhaps a default set of properties or an empty list of

results.

The circuit breaker will then attempt to try some of the incoming requests while in an *open* state. These sample requests will check to see whether or not functionality has been restored to the offending service call and that responses are returning without errors. In this case, the service will transition from an *open* state to a *half open* state. If the result of the sample requests succeed, without error, then the service's circuit breaker will be reset and transition back to the initial state. The state transition will move from the *half open* state to the *closed* state, and the behavior of the service will settle back into serving traffic normally to any service that requests its resources.

Each Spring Cloud application that has been configured as a *Hystrix client*, will automatically take advantage of the circuit breaker pattern, among other resiliency patterns that come with Hystrix.

Note

Hystrix is a Netflix OSS project that focuses on preventing cascading failures, monitoring latency, and enabling resiliency patterns in distributed systems. Hystrix integrates with Spring Boot applications as a part of Spring Cloud, and also provides a dashboard for monitoring clients.

Later in this book we take a deeper look into resiliency patterns that are used by *Hystrix* and how to take advantage of using them in your Spring Cloud enabled applications.

Chapter 14. Continuous Delivery

Continuous delivery is a simple concept with profound implications: use automation to reduce the cost, time and risk of delivering incremental (“small batches”) changes to users. Jez Humble and David Farley most famously chronicled the idea of continuous delivery in their classic tome, [*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation \(Addison-Wesley Signature Series \(Fowler\)\) 1st Edition.*](#)

There are [five principles at the heart of continuous delivery](#):

- **Build quality in:** use automated testing to find errors and produce immediate feedback. It’s better to find 90% of the errors on every `git push` through automation than to find 100% after months of manual testing.
- **Work in small batches:** every `git push` goes as far towards release as possible. The batch goes through testing which provides valuable feedback, quickly, instead of after a long accrual of work.
- **Computers perform repetitive tasks, people solve problems:** it’s inhumane (and error prone!) to make humans do mindless, repetitive work. Let the machines do it, instead.
- **Relentlessly pursue continuous improvement:** the process of transformation and improvement never stops.
- **Everyone is responsible:** work towards organizational goals, rather than optimizing for team or departments. “You build it, you run it.”

Work in an organization moves (hopefully!) from concept to production. As it is being developed, it passes from product management, user experience, developers, testers and various administrators (SAN, network, database and security administrators). Developing, testing, integrating and finally deploying code will take a long time if the scope of work is large. This time represents organizational waste; work remains undelivered to the customer -

the customer who is the *only* arbiter of the fitness of the work to purpose. The longer an organization waits to deliver software, the bigger the risk that the customer will not like the work, or that the customer will not use the work.

An organization can reduce this risk by reducing the scope (batch size) of work. Deliver features, not releases. A feature is meant to describe something that can be developed, tested and integrated in a *very small* batch of time - at most once a day. What does moving to a smaller batch imply for our workflow? What has to change to meet the goal of potentially delivering builds to production (the customer) at least once a day? What has to change so that what's on `master` is always releasable?

All code must live on master. If all code must be integrated and tested at minimum once a day, then anything that defers the integration of that code - like feature branches - introduces risk. In a continuous delivery pipeline, new features are integrated into master as soon as is feasible. Developers are expected to run all the tests (new and old) to confirm that everything works. Then, a developer should merge all changes from master into the local working copy and integrate the code, confirming everything works by re-running all the tests. Finally, the integrated code should be pushed to master so that others may integrate. If any developer breaks master, then it must be a priority to either fix master (within minutes, ideally) or revert the change until the build can be fixed.

The code needs to be fit for production, at *all times*. “Done means done.” Does the code require instrumentation to support observability? Changes for security? That certain libraries or frameworks be configured a certain way? There are a lot of non-functional requirements that we must meet to move code into production. I think most of us know about the dreaded organizational wiki page with “500 easy steps to production”! These things must be addressed, yet require a load of work if left to manual processes. You can use Spring Boot’s auto-configuration to codify these concerns and provide useful, and customizable, default behavior for things like framework integrations, auditing and observability, and security. Simply add a Spring Boot auto-configuration to the classpath of a Spring Boot application to benefit.

Version-control *everything*. If all code lives on master, and every build

could go to production, then everything required to test and deploy to production must be on master, as well. Here, we can benefit from the principles of immutable infrastructure and automated technologies like Cloud Foundry so that creating a production-like environment is a matter of version control-friendly configuration.

Testing must be as automated as possible. There is no time for exhaustive regression testing by a human. There are three types of automated tests: unit and integration tests, functional acceptance tests (end to end tests run in a production-like environment) and non-functional acceptance tests (performance, scaling). Humans should do exploratory testing, usability and showcases - where they take the users point of view - and the machines should handle the automated quality checks. Each type of test takes longer and longer periods of time. Ideally, you should be able to run all the unit and integration tests very quickly, in a matter of minutes, for example. Automated acceptance tests might take considerably longer for a complex system and this is expected. Run the faster tests and then the progressively more exhaustive, slower tests so as to fail-fast if possible.

Create a deployment pipeline to automate all these tests. A deployment pipeline should move work from the deliver team (developers) to version control, then to build and unit tests, then to automated acceptance tests, and finally to user acceptance tests. At this point, if all the tests have passed, the code should be releasable. It may be that the work gets stopped at a certain stage in which case the error should drive feedback to the delivery team where they can hopefully change the build or revert the contribution.

Prioritize keeping the code releasable. Every build is a potential release candidate. The job of the deployment pipeline is to detect and reject bad builds. If the code is releasable at any time, then we can drop traditional notions of lengthy integration and testing phases as they're built in to the concept of "done."

The developer workflow must explicitly support keeping master releasable. A developer should develop a feature and then run the full test suite, locally. Then, the developer should fetch and merge all upstream changes into their local repository, confirming that the full test suite still passes. Then, the developer should push the changes to master where it'll pass through the

pipeline. If anything goes wrong in the pipeline, then the developer must immediately fix the problem or revert the change.

The movement of changes should be fully controlled. A pipeline should surface information about the flow of changes through the pipeline. This has a tangent benefit that it also satisfies any auditing requirements most organizations will have.

Architect for small batches if tests require long lengths of time to complete, then perhaps the scope of the batch is still too large. If it's possible to identify the seams of bounded contexts within the larger application, those seams are natural candidates for being made into microservices.

Let's make these principles concrete with an example!

Start Here

We'll look at a simple example, a *micro* microservice; one so trivial we don't really have to focus on its implementation. It will expose a REST endpoint, /hi. Go to [the Spring Initializr](#) and select Web, then click Generate. Your browser will download a .zip archive that contains a simple project. Unzip it and import the Maven project into your favorite IDE. Modify DemoApplication to look something like this:

Example 14-1. the src/main/java/demo/DemoApplication.java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;

@SpringBootApplication
@RestController
public class DemoApplication {

    @RequestMapping(method = RequestMethod.GET, value = "/hi")
    Map<String, String> hello() {
        return Collections.singletonMap("greeting", "Hell
    }

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

You can confirm this works by running it, naturally. Simple run the main method in the application and then test that when you issue curl http://127.0.0.1:8080/hi, you get a JSON response, like this:

Example 14-2. the response from our simple service

```
{"greeting":"Hello, world!"}
```

Every Build is a Release Candidate

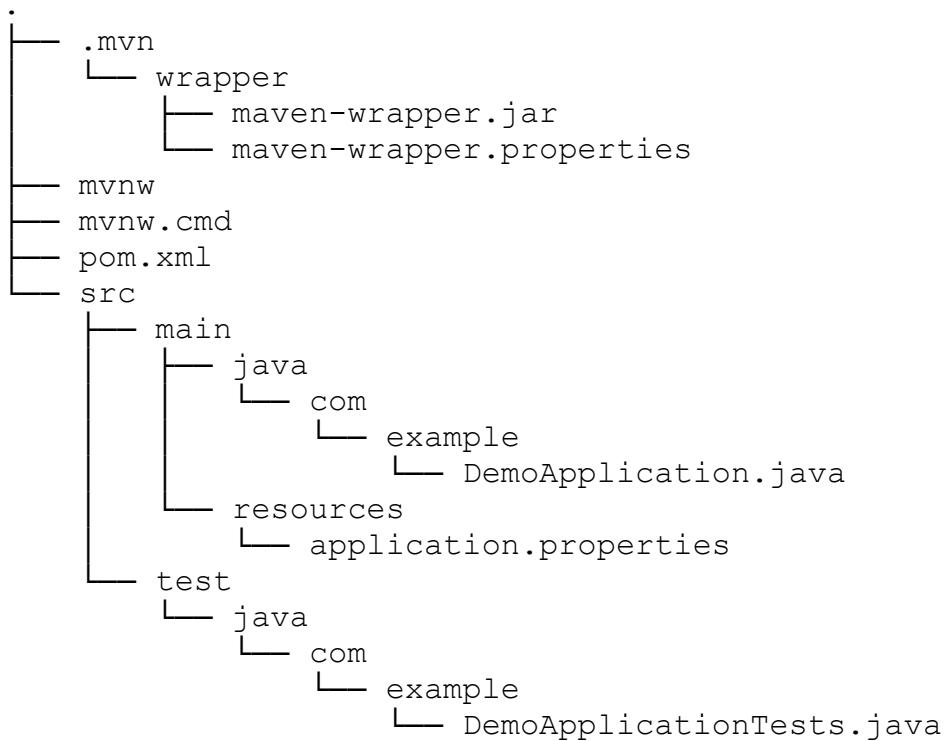
We need to rethink the way we handle artifact versions in the world of continuous delivery..

```
<build.number>0.0.1-SNAPSHOT</build.number>
```

