O'REILLY®

Compliments of Pivotal®

PREVIEW EDITION

Cloud Native Java

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

Josh Long & Kenny Bastani

Pivotal **Cloud Foundry**®

Cloud-Native Java At Your Service

Install, Deploy, Secure & Manage Spring Cloud's Service Discovery, Circuit Breaker Dashboard, and Config Server capabilities automatically as services managed on Pivotal Cloud Foundry®.

Engineered for apps built with Spring Boot

A distributed platform engineered for distributed Spring Cloud Apps

Cloud-Native stream and batch processing with Spring Cloud Data Flow

Learn more at **pivotal.io/platform**

Pivotal®

Cloud Native Java

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

This Preview Edition of *Cloud Native Java*, Chapters 1, 2, 7, and 10, is a work in progress. The final book is expected in April 2017, and will be available on oreilly.com and through other retailers when it's published.

Josh Long & Kenny Bastani

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Cloud Native Java
by Josh Long, Kenny Bastani

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

November 2015: 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

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

978-1-4493-7464-8

[LSI]

Table of Contents

CHAPTER 1

Bootcamp: Introducing Spring Boot and Cloud Foundry

From the project website, "Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration."

This pretty much says it all! Spring builds upon the Spring ecosystem and interesting third-party libraries, adding in opinions and establishing conventions to streamline the realization of production-worthy applications. This last aspect is arguably the most important aspect of Spring Boot. Any framework these days can be used to standup a simple REST endpoint, but a simple REST endpoint does not a production-worthy service make. When we say that an application is **cloud native**, it means that it is designed to thrive in a cloud-based production environment.

This chapter will introduce you to building Spring Boot applications and deploying them to production with Cloud Foundry. The rest of the book will talk about what it means to build applications that **thrive** in such an environment.

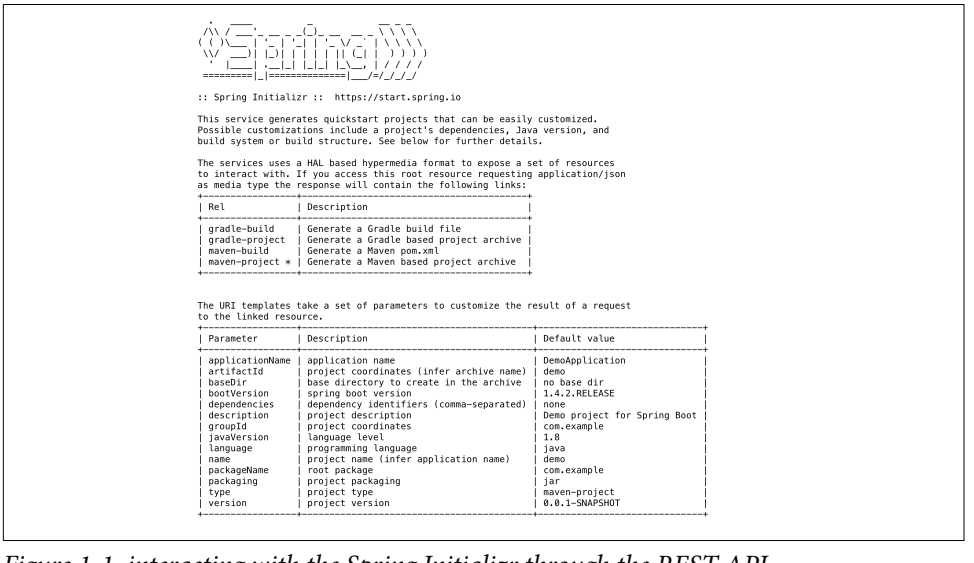
Getting Started with the Spring Initializr

The *Spring Initializr* is an open source project and tool in the Spring ecosystem that helps you quickly generate new Spring Boot applications. Pivotal runs an instance of the Spring Initializr hosted on Pivotal Web Services at *http://start.spring.io*. It generates Maven and Gradle projects with any specified dependencies, a skeletal entry point Java class and a skeletal unit test.

In the world of monolithic applications, this cost may be prohibitive, but it's easy to amortize the cost of that initialization across the lifetime of the project. When you move to the cloud native architecture, you'll end up with lots of small microservices. The first requirement, then, is to reduce the cost of creating a new service. The Spring Initializr helps reduce that upfront cost. The Spring Initializr application is both a web application that you can consume from your web browser, and as a REST API, that will generate new projects for you. You could generate a default project using `curl`:

```
curl http://start.spring.io
```

The results will look something like this:



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

:: Spring Initializr ::  https://start.spring.io

This service generates quickstart projects that can be easily customized.
Possible customizations include a project's dependencies, Java version, and
build system or build structure. See below for further details.