Version Control Everything

Here's what it looks like when unzipped:

Example 14-3. The newly unzipped project



The usual suspects are all here: a `src/{test,main}/{java,resources}` tree, a Maven `pom.xml` build, and, very helpfully, a Maven wrapper integration (`.mvn`, `mvnw`, and `mvnw.cmd`). From the website:

The Maven Wrapper is an easy way to ensure a user of your Maven build has everything necessary to run your Maven build. Why might this be necessary? Maven to date has been very stable for users, is available on most systems or is easy to procure: but with many of the recent changes in Maven it will be easier for users to have a fully encapsulated build setup provided by the project. With the Maven Wrapper this is very easy to do and it's a great idea borrowed from Gradle.

Chapter 15. Edge Services

For every service you build there is a client, something that will consume that service. These clients are myriad; HTML 5 clients, Android clients, iOS clients, PlayStations or XBoxes, Smart TVs, and indeed almost *anything* else these days has an IP address and could act as a client to your service. Clients often speak different protocols, have different security requirements, require different payload encodings and work with different bandwidths. These sorts of concerns are best handled centrally through a proxy layer or an API gateway layer.

Some clients require *adapted views* of the data coming from the services with which they interact; they may need the data synthesized from existing data, or processed, to support the UI requirements of a specific client.

The cost of retrofitting every microservice in a system with a non-trivial number of microservices to accommodate each new client is prohibitive. Our goal here is to stay agile, to control our destiny. Instead, handle these concerns in an intermediary service, called an *edge service*. Edge services are a great way to combat the limitations of the one-size-fits-all REST API. Clients have many different dimensions. Memory capacity or processing power which may affect how much content it can manage at a given time. Requirements for specific types of markup or content. Document models optimized for different clients - some hierarchical and some flat. Screen real estate may impact the content loaded. Document delivery may be more efficient streaming or chunked. User interactions may change the responses required, as well; they could influence the metadata fields, delivery method, interaction model, etc.

Edge services are a sort of doorway for requests coming into the application. They are in an enviable position to handle client-specific concerns as well as cross-service concerns like authentication and authorization, rate-limiting, routing (for more on this, see the chapter on [Routing]()), filtering, API translation, client-adapter scripts, proxying and more.

Greetings

In this chapter we'll look at addressing hte concerns of edge services, but we'll do so in terms of a service called the `greetings-service` and a Eureka servie registry to make discovering and working with that service easier.

First, let's standup a quick Netflix Eureka instance, the `service-registry` module.

Example 15-1. setup a simple Netflix Eureka service in the `service-registry` module

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEure
❶
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

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

❶

the `@EnableEurekaServer` annotation configures a bare-bones instance of the Eureka service registry.

Example 15-2. `application.properties` in the `service-registry` module

```
spring.application.name = eureka-service
server.port = ${PORT:8761}

❶
eureka.client.register-with-eureka = false
```

```
eureka.client.fetch-registry = false  
②  
eureka.server.enable-self-preservation = false
```

①

tell Eureka not to register with itself

②

ensure that Eureka doesn't bother to aggressively cache cluster information if it detects nodes are not responding within a certain threshold. This is a convenience during development but you might disable it in production.

Run the `service-registry` and it'll spin up on port 8761. Then we'll need a simple REST API with which to work - a `greetings-service` that responds with a salutation on invocation. The service is fairly unremarkable except that it participates in service registration and discovery, making itself discoverable through Netflix Eureka.

Example 15-3. `GreetingsServiceApplication.java` in the `greetings-service` module.

```
package greetings;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscovery  
  
①  
@EnableDiscoveryClient  
@SpringBootApplication  
public class GreetingsServiceApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(GreetingsServiceApplication.class,  
    }  
}
```

①

participate in service registration and discovery with the Spring Cloud

`DiscoveryClient` abstraction.

We have many different permutations of the same REST API in this chapter, so we've used Spring profiles to conditionally enable or disable them. The default `greetings-service`, the one that will run absent any other specified profile, looks like this:

Example 15-4. `DefaultGreetingsRestController.java`

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;

@Profile("default")
@RestController
@RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
class DefaultGreetingsRestController {

    @RequestMapping
    Map<String, String> hi(@PathVariable String name) {
        return Collections.singletonMap("greeting", "Hello, " + n
    }
}
```

We'll revisit this example a few times in this chapter. Run it and it will register itself in Eureka as `greetings-service`, and spin up on port 8081 by default.

A Simple Edge Service

Let's stand up a simple client API - an *edge service* - that will act as a go-between the outside world and our downstream services. this endpoint needs only contact the downstream service. We want to act as a client. In this section we'll look at a few different ways to do that.

The entry class looks simple enough: we'll use service registration and discovery to activate Eureka.

Example 15-5. `GreetingsClientApplication.java`

```
package greetings;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscovery
① @EnableDiscoveryClient
@SpringBootApplication
public class GreetingsClientApplication {

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

①

activate service registration and discovery

then we'll build a simple endpoint that calls the downstream service using REST. We're using the type-token pattern to make short work of coercing the resulting JSON into a type we can work with.

Example 15-6. `RestTemplateGreetingsClientApiGateway.java`

```
package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Profile;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

@Profile("default")
@RestController
@RequestMapping("/api")
class RestTemplateGreetingsClientApiGateway {

    private final RestTemplate restTemplate;

    @Autowired
    RestTemplateGreetingsClientApiGateway(@LoadBalanced RestTempl
        this.restTemplate = restTemplate;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/resttem
    Map<String, String> restTemplate(@PathVariable String name) {

        ParameterizedTypeReference<Map<String, String>> type =
            new ParameterizedTypeReference<Map<String, String
        };

        return this.restTemplate.exchange(
            "http://greetings-service/greet/{name}",
            HttpMethod.GET, null, type, name).getBody();
    }
}
```

Netflix Feign

when working with APIs it become sueful to build clients for the different APIs. differnet orgs have different stances on this. Amazon says to biuld clean rom implementtions. Netflix has Netfli Feign.

We can activate it by adding `@EnableFeignClients` and `spring-cloud-starter-feign` to the classpath. Heres a simple client built as an interface. Make sure you start the `greetings-client` with the `feign` Spring profile.

Example 15-7. `GreetingsClient.java`

```
package greetings;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Map;

@FeignClient(serviceId = "greetings-service")
interface GreetingsClient {

    @RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
    Map<String, String> greet(@PathVariable("name") String name);
}
```

then we can use it in our client API.

Example 15-8. `FeignGreetingsClientApiGateway.java`

```
package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.bind.annotation.RestController;
import java.util.Map;

@Profile("feign")
@RestController
@RequestMapping("/api")
class FeignGreetingsClientApiGateway {

    private final GreetingsClient greetingsClient;

    @Autowired
    FeignGreetingsClientApiGateway(GreetingsClient greetingsClient)
        this.greetingsClient = greetingsClient;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/feign/{name}")
    Map<String, String> feign(@PathVariable String name) {
        return this.greetingsClient.greet(name);
    }

}
```

Reactive Programming

Use Reactor 2! (CompletableFuture, RxJava, Reactor?)

- <http://www.nurkiewicz.com/2015/07/server-sent-events-with-rxjava-and.html>
- <http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html>
- <https://objectpartners.com/2014/11/18/udp-server-with-spring-boot-and-reactor/>

Proxies with Netflix Zuul

a lot of concerns are easily handled in a generic way. if all u wanna do is interpose filter-like concerns in between the clients and the downstream services, then just proxy the requests. we can use a microproxy, called Netflix Zuul, to make short work of this. Zuul is a small, embeddable microproxy.