The services uses a HAL based hypermedia format to expose a set of resources
to interact with. If you access this root resource requesting application/json
as media type the response will contain the following links:
+-----------------+-----------------------------------------+
| Rel             | Description                             |
+-----------------+-----------------------------------------+
| gradle-build    | Generate a Gradle build file            |
| gradle-project  | Generate a Gradle based project archive |
| maven-build     | Generate a Maven pom.xml                |
| maven-project *  | Generate a Maven based project archive  |
+-----------------+-----------------------------------------+


The URI templates take a set of parameters to customize the result of a request
to the linked resource.
+-----------------+-----------------------------------------+-----------------------------+
| Parameter       | Description                             | Default value               |
+-----------------+-----------------------------------------+-----------------------------+
| applicationName | application name                        | DemoApplication             |
| artifactId      | project coordinates (infer archive name) | demo                        |
| baseDir         | base directory to create in the archive  | no base dir                 |
| bootVersion     | spring boot version                     | 1.4.2.RELEASE               |
| dependencies    | dependency identifiers (comma-separated) | none                        |
| description     | project description                     | Demo project for Spring Boot |
| groupId         | project coordinates                     | com.example                 |
| javaVersion     | language level                          | 1.8                         |
| language        | programming language                    | java                        |
| name            | project name (infer application name)   | demo                        |
| packageName     | root package                            | com.example                 |
| packaging       | project packaging                       | jar                         |
| type            | project type                            | maven-project               |
| version         | project version                         | 0.0.1-SNAPSHOT              |
+-----------------+-----------------------------------------+-----------------------------+
```

Figure 1-1. interacting with the Spring Initializr through the REST API

Alternatively, you can use the Spring Initializr from the browser as in shown here.

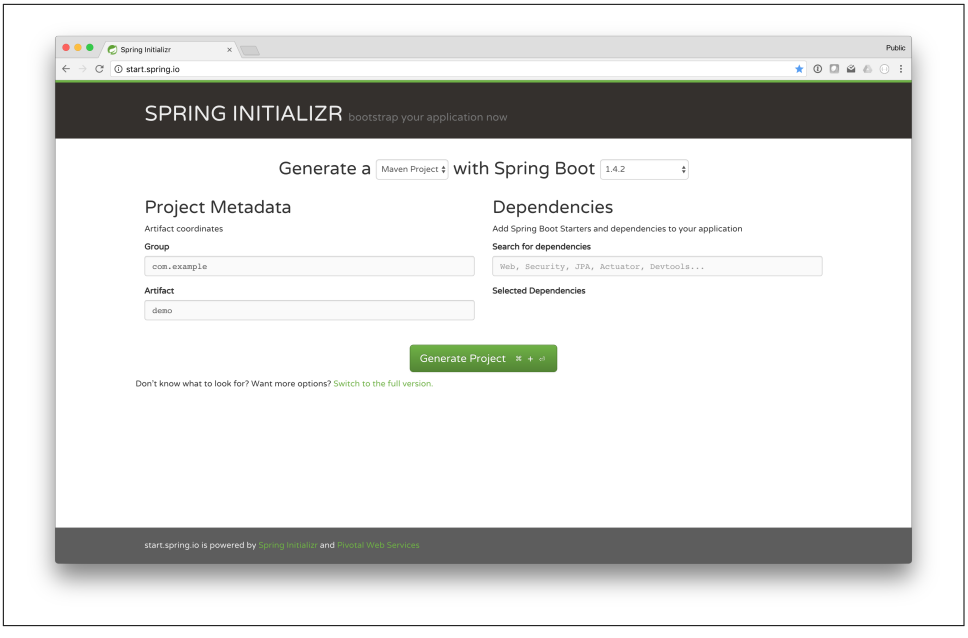


Figure 1-2. The Spring Initializr website

Let's suppose we want to build a simple REST service that talks to a SQL database (H2). We'll need different libraries from the Spring eocsystem, including Spring MVC, Spring Data JPA and Spring Data REST. We'll look at what these libraries do, later.

Search for *Dependencies* in the search box or click *Switch to the full version* and manually selecting checkboxes for the desired dependencies. Most of these are *starter* dependencies.

In Spring Boot, a *starter* dependency is an opinionated dependency. It is a Maven `pom.xml` that is published as a `.jar` that brings in other Maven dependencies, but has no Spring Boot code in of itself. The interesting Java code from Spring Boot lives in only a couple of `.jar` libraries. The Spring Boot *starter* depdencies transitively import these core libraries.

If we were to build a Spring Boot application from scratch, we would have to weather the web of dependency version conflicts. We might encounter a version conflict if our application used dependency `A` and dependency `B` and they shared a common, but conflicting, dependency on `C`. Spring Boot solves this for you in Maven and Gradle builds, allowing you to focus on what you want on your CLASSPATH, not how to resolve conflicts from having it.

Table 1-1. Some example Spring Boot Starters for a typical Spring Boot application

Spring Project	Starter Projects	Maven `artifactId`
Spring Data JPA	JPA	`spring-boot-starter-data-jpa`
Spring Data REST	REST Repositories	`spring-boot-starter-data-rest`
Spring Framework (MVC)	Web	`spring-boot-starter-web`
Spring Security	Security	`spring-boot-starter-security`
H2 Embedded SQL DB	H2	`h2`

Make your selections and include just `Web`, `H2`, `REST Repositories`, and `JPA`. We'll leave everything else as default. Click **Generate Project** and an archive, `demo.zip`, will begin downloading. Decompress the archive and you'll have a skeletal project ready to import into an IDE of your choice.

Example 1-1. the contents of the generated Spring Boot application archive once decompressed

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── DemoServiceApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── example
                    └── DemoServiceApplicationTests.java
```

In the listing of our application's directory structure we can see the directory structure of our generated application. The Spring Initializr provides a wrapper script - either the Gradle wrapper (`gradlew`) or a Maven wrapper (from the Maven wrapper project) `mvnw` - as a part of the contents of the generated project. You can use this wrapper to build and run the project. The wrapper downloads a configured version of the build tool on its first run. The version of Maven or Gradle are version controlled. This means that all subsequent users will have a reproducible build. There's no risk that someone will try to build your code with an incompatible version of Maven or

Gradle. This also greatly simplifies continuous delivery: the build used for development is the exact same one used in the continuous integration environment.