Zuul's architect, Mikey Cohen, gave a nice talk [at SpringOne Platform 2016 that introduces some of Zuul's applications](#). It's used to serve most of the front-ends for Netflix.com and more than 1000 device types. It is the adapter layer that handles hundreds of permutations of protocols and device versions. It's fronted by more than 50 AWS elastic-loadbalancers in Netflix's environment and handles tens of billions of requests per day across three AWS regions. It's deployed in over 20 production Zuul clusters fronting more than ten origin systems for Netflix.com. Zuul is, in short, a workhorse component.

Zuul filters are deeply rooted in production. They contain dynamic routing logic, handle load-balancing and service protection, and provide tools to debug and analyse problem pathways. A Zuul gateway can be a quality assurance tool in that it provides a single place to observe the flow of data through a system.

Zuul filters can be used to create a top level context for requests into a system, handling concerns like geolocation and cookie and token decryption. They can address cross-cutting concerns like authentication. They can be used in request or response normalization, handling device specific weirdness like chunked encoding requirements, header truncations, URL encoding fixes, etc. They are an ideal place to do targeted routing, perhaps in order to isolate specific albeit faulty requests. They can be used to handle traffic shaping, do global routing world wide, handle dead-node and zone-failover logic, and detect and prevent attacks.

All of these possibilities are distinctly generic - they apply to all manner of services. Zuul is *not* the right place to put business logic specific to a

particular service. Keep business logic out of the edge-tier.

Zuul filters are not the same as a traditional Servlet Filter. It's assumed that you're filtering the results of *proxied* requests and not, particularly, the requests to the edge service itself.

Zuul filters are an ideal place to address routing from the client to downstream services. If your services are registered in a service registry for which there is a Spring Cloud DiscoveryClient implementation configured, then it's a simple matter of adding a @EnableZuulProxy to your application code and Spring Cloud, working with the DiscoveryClient, will setup routes for you. You can programmatically interrogate those routes using the RouteLocator, as we do in this example.

Example 15-9. zuulConfiguration.java

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.cloud.netflix.zuul.filters.RouteLocato
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableZuulProxy
class ZuulConfiguration {

    @Bean
    CommandLineRunner commandLineRunner(RouteLocator routeLocator
        Log log = LogFactory.getLog(getClass());
        return args -> routeLocator.getRoutes()
            .forEach(r -> log.info(String.format("%s (%s) %s"
                r.getId(), r.getLocation(), r.getFullPath
            )
        }
    }
}
```



the RouteLocator is an abstraction.

Zuul has setup convenient routes for us based on the service IDs, so assuming we've already stood up the `service-registry` and the `greetings-service` then we should see reflected in the output of the previous example that there's a service, `greetings-service`, available for consumption through the proxy. The `greetings-client` is supposed to start at `http://localhost:8082`, so you could invoke the greeting service at `http://localhost:8082/greetings-service/greet/World` where the context-path, `greetings-service`, is drawn from the service ID in the registry. You can also setup arbitrary routes, e.g:

<http://joshsmacbook-pro.local:8082/hi/greet/Jane>

Example 15-10. `zuulConfiguration.java`

```
zuul.routes.hi.path = /lets/** ❶  
❷  
zuul.routes.hi.serviceId = greetings-service
```

❶

the path on the edge service that should be mapped to..

❷

a service (or, alternatively, a full URL specified *instead* of `serviceId` as the `.location`)

now the service is available as `http://localhost:8082/lets/greet/World`. These routes are an ideal thing to keep in the Spring Cloud Config Server where they can be updated later on without restarting the edge service. You can force Zuul to dynamically reload these routes. One way to achieve this is to simply invoke the Spring Cloud `refresh` endpoint that we discussed in the chapter on Configuration. Simply change the configuration in the Spring Cloud Config Server, commit the changes (to Subversion or Git) and then send an empty HTTP POST (e.g.: `curl -d{}`

`http://localhost:8082/refresh`) to the `refresh` endpoint on or to the Spring Cloud event bus (`/bus/refresh`). This will result in the publication of a `RefreshScopeRefreshedEvent` which components in the Zuul machinery

will respond to by reconfiguring themselves.

Alternatively, you can interact with the `routes` Actuator endpoint that using Zuul automatically contributes for you. Routes will return the configured routes if you send an HTTP `GET` and will trigger a refresh of the configuration if you send an HTTP `POST`. This is an alternative to using the `refresh` endpoint because it applies *only* to the Zuul routes, and not to any other (potentially heavier-weight) infrastructure that may be refreshed.

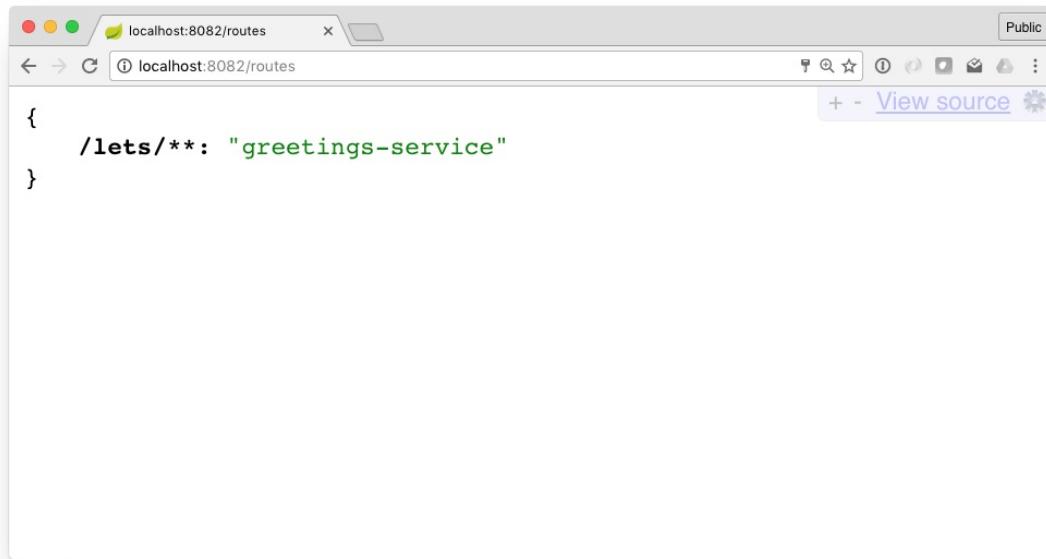


Figure 15-1. the Zuul routes from the `routes` endpoint

Zuul will also invalidate and reconfigure existing routes based on `HeartbeatEvent` events being published from the `DiscoveryClient`. Thus, if a node should de-register from Eureka (or any other service registry supported through the `DiscoveryClient`) then Zuul will remove the route from its configuration.

Naturally, if you have some sort of internal state that depends on these routes you can listen for these events as well.

Example 15-11. `RoutesListener.java`

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient
import org.springframework.cloud.client.discovery.event.Heartbeat
import org.springframework.cloud.netflix.zuul.RoutesRefreshedEven
import org.springframework.cloud.netflix.zuul.filters.RouteLocato
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
class RoutesListener {

    private final RouteLocator routeLocator;
    private final DiscoveryClient discoveryClient;

    private Log logger = LogFactory.getLog(getClass());

    ①
    @EventListener(HeartbeatEvent.class)
    public void onHeartbeatEvent(HeartbeatEvent event) {
        this.logger.info("onHeartbeatEvent()");
        this.discoveryClient.getServices()
            .stream().map(x -> " " + x)
            .forEach(this.logger::info);
    }

    ②
    @EventListener(RoutesRefreshedEvent.class)
    public void onRoutesRefreshedEvent(RoutesRefreshedEvent event
        this.logger.info("onRoutesRefreshedEvent()");
        this.routeLocator.getRoutes()
            .stream().map(x -> " " + x)
            .forEach(this.logger::info);
    }

    @Autowired
    public RoutesListener(DiscoveryClient dc, RouteLocator rl) {
        this.routeLocator = rl;
        this.discoveryClient = dc;
    }
}
```

①

listen to events being published by the `DiscoveryClient` machinery

②

listen to events being published by the Zuul routing machinery, specifically

Zuul, at its core, is a proxy but this makes the Zuul layer an ideal place to insert cross-cutting concerns that apply to all downstream services. You can do this using classic `javax.servlet.Filter` implementations, of course. Let's look at a simple `Filter` that exposes access-control headers to allow cross-origin request scripting so that other JavaScript clients can call this service from another origin. It works by looking at the incoming request and matching it against registered nodes in the Eureka registry. The client requests could be manually registered in the registry using a sidecar [like Netflix Prana](#) and the assets for the JavaScript and HTML5 application served on a content delivery network (CDN).