The following command will run a clean installation of the Maven project, downloading and caching the dependencies specified in the `pom.xml` and installing the built `.jar` artifact into the local Maven repository (typically `$HOME/.m2/repository/*`).

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

To run the Spring Boot application from the command line, use the provided Spring Boot Maven plugin, configured automatically in the generated `pom.xml`:

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

The web application should be up and running and available at *http://localhost:8080*. Don't worry, nothing particularly interesting has happened, *yet*.

Open the project's `pom.xml` file in a text editor of your choice. `emacs`, `vi`, TextMate, Sublime, Atom, Notepad.exe, etc., are all valid choices.

Example 1-2. The `demo-service` project's `pom.xml` file's dependencies section.

```
<dependencies>

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

        ❷
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>

        ❸
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        ❹
        <dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <scope>runtime</scope>
        </dependency>

        ❺
```

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

❶ `spring-boot-starter-data-jpa` brings in everything needed to persist Java objects using the Java ORM (Object Relational Mapping) specification, JPA (the Java Persistence API), and be productive out of the gate. This includes the JPA specification types, basic SQL Java database connectivity (JDBC) and JPA support for Spring, Hibernate as an implementation, Spring Data JPA and Spring Data REST.

❷ `spring-boot-starter-data-rest` makes it trivial to export hypermedia-aware REST services from a Spring Data repository definition.

❸ `spring-boot-starter-web` brings in everything needed to build REST applications with Spring. It brings in JSON and XML marshalling support, file upload support, an embedded web container (the default is the latest version of Apache Tomcat), validation support, the Servlet API, and so much more. This is a redundant dependency as `spring-boot-starter-data-rest` will automatically bring it in for us. It's highlighted here for clarity.

❹ `h2` is an in-memory, embedded SQL database. If Spring Boot detects an embedded database like H2, Derby or HSQL on the classpath and it detects that you haven't otherwise configured a `javax.sql.DataSource` somewhere, it'll configure one for you. The embedded `DataSource` will spin up when the application spins up and it'll destroy itself (and all of its contents!) when the application shuts down.

❺ `spring-boot-starter-test` brings in all the default types needed to write effective mock and integration tests, including the Spring MVC test framework. It's assumed that testing support is needed, and this dependency is added by default.

The parent build specifies all the versions for these dependencies. In a typical Spring Boot project the only version explicitly specified is the one for Spring Boot itself. When one day a new version of Spring Boot is available, point your build to the new version and all the corresponding libraries and integrations get updated with it.

Next, open up the application's entry point class, `demo/src/main/java/com/example/DemoApplication.java` in your favorite text editor and replace it with the following code:

Example 1-3.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@SpringBootApplication ❶
public class DemoApplication {

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

❸
@Entity
class Cat {

        @Id
        @GeneratedValue
        private Long id;
        private String name;

        @Override
        public String toString() {
                return "Cat{" +
                                "id=" + id +
                                ", name='" + name + '\'' +
                                '}';
        }

        Cat() {
        }

        public Cat(String name) {
                this.name = name;
        }

        public Long getId() {
                return id;
        }

        public String getName() {
                return name;
        }
```

```
}
```

❹

```
@RepositoryRestResource
interface CatRepository extends JpaRepository<Cat, Long> {
}
```

❶ annotates a class as a Spring Boot application.

❷ starts the Spring Boot application

❸ a plain JPA entity to model a `Cat` entity

❹ a Spring Data JPA repository (which handles all common create-read-update-and-delete operations) that has been exported as a REST API

This code *should* work, but we can't be sure unless we have a test! If we have tests, we can establish a baseline working state for our software and then make measured improvements to that baseline quality. So, open up the test class, `demo/src/test/java/com/example/DemoApplicationTests.java`, and replace it with the following code:

Example 1-4.

```
package com.example;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
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 java.util.stream.Stream;

import static org.junit.Assert.assertTrue;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuild
ers.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.con
tent;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.sta
tus;
```

❶

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
```

```
@AutoConfigureMockMvc
public class DemoApplicationTests {

        ❷
        @Autowired
        private MockMvc mvc;

        ❸
        @Autowired
        private CatRepository catRepository;

        ❹
        @Before
        public void before() throws Exception {
                Stream.of("Felix", "Garfield", "Whiskers").forEach(n -> catReposi
tory.save(new Cat(n)));
        }

        ❺
        @Test
        public void catsReflectedInRead() throws Exception {
                MediaType halJson = MediaType.parseMediaType("application/hal
+json;charset=UTF-8");
                this.mvc.perform(get("/cats"))
                                .andExpect(status().isOk())
                                .andExpect(content().contentType(halJson))
                                .andExpect(mvcResult -> {
                                        String contentAsString = mvcResult.getRes
ponse()
                                                        .getContentAsString();
                                        assertTrue(contentAsString.split("totalEle
ments")[1].split(":")[1]
                                                        .trim().split(",")
[0].equals("3"));
                                });
        }
}
```

❶ this is a unit test that leverages the Spring framework test runner. We configure it to also interact nicely with the Spring Boot testing apparata, standing up a mock web application.

❷ inject a Spring MVC test `MockMvc` client with which we make calls to the REST endpoints

❸ we can reference any other beans in our Spring application context, including `CatRepository`

❹ install some sample data in the database

❺ invoke the HTTP `GET` endpoint for the `/cats` resource.

We'll learn more about testing in our discussion on testing. Start the application and manipulate the database through the HAL-encoded hypermedia REST API on `http://locahost:8080/cats`. We'll learn more about data manipulation in Spring in ??? and we'll learn more about processing data in Chapter 3. We'll learn more about REST and web applications in the ??? chapter. If you run the test, it should be green! We'll learn more about testing in the chapter on ???.

Thus far we've done everything using a text editor, but most developers today use an IDE. There are many fine choices out there and Spring Boot works well with any of them. If you don't already have a Java IDE, you might consider the Spring Tool Suite (STS). The Spring Tool Suite is an Eclipse-based IDE that packages up common ecosystem tooling to deliver a smoother Eclipse experience than what you'd get out of the box if you were to download the base Eclipse platform from Eclipse. STS is freely-available under the terms of the Eclipse Public License.

STS provides an in-IDE experience for the Spring Initializr. Indeed, the functionality in Spring Tool Suite, IntelliJ Ultimate edition, NetBeans and the Spring Initializr web application itself all delegate to the Spring Initializr's REST API, so you get a common result no matter where you start.

Let's build a new project in the Spring Tool Suite.

Getting Started with the Spring Tool Suite

You do *not need* to use any particular IDE to develop Spring or Spring Boot applications. The authors have built Spring applications in emacs, plain Eclipse, Apache Netbeans (with and without the excellent Spring Boot plugin support that engineers from no less than Oracle have contributed) and IntelliJ IDEA Community edition and IntelliJ IDEA Ultimate edition with no problems. Spring Boot 1.x, in particular, needs Java 6 or better and support for editing plain `.properties` files as well as support for working with Maven or Gradle-based builds. Any IDE from 2010 or later will do a good job here.

If you use Eclipse, the Spring Tool Suite has a lot of nice features that make working with Spring Boot-based projects even nicer:

- you can access all of the Spring guides in the STS IDE
- you can generate new projects with the Spring Initializr in the IDE.
- If you attempt to access a type that doesn't exist on the CLASSPATH, but can be discovered within one of Spring Boot's `-starter` dependencies, then STS will automatically add that type for you.

- The **Boot Dashboard** makes it seamless to edit local Spring Boot applications and have them synchronized with a Cloud Foundry deployment. You can further debug and live-reload deployed Cloud Foundry applications, all within the IDE.
- STS makes editing Spring Boot `.properties` or `.yml` files effortless, offering autocompleion out of the box.
- The Spring Tool Suite is a stable, integrated edition of Eclipse that's released shortly after the mainline Eclipse release.

Installing Spring Tool Suite (STS)

Download and install the Spring Tool Suite (STS), 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. Select 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.

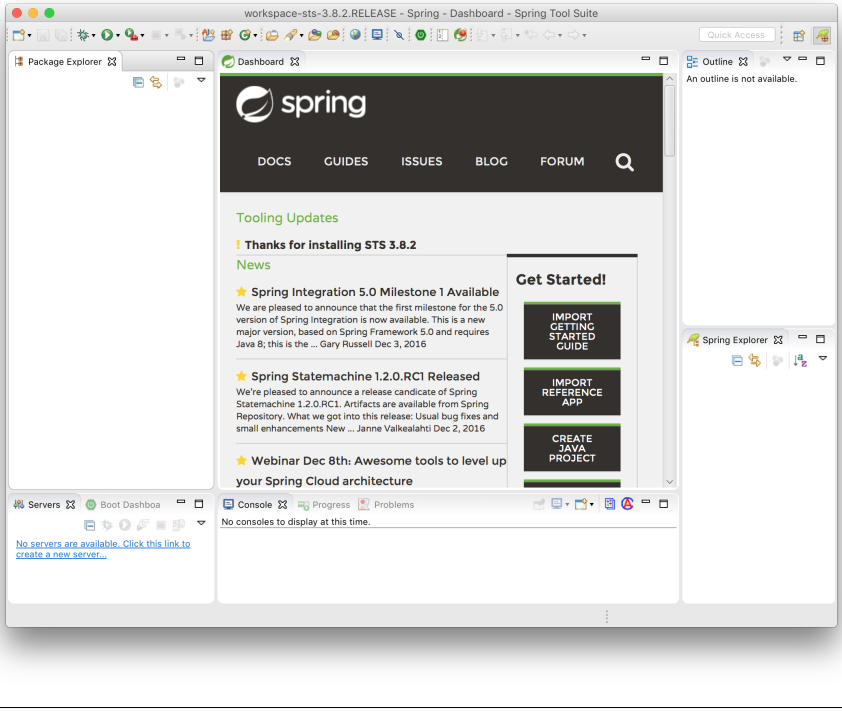


Figure 1-3. The STS dashboard

There are also packages available for numerous operating systems, as well. For example, if you use Homebrew Cask on OS X or macOS Sierra, you can use the Pivotal tap and then say `brew cask install sts`.