Example 15-12. `CorsFilter.java` - you'll need to start the application with the `cors` Spring profile activated

```
package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient
import org.springframework.cloud.client.discovery.event.Heartbeat
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.net.URI;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
```

```
@Profile("cors")
@Component
class CorsFilter implements Filter {

    private final Map<String, List<ServiceInstance>> catalog = ne

    private final DiscoveryClient discoveryClient;

    ①
    @Autowired
    public CorsFilter(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
        this.refreshCatalog();
    }

    ②
    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
        HttpServletResponse response = HttpServletResponse.class.
        HttpServletRequest request = HttpServletRequest.class.cas
        String originHeaderValue = request.getHeader(HttpHeaders.
        boolean clientAllowed = isClientAllowed(originHeaderValue

    ③
    if (clientAllowed) {
        response.setHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_O
    }
    chain.doFilter(req, res);
}

③
private boolean isClientAllowed(String origin) {
    if (StringUtils.hasText(origin)) {
        URI originUri = URI.create(origin);
        String match = originUri.getHost() + ':' + originUri.
        return this.catalog
            .keySet()
            .stream()
            .anyMatch(
                serviceId ->
                    this.catalog.get(serviceId)
                        .stream()
                        .map(si -> si.getHost() + ':' + si.ge
                        .anyMatch(hp -> hp.equalsIgnoreCase(m
            )
    }
    return false;
}
```

```

④
@EventListener(HeartbeatEvent.class)
public void onHeartbeatEvent(HeartbeatEvent e) {
    this.refreshCatalog();
}

// we don't want to constantly hit the registry, so proactive
private void refreshCatalog() {
    discoveryClient.getServices()
        .forEach(svc -> this.catalog.put(svc, this.discov
}

@Override
public void init(FilterConfig filterConfig) throws ServletException
}

@Override
public void destroy() {
}
}

```

①

we'll use the Spring Cloud `DiscoveryClient` to interrogate the service topology

②

the `doFilter` method for a standard Servlet `Filter` is as you'd expect..

③

it sets the access-control headers if the client is determined to be within a known set of services

④

proactively invalidate the local cache and then cache the set of registered services whenever the `DiscoveryClient` publishes a heartbeat event

Once that's done, we need a static HTML5 application to try things out. This application - in the `html5-client` module - leverages service registration and discovery through the Spring Cloud `DiscoveryClient` abstraction. It exposes

an endpoint, `/greetings-client-uri`, which in turn returns a valid URI for an instance of the `greetings-client` service, running on another node. We use this to then call that service with a cross-origin request.

Example 15-13. `Html5Client.java`

```
package client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;

@EnableDiscoveryClient
@SpringBootApplication
@RestController
public class Html5Client {

    private final DiscoveryClient discoveryClient;

    @Autowired
    public Html5Client(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
    }

    public static void main(String[] args) {
        SpringApplication.run(Html5Client.class, args);
    }

    ①
    @RequestMapping(value = "/greetings-client-uri",
                    method = RequestMethod.GET,
                    produces = MediaType.APPLICATION_JSON_VALUE)
    Map<String, String> greetingsClientURI() throws Exception {
        return discoveryClient.getInstances("greetings-client")
            .stream()
            .findAny()
```

```

        .map(serviceInstance -> Collections.singletonMap(
        .orElse(null);
    }
}

```

❶

this endpoint returns a single instance of the downstream `greetings-client` endpoint whose endpoints we can access thanks to the edge services access-control headers.

The HTML5 application itself is nothing all that exciting - a jQuery application - that invokes the `greetings-client` to in turn invoke the `greetings-service` and update a DOM node with the returned value.

Example 15-14. `src/main/resources/static/index.html`

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <title>Demo</title>
    <meta name="description" content="" />
    <meta name="viewport" content="width=device-width"/>
    <base href="/" />
</head>
<body>

<div id="message"></div>

<script type="text/javascript" src="/webjars/jquery/jquery.min.js">
<script type="text/javascript">

❶
var greetingsClientUrl = location.protocol + "://" + window.lo
    + "/greetings-client-uri";
$.ajax({url: greetingsClientUrl}).done(function (data) {
    var nameToGreet = window.prompt("who would you like to gr
    var greetingsServiceUrl = data['uri'] + "/greetings-servi

❷
$.ajax({url: greetingsServiceUrl}).done(function (greetin
    $("#message").html(greeting['greeting']);

```