Creating a new Project with the Spring Initializr

We could import the example that we just created from the Spring Initializr, directly, by going to `File` > `Import` > `Maven` and then pointing the import to the `pom.xml` at the root of our existing `demo` project, but let's instead 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 applications. To create a new Spring Boot application using a Spring Boot Starter project, choose from the menu `File` > `New` > `Spring Starter Project`.

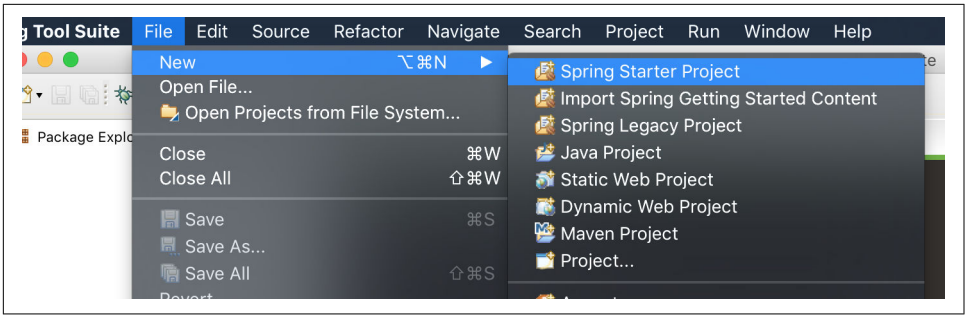


Figure 1-4. Create a new Spring Boot Starter Project

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

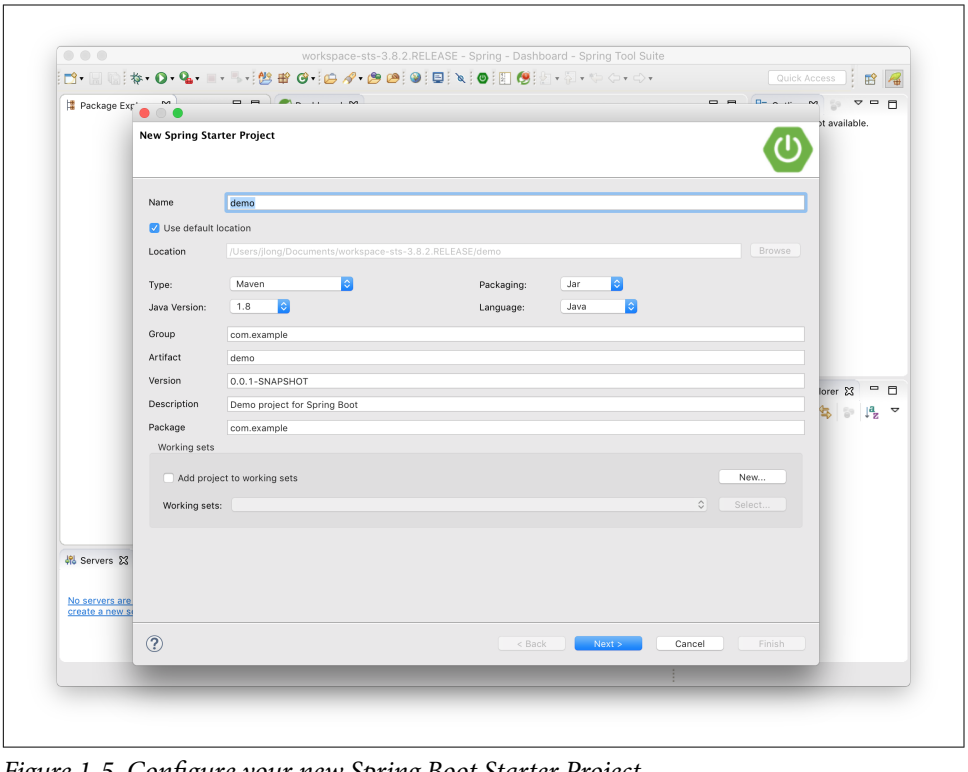


Figure 1-5. Configure your new Spring Boot Starter Project

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 select the **Web** support.