```
    });
  });
</script>
</body>
</html>
```

①

load from the current host information on where to find the `greetings-client`

②

make a call to the cross-origin service using the resolved URI

To see everything working, open up Eureka (`http://localhost:8761`) to find the IP under which the `html5-client` instance is now available. Grab that IP and paste it into the browser. On this machine, it was `http://10.0.1.14:8083`. It's important that you use the correct host and port to access the HTML 5 client because otherwise the `CorsFilter` won't permit the request since it checks incoming requests for hosts and ports registered in the service registry.

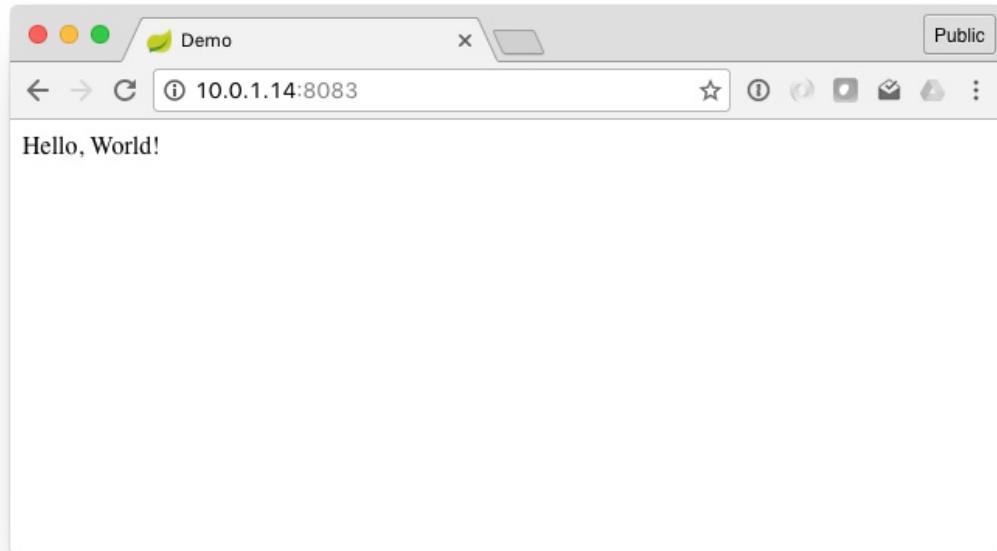


Figure 15-2. a CORS request from our HTML5 client

A Custom Zuul Filter

The `CorsFilter` is a standard `Filter`, and it works for all requests made into the edge service, including all requests made into Zuul (and the Zuul Servlet, which Spring Boot automatically wires up for you). That said, Zuul has a specialized pipeline for filtering requests that are specifically being routed through the Zuul proxy. There are four default types of Zuul filters, though of course you can add custom types if the need should arise:

The `CorsFilter` is a standard `Filter`, and it works for all requests made into the edge service, including all requests made into Zuul (and the Zuul Servlet, which Spring Boot automatically wires up for you). That said, Zuul has a specialized pipeline for filtering requests that are specifically being routed through the Zuul proxy. There are four default types of Zuul filters, though of course you can add custom types if the need should arise:

- *pre* filters are executed before the request is routed
- *routing* filters can handle the actual routing of the request
- *post* filters are executed after the request has been routed
- *error* filters execute if an error occurs in the course of handling the request

Spring Cloud registers several useful Zuul filters out of the box!

- `AuthenticationHeaderFilter` (**pre**) - a filter that looks for requests that are being proxied removes the `authorization` header before it's sent downstream
- `OAuth2TokenRelayFilter` (**pre**) - a filter that propagates an OAuth access token if it's available in the incoming request
- `ServletDetectionFilter` (**pre**) - detects whether a HTTP Servlet request has already been run through the Zuul filter pipeline

- `Servlet30WrapperFilter` (**pre**) - wraps an incoming HTTP request with a Servlet 3.0 decorator
- `FormBodyWrapperFilter` (**pre**) - wraps an incoming HTTP request with a decorator containing metadata about the multipart file upload
- `DebugFilter` (**pre**) - if an Archaius property is present in the request this will enable Zuul's `RequestContext` debug parameters which can then be used to
- `SendResponseFilter` (**post**) - commits a response to the output (if the upstream filters have contributed one)
- `SendErrorFilter` (**post**) - if the reply contains an error, this filter forwards it to a configurable endpoint
- `SendForwardFilter` (**post**) - if the reply contains a forward, this filter sends the requisite forward
- `SimpleHostRoutingFilter` (**route**) - a filter that takes the incoming request and decides, based on the URL, to which node to route the proxied request.
- `RibbonRoutingFilter` (**route**) - a filter that takes the incoming request and decides, based on the URL, to which node to route the proxied request. It does this using configurable routing and client-side load-balancing powered by Netflix Ribbon.

Let's suppose we wanted to add another generic, cross-cutting concern. A simple example might be a rate-limiter. A rate helps systems guarantee SLAs by staggering requests that exceed a certain rate to the downstream services. The [token bucket algorithm](#) is based on an analogy of a fixed capacity bucket into which tokens, normally representing a unit of bytes or a single packet of predetermined size, are added at a fixed rate. An alternative algorithm, the [leaky bucket algorithm](#), is based on an analogy of how a bucket with a leak will overflow if either the average rate at which water is poured in exceeds the rate at which the bucket leaks, or if more water than the capacity of the bucket is poured in all at once.

Security on the Edge

Edge services represent the first line of defense against possibly malicious requests from the outside. Edge services are a logical place to handle cross cutting concerns like security. Security infrastructure is tedious and as it needs to be addressed for every service it can end up being duplicative. Instead of handling on a node-by-node basis, handle it once and then propagate the authentication context to downstream services.

What we want to do is to handle authentication once and then reuse that information in downstream services. what is authentication? could we get away with using usernames and passwords? or x509 certificates? what problems are we trying to solve? if we are simply trying to answer the question: “who is making this request?” then that might very well be enough. But if you think about it, this is probably not all that you’re trying to solve.

How do you know who’s making the request? does the way in which the user signs in or authenticate change the information that we send to the downstream services? does the user always sign in the exact same way, be it on an html5 client, or an iPhone or android client? indeed, can we foresee the variety and scope of the clients that will connect? Can we trust them all equally? Do clients get different permissions? Are all clients able to keep the same guarantees about the security and integrity of transmitted data?

These questions are leading. If you do not expect there to be a variety of clients and if you do not need to assign authority and control access based on the client on which a user is accessing the service, then perhaps you can get away with a simple username and password model. In today’s world - where virtually anything could be a client - it’s safe to assume that your situation is not so simple.

OAuth

Enter OAuth. OAuth is a standard that has three incarnations: 1.0, 1.0.a, and 2.0. OAuth 2.0 is not backwards compatible with OAuth 1.x so we'll focus on OAuth 2.0. OAuth (which is short for "open authorization") is a standard for token-based authentication and authorization on the internet. The token lets clients authenticate without exposing the user's passwords. Tokens can be client-specific. OAuth provides to clients a "secure delegated access" to services on behalf of the owner of the data being accessed on the service.

Tokens reduce the window of time when usernames and passwords are exposed, making it easier to secure the occasional windows when they are exposed. OAuth is an authorization protocol ("what permissions does the user have?"), rather than an authentication protocol ("who is this user?"), so you still have to handle the authentication somewhere and then transfer control to OAuth. OAuth 2.0 is a framework and some parts of OAuth that relate to the integration with the authentication service and to the information implied by a token are non-standard.

Some terminology when thinking about OAuth:

- **client:** an application that's making protected requests on behalf of the end user (the resource owner). This could be an iPhone or Android application, an HTML 5 browser-client, even a watch!
- **resource owner:** usually this refers to the end-user ("Juergen", "Michelle", "Dave", "Margarette", etc.) that can grant a client permission to access their information.
- **resource server:** the server hosting the protected resource, capable of accepting and responding to protected requests that contain access tokens. This refers to the secured API that the client is trying to talk to.
- **authorization server:** the server issuing access tokens to the client after successful authenticating the resource owner and obtaining authorization.

OAuth supports various interactions between these actors, or roles, so that users (or clients, by themselves without a user context) may obtain tokens with which they can make authorized requests to resource owners.

The first step in OAuth is to obtain authorization - basically to get past the point of authentication. There are four well-known grant types, or flows, to achieve this.

Server Side Applications

It's very common on third party websites to see a "Sign in With Facebook" button. These kinds of applications are written in a server-side language where the source code isn't available to the public. It is possible, thus, for a server-side In this flow, a user clicks on a "Sign In" button somewhere on a third party website and is redirected to an SSL secured, well-known, trusted section of Facebook.com and - if not already logged in - prompted to login. Once logged in the user may be prompted to confer certain permissions - *Post to Wall, Read Email, etc.* - to the third party website. If the user clicks "Allow," Facebook.com redirects back to the third party website with an authorization code. The server then POSTS to the authorization server and exchanges the authorization code for an access token. The third party website may now transmit this access token in all requests it makes to the Facebook.com REST APIs and Facebook.com will happily return the information requested as though the user was making the requests directly. At no point in this flow does the third party website have the user's username and password. A misbehaving application, in this case, can only go so far and could never lock a user out of his or her account.

In this scenario, the client (on behalf of the user) makes a request to the authorization server (Facebook.com) identifying itself. The authorization server redirects back to the client with an authorization code in the request headers. The server side business logic in the client application then sends a request to exchange the authorization code for an access token. If the user isn't already signed in (authenticated), then he or she will need to sign in. This access token is stored on the server side, and the client never sees it. The only thing that's visible to the client (or to the user driving the client) is the authorization code. This indirection protects against-man-in-the-middle

attacks where somebody could otherwise intercept an access token and use that to make calls on your behalf.

Mobile applications are very similar to browser-based applications in that they can't be trusted to ensure the integrity of a client-specific secret. Other than that, the flow is very similar to the flow used by the server side web application, except that instead of redirecting to, say, `http://facebook.com`, the mobile application may redirect to a custom URL like `facebook://..` which in turn activates the native Facebook application on the device itself.

This is called the **authorization code grant**. It is useful when you want to prevent the access token from leaking beyond the client.

the HTML 5 and JavaScript Single Page Applications

Single page applications run entirely in the browser after loading the source code from a web page. The SPA can't keep any secrets, of course, because its source code is visible by clicking *View Source*. In this flow a user clicks a "Sign In" button and is prompted to approve access, just as before. If the user clicks "Allow" the service redirects the user back to the single page application site with an access token in the URL fragment (the part after the # part of a URL, e.g.: `http://some-third-party-service.com/an_oauth_callback#token=12345..`). JavaScript applications are able to read this part of the URL and changing the fragment in a URL doesn't force the browser to make a new request. This is called the **implicit grant**.

Applications without Users

In all of the examples we've looked at thus far we've always talked about a client acting on behalf of a user to access that user's data. It is possible for a client, without a particular user, to request an access token using only the client credentials. This might be useful for applications that need to access protected information but that don't have a particular user context. This is called the **client credentials grant**.

Trusted Clients

OAuth 2 supports a password grant that would be useful if you were developing the client for a service that trusted the client. Imagine you were developing the Facebook.app experience for Facebook.com; in this case the user experience of having to redirect to Facebook.com inside of Facebook.app and then *Approve* that Facebook.app has permissions to read and manipulate your Facebook.com data would seem a little redundant. Surely it does! In this instance, the trusted client - developed by engineers who also work at Facebook itself - would simply transmit the user's username and password. There's no real risk of Facebook.app suddenly obtaining your username and password as, of course, they already had it! Indeed, if you can't trust Facebook.app to not act maliciously with your Facebook.com data then you may want to reconsider Facebook as a thing! In this flow, the user transmits a username and password directly to the authorization server in exchange for an access token. This is called the **Resource Owner Password Credentials Grant**.

It is beyond the scope of this book to fully explore OAuth. We recommend the excellent (and concise!) O'Reilly book on the topic, [*Getting Started with OAuth 2.0*](#) by Ryan Boyd.

Building an OAuth Authorization Server

We'll setup a new microservice, one that's going to handle authorization duties in a module called `auth-service`. This module will use `spring-cloud-starter-config`, `spring-cloud-starter-eureka`, `spring-cloud-starter-oauth2`, `spring-boot-starter-data-jpa`, `spring-boot-starter-web`, and `spring-boot-starter-actuator`. This authorization server works with the Netflix Eureka instance introduced earlier and so participates in service registration and discovery using the `@EnableDiscoveryClient` annotation.

Spring Security

introduce the `AuthenticationManager`, `AuthenticationProvider`, `Authentication`, `UserDetailsService`, etc.

then let's setup a simple Spring Security application that

Example 15-15. `AccountConfiguration.java` - in this class we contribute a `UserDetailsService` instance that knows about our `Account` types.