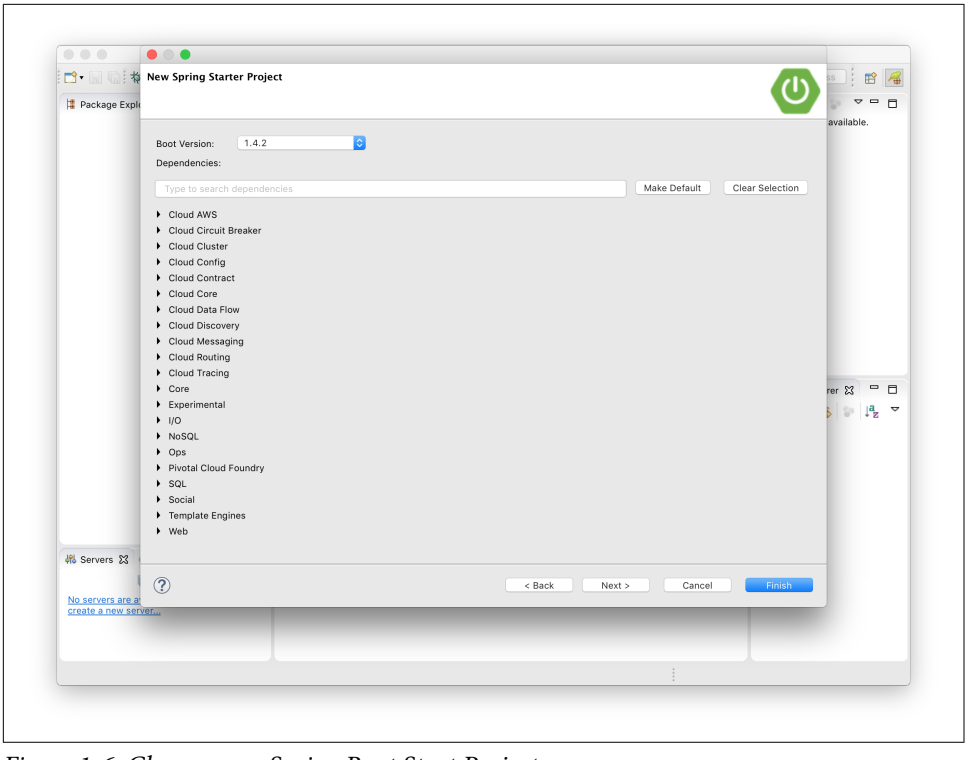


Figure 1-6. Choose your Spring Boot Start Project

Once you've made your selections, click **Finish**. After you click **Finish**, your Spring Boot application will be created and imported into the IDE workspace for you and visible in the package explorer.

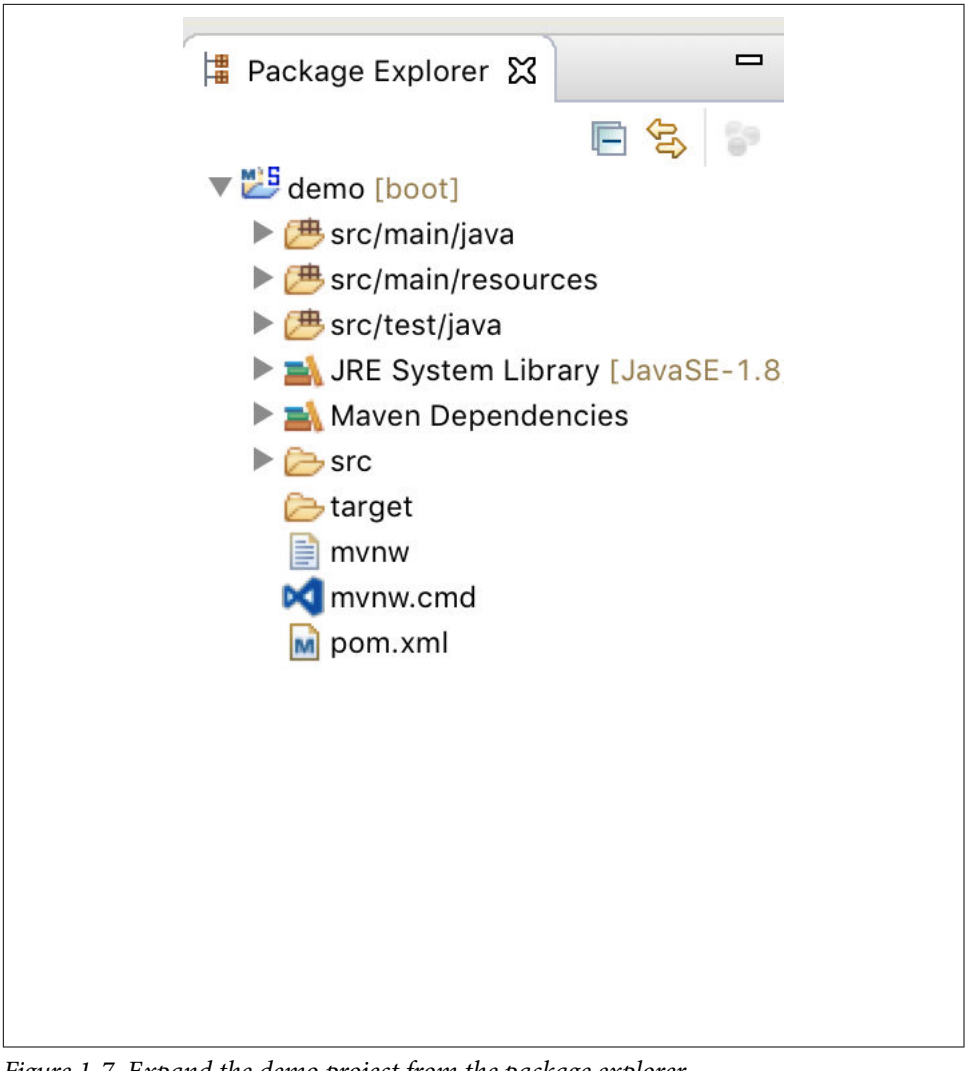


Figure 1-7. Expand the demo project from the package explorer

If you haven't already, expand the ***demo [boot]*** node in the Package Explorer 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`. Let's go ahead and run the application. Run the application from the **Run** > **Run** menu.

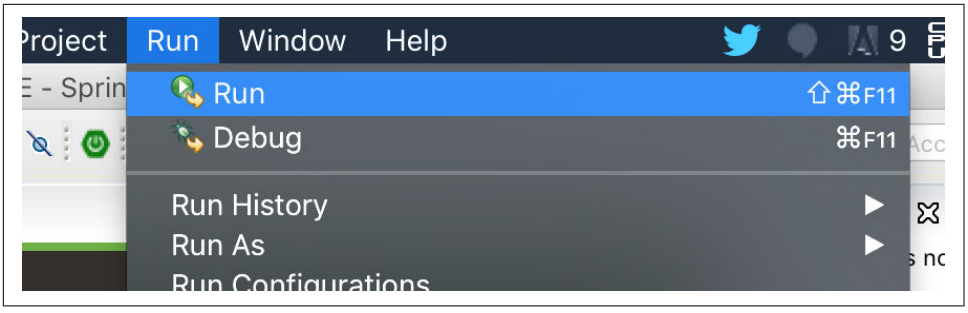


Figure 1-8. Run the Spring Boot application

After choosing the **Run** option from the menu, you'll be presented with a *Run As* dialog. Choose **Spring Boot App** and then click **OK**.

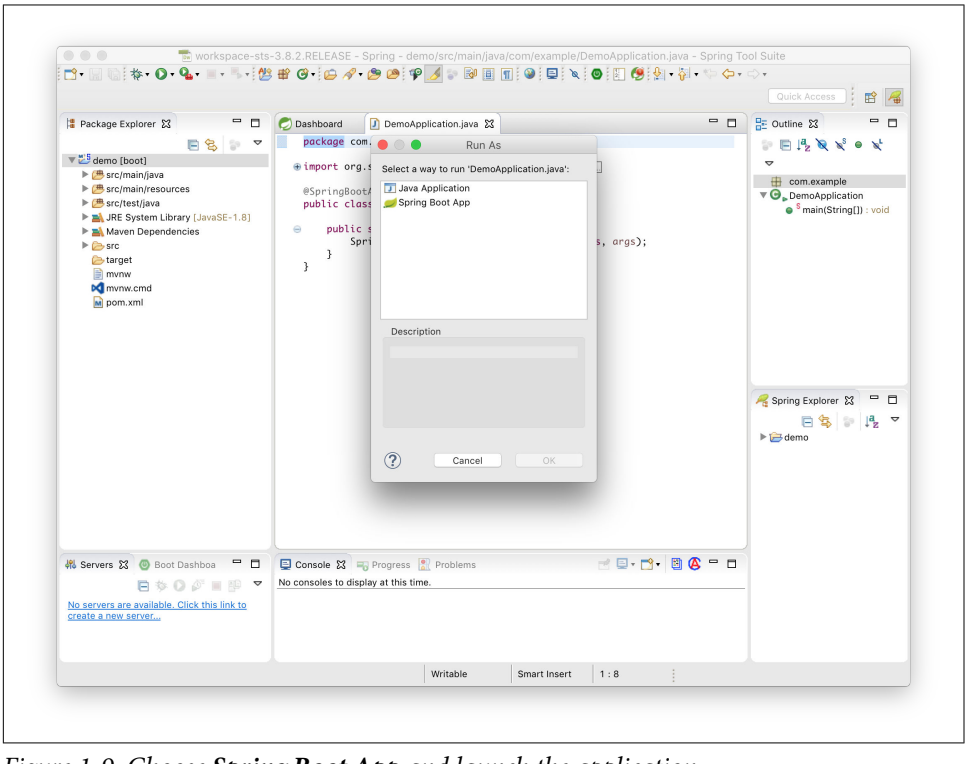


*Figure 1-9. Choose **Spring Boot App** and launch the application*

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***.

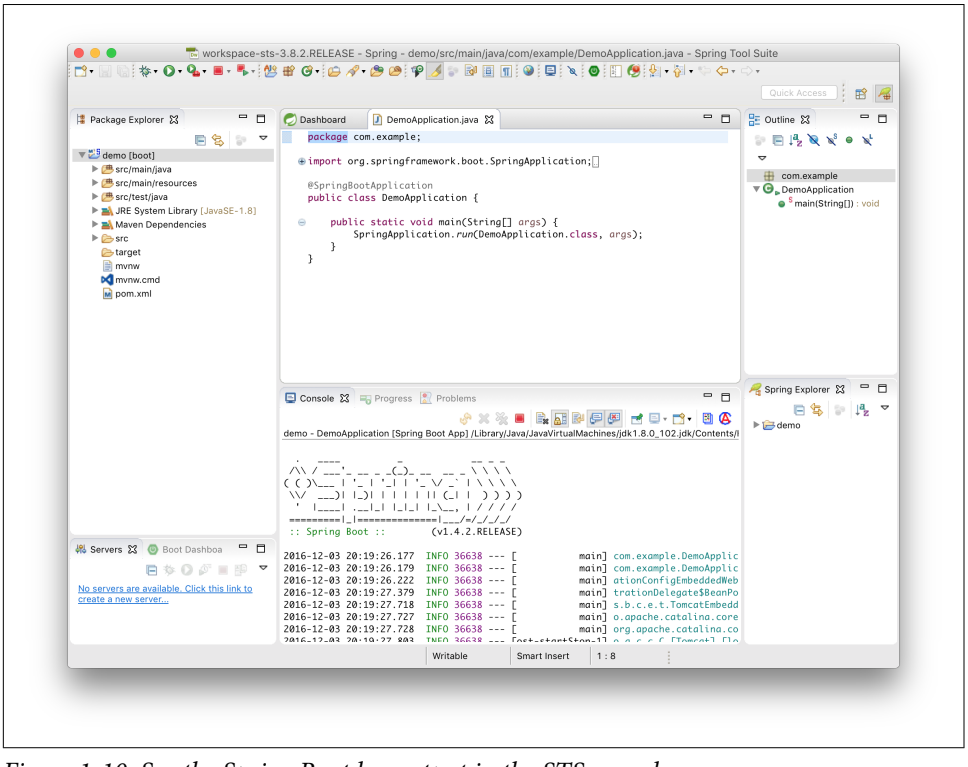


Figure 1-10. See the Spring Boot log output in the STS console

Congratulations! You've just created your first Spring Boot application with the Spring Tool Suite!