```
package auth.accounts;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.authority.AuthorityUtils
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService
import org.springframework.security.core.userdetails.UsernameNotF

@Configuration
public class AccountConfiguration {

    @Bean
    UserDetailsService userDetailsService(AccountRepository accou
    ①
        return username -> accountRepository.findByUsername(username)
            .map(account -> {
```

```

        boolean active = account.isActive();
        return new User(
            account.getUsername(),
            account.getPassword(),
            active, active, active, active,
            AuthorityUtils.createAuthorityList("ROLE_ADMIN")
        )
        .orElseThrow(() -> new UsernameNotFoundException(Stri
    }
}

```

❶

the contract for a `UserDetailsService` implementation is simple: given a `String username`, return a `UserDetails` implementation or throw a `UsernameNotFoundException`. In no case, however, should you return null

Spring Security OAuth

let's introduce `ClientDetailsService` and `ClientDetails` and the `AuthServerConfigurerAdapter`.

Let's then test out the simple password flow to see if we can get an access token.

Example 15-16. `clientRepository.java`

```

package auth.clients;

import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;

public interface ClientRepository extends JpaRepository<Client, L
    Optional<Client> findByClientId(String clientId);
}

```

Example 15-17. `AuthorizationServerConfiguration.java`

```

package auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;
import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfiguration;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;
import org.springframework.security.oauth2.provider.ClientDetails;

@Configuration
@EnableAuthorizationServer
class AuthorizationServerConfiguration
    extends AuthorizationServerConfigurerAdapter {

    private final AuthenticationManager authenticationManager;
    private final ClientDetailsService clientDetailsService;

    @Autowired
    public AuthorizationServerConfiguration(AuthenticationManager authenticationManager,
                                            ClientDetailsService clientDetailsService) {
        this.authenticationManager = authenticationManager;
        this.clientDetailsService = clientDetailsService;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.withClientDetails(clientDetailsService);
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
        throws Exception {
        endpoints.authenticationManager(authenticationManager);
    }
}

<1>
```

Example 15-18. PrincipalRestController.java

```

package auth;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.security.Principal;

@RestController
class PrincipalRestController {

    @RequestMapping("/user")
    Principal principal(Principal principal) {
        return principal;
    }

}
```

<1>

Building an Implicit OAuth Client with Angular.js

let's look at a simple HTML5 browser-based application.

Building a Social OAuth Authorization Server

let's look at building an OAuth authorization server that in turn delegates to other social OAuth service providers like Facebook and Github.

Chapter 16. Routing

Locational Decoupling with Service Registration and Discovery

Cloud native systems are dynamic; services come and go as demand requires. Services must be able to discover and communicate with other services even if one instance of a service disappears or new capacity is added to the system. We can't rely on static fixed notion of IP addresses.

We *could* use DNS, but DNS isn't necessarily a great fit in a cloud environment. DNS benefits from caching. you can invalidate those caches quickly with low time-to-live value but you'll spend a lot of time re-resolving DNS. Sometimes its just not necessary to have a DNS server managed by another team with additional operational complexity. In a cloud environment DNS requires extra time for resolution, because requests must leave the cloud and then reenter through the router. Alternatively, you can build a system based on multi-homed DNS, public and private, but this adds complexity to the code.

DNS is also a pretty stupid protocol. It doesn't have the ability to answer basic questions about the state of the system, the topology of the system. Suppose you have a REST client calling a service running on a node, mapped to DNS. if the instance is down, then our client will block. Hopefully we've all applied the suggestions in Michael Nygard's amazing book, "Release It!," and specified aggressive client-side timeouts. most people don't. (Pop quiz! what's the default timeout of Java's `java.net.URLConnection`? Are you *sure* you've configured appropriate client-side timeouts in all your code? **Positive?**)

Routing becomes particularly involved when you deal with DNS load-balancing. A load-balancer is yet another piece of operational infrastructure that you need to manage (typically by asking another team) that has limitations. Your common load balancer will do fine for common things like round-robin load-balancing and most will even handle "sticky sessions," where requests from a client are pinned to a specific node based on common

headers (like Java's JSESSIONID). Load balancers fall apart when you need to do anything more involved. What if you want to pin requests from a client to a specific node based on something besides a JSESSIONID, like an OAuth Access token, or a completely stateless JSON Web Token (JWT). Perhaps you're doing something stateful, like streaming video on a specific node - which you can't really loadbalance - but don't have a notion of an HTTP session.

All nodes in a DNS load-balancing pool must also be routable. While they may be behind a firewall, their front end IPs are visible and reachable from clients. Sometimes we don't want our services to be routable via DNS! Security through obscurity is no security at all, of course, but we should always strive to keep our publicly exposed surfaces as minimal as possible. Indeed, even if we're OK with clients connecting directly to the node, we must ensure that nodes are correctly evicted from a load-balancing ensemble if they're sick. Otherwise a client will be routed to a node that can't handle the request. Not all load-balancers are smart enough to do that. Some will have the ability to query a service instance and ask it for its health and - based on the status code of the response - evict the node from the pool. (You might use Spring Boot's `/health` Actuator endpoint for this). Some languages and technologies (like those built on the JDK) tend to cache the first resolved IP from DNS, and reuse it on subsequent connections. Naturally, this is a good idea on the large, but it can defeat DNS load-balancing.

You can get around some of these limitations using a virtual load balancer which acts as a sort of proxy, but virtual load-balancers still don't solve the biggest problem. All centralized load balancers are ignorant of your system's state and its workloads. Even an ideal round-robin DNS load balancer will distribute the initial connections to different services, but not necessarily the workloads themselves. Not all requests are created equal. Some clients consume more resources than others, and may take longer to process. This can overwhelm some nodes and leave others idle.

The DiscoveryClient Abstraction

There are alternative ways to get the effects of centralized load-balancing. We want a logical mapping from service's ID to the hosts and ports (the nodes) on which that service is available. A service registry is a good fit here. The main drawback with service registries is that they're invasive in your application code. Your code must be aware of, and work with, the registry. Some service registries, like Hashicorp Consul, can act as a DNS server for clients that can't adapt. A service registry, ultimately, is like a phone book for the cloud. It's a table of service instances and it provides an API. Some service registries are more sophisticated than others, but they all minimally support discovering which services are available and where those service instances live.

Spring Cloud provides the `DiscoveryClient` abstraction to make it easy for clients to work with different types of service registries. The Spring Cloud `DiscoveryClient` abstraction makes it easy to plugin different implementations with ease. Spring Cloud plugs the `DiscoveryClient` abstraction into various parts of the stack, making its use almost transparent. As this is being written, there are production-worthy `DiscoveryClient` implementations for Cloud Foundry, Apache Zookeeper, Hashicorp Consul and Netflix's Eureka, with at least one other non-production-worthy implementation available for ETCD. The abstraction is simple enough that you could adapt Spring Cloud to work with another service registry if you'd like to. Conceptually, the `DiscoveryClient` is read-only.

Example 16-1. the `DiscoveryClient` abstraction

```
package org.springframework.cloud.client.discovery;

import java.util.List;
import org.springframework.cloud.client.ServiceInstance;

public interface DiscoveryClient {
    String description();
}
```

```

        ServiceInstance getLocalServiceInstance();

        List<ServiceInstance> getInstances(String var1);

        List<String> getServices();
    }
}

```

Some service registries need the client to register themselves with the registry. The various `DiscoveryClient` abstraction implementations do this for you on application startup. The Cloud Foundry `DiscoveryClient` abstraction implementation does not require the client to register itself with the registry because Cloud Foundry already knows on which host and port a service lives; it **put** the service there in the first place!

There are many great choices for service registries. Because Spring provides an abstraction we're not constrained in picking one. We can easily switch to another implementation later. In this chapter, we'll look at Netflix Eureka. Netflix Eureka has served Netflix well, at scale, for years. It's also easy enough to install and run entirely using Spring Boot, which makes it doubly awesome!

In order to setup a Eureka service registry, you'll need `org.springframework.cloud: spring-cloud-starter-eureka-server` in a Spring Boot project and then you'll need to initialize it using `@EnableEurekaServer`:

Example 16-2. starting up a bare-bones Eureka service registry

```

package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEure
❶
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

    public static void main(String args[]) {
        SpringApplication.run(EurekaServiceApplication.cl
}

```

```
    }  
}
```

❶

the `@EnableEurekaServer` annotation configures a bare-bones instance of the Eureka service registry.

You'll need to configure the service registry. It's a common convention to start the service on port 8761, and we don't want the registry to try to register itself with other nodes.

Example 16-3. configuring a simple Eureka service registry.

```
server.port = ${PORT:8761}  
  
❶  
eureka.client.register-with-eureka = false  
eureka.client.fetch-registry = false  
  
❷  
eureka.server.enable-self-preservation = false  
  
❸
```

we don't want Eureka to try to register with itself.

❷

if a significant portion of the registered instances stop delivering heartbeats within a time span, Eureka assumes the issue is due to the network and does not de-register the services that are not responding. This is Eureka's self-preservation mode. Leaving this set to true is probably a good idea, but it also means that if we have a small set of instances (as you will, if you're trying these examples with a few nodes on a local machine) then you won't see the expected behavior when instances de-register.

Figure 16-1. just look at this snazzy new Eureka service registry! If you mouseover the Spring leaf, you'll see it's animated. We have people for that.

At this point, our registry is available. It is **not** highly available, and in a production scenario this would be unacceptable! Indeed, Eureka will occasionally bark at us with a red error message in the console, basically saying we need to configure replica nodes, otherwise we risk creating a very fragile deployment. Remember, Eureka will be the nervous system for our REST services: other services will talk to each other through Eureka.

But, it's available. Let's stand up a client and a service and have them both register with the registry. We'll first build a trivial REST API, so that we have something with which to work. This application uses `org.springframework.cloud:spring-cloud-starter-eureka` (which provides an implementation of Spring Cloud's `DiscoveryClient` for Netflix Eureka) and `org.springframework.boot:spring-boot-starter-web`.

Example 16-4. A simple REST API that returns a greeting

```
package com.example;
```

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;

❶
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsServiceApplication {

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

@RestController
class GreetingsRestController {

    ❷
    @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
    Map<String, String> hi(@PathVariable String name) {
        return Collections.singletonMap("greeting", "Hello " + name);
    }
}

```

❶

`@EnableDiscoveryClient` activates the Spring Cloud DiscoveryClient abstraction.

❷

this will respond a simple JSON structure with a `greeting` attribute.

Our service needs only to identify itself and configure how it will interact with Eureka.

Example 16-5. `bootstrap.properties`

```
spring.application.name=greetings-service ❶  
❷  
server.port=${PORT:0}  
  
❸  
eureka.instance.instance-id=\  
 ${spring.application.name}:${spring.application.instance_id:${r
```

❶

the application will be registered as greetings-service

❷

..and it will start on a random port

❸

what distinct ID do we want for each registered service? Spring Cloud will give you a useful, node-specific, default. We've overridden the default registered ID so that they're unique.

Start up a few instances of the greetings-service and then refresh the Eureka service registry and you'll see the newly registered instances. They're now available for consumption by other services.

The screenshot shows the Spring Eureka dashboard. At the top, it displays 'HOME' and 'LAST 1000 SINCE STARTUP'. Below this is a 'System Status' section with two tables:

| Environment | test |
|-------------|---------|
| Data center | default |

| | Current time | 2016-07-10T19:43:41 +0900 |
|--------------------------|--------------|---------------------------|
| Uptime | 00:01 | |
| Lease expiration enabled | true | |
| Renews threshold | 5 | |
| Renews (last min) | 2 | |

A red warning message at the bottom of the status section reads: 'THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.'

Below the status section is a 'DS Replicas' section with a search bar containing 'localhost'. Underneath is a table titled 'Instances currently registered with Eureka' showing one instance:

| Application | AMIs | Availability Zones | Status |
|-------------------|---------|--------------------|-----------------------------------------------------------------------------------------------------------------|
| GREETINGS-SERVICE | n/a (2) | (2) | UP (2) - greetings-service:5b07cc4ab432fd5f117869c7be5d1160 , greetings-service:c9b28e027f97721c94b3cfce316254f |

Finally, there is a 'General Info' section with a table:

| Name | Value |
|--------------------|-------|
| total-avail-memory | 507mb |
| environment | test |

Figure 16-2. Eureka, now with registered instances!

Let's stand up a simple client - an edge service. Our client will interact with the `greetings-service` after resolving the service in the service registry.

Our new client also uses `spring-cloud-starter-eureka` and is annotated with `@EnableDiscoveryClient`. We can use the `DiscoveryClient` abstraction directly, if we'd like, to interrogate the registered service instances.

Example 16-6. `DiscoveryClientCLR.java`

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient
import org.springframework.stereotype.Component;
```

```

@Component
public class DiscoveryClientCLR implements CommandLineRunner {

    private final DiscoveryClient discoveryClient;

    private Log log = LoggerFactory.getLog(getClass());

    ①
    @Autowired
    public DiscoveryClientCLR(DiscoveryClient discoveryClient
        this.discoveryClient = discoveryClient;
    }

    @Override
    public void run(String... args) throws Exception {

        ②
        this.log.info("localServiceInstance");
        this.logServiceInstance(this.discoveryClient.getL

        ③
        String serviceId = "greetings-service";
        this.log.info(String.format("registered instances
        this.discoveryClient.getInstances(serviceId).forE
            this::logServiceInstance);
    }

    private void logServiceInstance(ServiceInstance si) {
        String msg = String.format("host = %s, port = %s,
            si.getHost(), si.getPort(), si.ge
        log.info(msg);
    }
}

```

①

inject the Spring Cloud configured `DiscoveryClient` implementation

②

get information on the current instance, that is, what information will be registered in Eureka for the current running application?

③

find and enumerate all the registered instances of the greetings-service

The `DiscoveryClient` makes it easy to interact with the registry and enumerate all the registered instances. If there are more than one instance of the registered service then it falls on us to select a specific version. We could randomly select from among the returned instances, which is effectively round-robin load-balancing. We might want to do something else that takes advantage of awareness of the application and the instances, like weighted-response load-balancing. Whatever the strategy, we need only describe it in code once, and then reuse it whenever we make a service call. This is the very essence of client-side load-balancing.

Netflix Ribbon is a client-side load-balancing library. Netflix Ribbon supports different strategies for load-balancing, including round-robin and weighted-response load-balancing, but its real power lies in its extensibility. Let's look at a low-level example.

Example 16-7. `DiscoveryClientCLR.java`

```
package com.example;

import com.netflix.loadbalancer.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.discovery.DiscoveryClient
import org.springframework.stereotype.Component;

import java.net.URI;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@Component
public class RibbonCLR implements CommandLineRunner {

    private final DiscoveryClient discoveryClient;

    private final Log log = LogFactory.getLog(getClass());
}
```

```

    @Autowired
    public RibbonCLR(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
    }

    @Override
    public void run(String... args) throws Exception {
        String serviceId = "greetings-service";

        ①
        List<Server> servers = this.discoveryClient.getInstances(
            .stream().map(si -> new Server(si.getHost(), si.g
            .collect(Collectors.toList()));

        ②
        IRule roundRobinRule = new RoundRobinRule();

        BaseLoadBalancer loadBalancer = LoadBalancerBuilder.newBuilder
            .withRule(roundRobinRule)
            .buildFixedServerListLoadBalancer(servers);

        IntStream.range(0, 10).forEach(i -> {
            ③
            Server server = loadBalancer.chooseServer();
            URI uri = URI.create("http://" + server.getHost() + "
                + server.getPort() + "/");
            log.info("resolved service " + uri.toString());
        });
    }
}

```

①

inject the Spring Cloud configured `DiscoveryClient` implementation

②

get information on the current instance, that is, what information will be registered in Eureka for the current running application?

③

find and enumerate all the registered instances of the `greetings-service`

This example worked, but it's tedious! Thankfully, Spring Cloud automatically integrates client-side load-balancing at various levels of the framework automatically. A particularly useful example is the auto-configuration for Spring's `RestTemplate`. The `RestTemplate` supports interceptors to pre- and post-process HTTP requests as they're made. We can use the Spring Cloud `@LoadBalanced` interceptor to signal to Spring Cloud that we want it to configure a Ribbon-aware, load-balancing interceptor on the `RestTemplate` for us.

Example 16-8. `LoadBalancedRestTemplateConfiguration.java`

```
package com.example;

import org.springframework.cloud.client.loadbalancer.LoadBalanced
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class LoadBalancedRestTemplateConfiguration {

    ①
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

①

the `@LoadBalanced` annotation is a qualifier annotation. Spring uses these to disambiguate bean definitions and to flag certain beans for processing. Here, we're flagging this particular `RestTemplate` instance as needing to have a load-balancing interceptor configured.

This done, our work is much simpler! The load-balancer will extract the URI for any HTTP request and treat the host as a service ID to be resolved using a configured `DiscoveryClient` (in this case, Netflix Eureka), and to be load-balanced using Netflix Ribbon.

Example 16-9. LoadBalancedRestTemplateCLR.java

```
package com.example;

import com.fasterxml.jackson.databind.JsonNode;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.loadbalancer.LoadBalanced
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Collections;
import java.util.Map;

@Component
public class LoadBalancedRestTemplateCLR implements CommandLineRu

    private final RestTemplate restTemplate;
    private final Log log = LogFactory.getLog(getClass());

❶
    @Autowired
    public LoadBalancedRestTemplateCLR(
        @LoadBalanced RestTemplate restTemplate)
        this.restTemplate = restTemplate;
    }

    @Override
    public void run(String... strings) throws Exception {

        Map<String, String> variables =
            Collections.singletonMap("name",
❷
            ResponseEntity<JsonNode> response = this.restTemp
                "/greetings-service/hi/{name}",
            JsonNode body = response.getBody();
            String greeting = body.get("greeting").asText();
            log.info("greeting: " + greeting);
        }
    }

❸
```

we use the same `@LoadBalanced` qualifier annotation at both the bean producer and injection site.

②

we use the `RestTemplate` as we might otherwise, specifying for the host a service ID (`greetings-service`) instead of a DNS address.

Cloud Foundry Route Services

Client-side load-balancing gives us a lot of power over routing decisions and we have an easier time, as developers, dictating routing logic in code. The routing is local to our system of services, and if we need to change it, we can without worrying about impacting somebody else consuming the service. It's not realistic to expect that you can demand all third party clients consume your API in the same way, though. This becomes especially true when standing up edge services (which we talk about in another chapter) that respond to client-side requests, requests from any number of different devices and experiences. iOS clients, for example, aren't going to want to use Apache Zookeeper and Netflix Ribbon to do their work. Here, it can be useful to standup an intermediary service that handles the request, handles routing, and then forwards the requests to the downstream services. Such a component also has the benefit of being applicable to services of all types, no matter the implementation language.

Cloud Foundry supports a special kind of component - a *route service* - that gives us most of the benefits of client-side loadbalancing and having a centralized component that enforces routing behavior. A route service is another extension plane in Cloud Foundry, along with buildpacks, applications, and service brokers, that you can develop and plugin to your platform. A route service is ultimately an HTTP service that accepts all HTTP requests for a given host and domain and then does whatever it would like with them.

Route services are meant to be generic proxies that it can interpose in the request chain for - theoretically - any kind of application, so some constraints apply. They must support both HTTP *and* HTTPS. Cloud Foundry, presently, doesn't support chaining route services for one request to another. Cloud Foundry treats route services as another type of user-provided service. They're a little different than the other types of user-provided services we've discussed in this book in that the connection between a route service and an application is specified in terms of the hostname and domain of the applications, not the application name.

Let's look at an example. Since we need to accept and just pass through all incoming HTTP requests, we need to configure Spring's `RestTemplate` to trust everything and to ignore errors.

Example 16-10. `RouteServiceApplication.java`

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
import org.springframework.web.client.DefaultResponseErrorHandler;
import org.springframework.web.client.RestTemplate;

import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.Proxy;
import java.net.URL;
import java.security.KeyManagementException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

/**
 * @author Ben Hale
 */
@SpringBootApplication
public class RouteServiceApplication {

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

    ①
    @Bean
    RestTemplate restOperations() {
        RestTemplate restTemplate = new RestTemplate(
            new TrustEverythingClientHttpRequestFactory()); ②
    }
}
```

```

        restTemplate.setErrorHandler(new NoErrorsResponseErrorHandler());
        return restTemplate;
    }

    private static class NoErrorsResponseErrorHandler
        extends DefaultResponseErrorHandler {

        @Override
        public boolean hasError(ClientHttpResponse response) throws
            return false;
    }
}

private static final class TrustEverythingClientHttpRequestFactory
    extends SimpleClientHttpRequestFactory {

    @Override
    protected HttpURLConnection openConnection(URL url, Proxy proxy)
        HttpURLConnection connection = super.openConnection(url);
        if (connection instanceofHttpsURLConnection) {
            HttpsURLConnection httpsConnection = (HttpsURLConnection) connection;
            SSLContext sslContext = getSSLContext(new TrustEverythingTrustManager());
            httpsConnection.setSSLSocketFactory(sslContext.getSocketFactory());
            httpsConnection.setHostnameVerifier((s, session) -> true);
        }
        return connection;
    }

    private static SSLContext getSSLContext(TrustManager trustManager)
        try {
            SSLContext sslContext = SSLContext.getInstance("TLS");
            sslContext.init(null, new TrustManager[]{trustManager}, null);
            return sslContext;
        } catch (KeyManagementException | NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}

private static final class TrustEverythingTrustManager
    implements X509TrustManager {

    @Override
    public void checkClientTrusted(X509Certificate[] x509Certificates, String
        }
    }

    @Override

```

```

        public void checkServerTrusted(X509Certificate[] x509Cert
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
}

```

❶

configure a RestTemplate..

❷

that trusts everything..

❸

and ignores errors

This route service logs all incoming requests before and after accepting them.

Example 16-11. Controller.java

```

package com.example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestOperations;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

/**
 * @author Ben Hale
 */
@RestController

```

```

class Controller {

    private final Logger logger = LoggerFactory.getLogger(this.ge
    ①
    private static final String FORWARDED_URL = "X-CF-Forwarded-U
    private static final String PROXY_METADATA = "X-CF-Proxy-Meta
    private static final String PROXY_SIGNATURE = "X-CF-Proxy-Sig

    private final RestOperations restOperations;

    ②
    @RequestMapping(headers = {FORWARDED_URL, PROXY_METADATA, PRO
    ResponseEntity<?> service(RequestEntity<byte[]> incoming) {

        this.logger.info("incoming request: {}", incoming);

        HttpHeaders headers = new HttpHeaders();
        headers.putAll(incoming.getHeaders());

    ③
    URI uri = headers
        .remove(FORWARDED_URL)
        .stream()
        .findFirst()
        .map(URI::create)
        .orElseThrow(() -> new IllegalStateException(
            String.format("No %s header present", FOR
    }

    ④
    RequestEntity<?> outgoing = new RequestEntity<>(
        ((RequestEntity<?>) incoming).getBody(),
        headers,
        incoming.getMethod(),
        uri);

    this.logger.info("outgoing request: {}", outgoing);

    return this.restOperations.exchange(outgoing, byte[].clas
    }

    @Autowired
    Controller(RestTemplate restOperations) {
        this.restOperations = restOperations;
    }
}

```

❶

Cloud Foundry passes along custom headers that provide context for an incoming request.

❷

we use the presence of these headers as a selector to determine if the route service should handle the request.

❸

in the request processing method we determine the ultimate request by extracting the appropriate header

❹

and adapt the incoming request into an outgoing request to be executed using the `RestTemplate`

This is a trivial example, but the possibilities are endless. We could pre-process requests and handle authentication, or perhaps adapt one type of authentication for another. We could insert logic to handle rate limiting, or we could forward requests in a certain way by relying on client-side loadbalancing. We could handle metering for requests going into the system. We could simply use this as a sort of generic service-level interstitial and enrich the request body with global state or request context (the user's clickstream information, perhaps?)

A lot of the things you may want to do here - metering, rate limiting, authentication, etc. - are generic concerns that you don't need necessarily to write code for. A lot of this functionality is well addressed by API-gateway products like Apigee, who also distribute a Cloud Foundry route service that you can configure for your application. The Apigee API gateway is connected to your Apigee account and all the policy you centrally configure there is automatically applied to your requests.

Using a route-service is fairly straightforward, but not exactly the same as using any of the other types of route-services. A route-service is ultimately just a Cloud Foundry application that you've deployed. Suppose you've

already deployed two applications to Cloud Foundry: `route-service` (the logging route service we just looked at) and a regular Spring Boot and Spring MVC application `downstream-service`.

Example 16-12. connecting an application (`downstream-service`) to a route service (`route-service`) for applications deployed on Pivotal Web Services where the domain is `cfapps.io` by default. If you're using a deployed Cloud Foundry instance elsewhere then this won't be the case.

```
cf create-user-provided-service route-service -r https://my-route  
②  
cf bind-route-service cfapps.io route-service -hostname my-downst  
①
```

create a user provided route service, pointing it to the URL for our deployed `route-service`

②

bind that route service to any application whose route is `http://my-downstream-service.cfapps.io`

Now, visit the `downstream-service` in the browser a few times and then tail the logs for the route service (`cf logs --recent route-service`), to see the log output reflecting the requests.

Next Steps

For services deployed over HTTP (nothing says that they have to be, though!), routing is a critical concern which becomes even more important when the lifetime of a service is dynamic and when service topologies changes. We want, for the purposes of homogeneous clients accessing the services from outside our system, to project that services are available in a fixed, well-known places but retain complete flexibility for intra-service communication. We can achieve these seemingly conflicting concerns with a little thoughtfulness and by leveraging some of the practices described in this chapter.

Index