The Spring Guides

The Spring Guides are a set of small, focused introductions to all manner of different topics in terms of Spring projects. There are many guides, most of which were written by experts on the Spring team, though some of them were contributed from ecosystem partners. 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 (in addition to this book, of course!) when getting started with Spring Boot. To get started with the *Spring Guides*, head over to *https://spring.io/guides*.

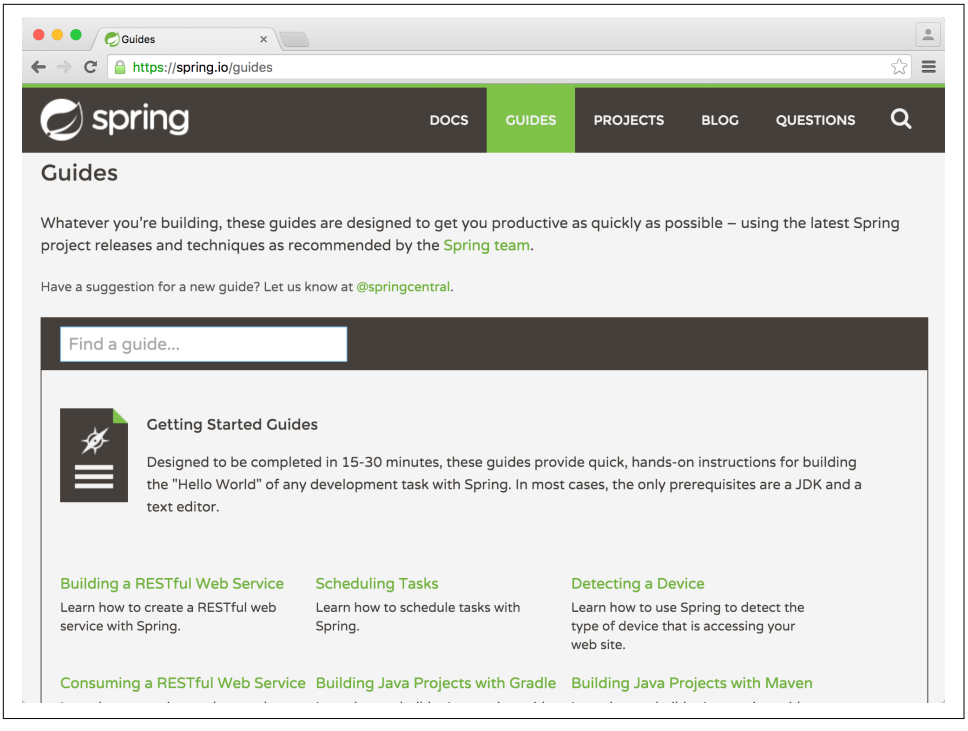


Figure 1-11. The Spring Guides website

As we can see from Figure 1-11, the *Spring Guides* website provides a collection of maintained examples that target a specific use case.

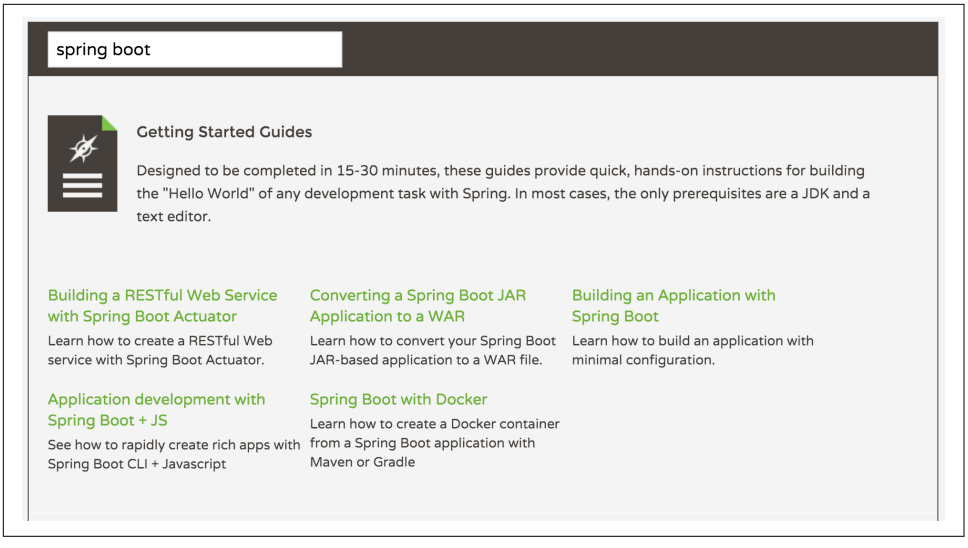


Figure 1-12. Exploring the Spring Guides

In Figure 1-12 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.

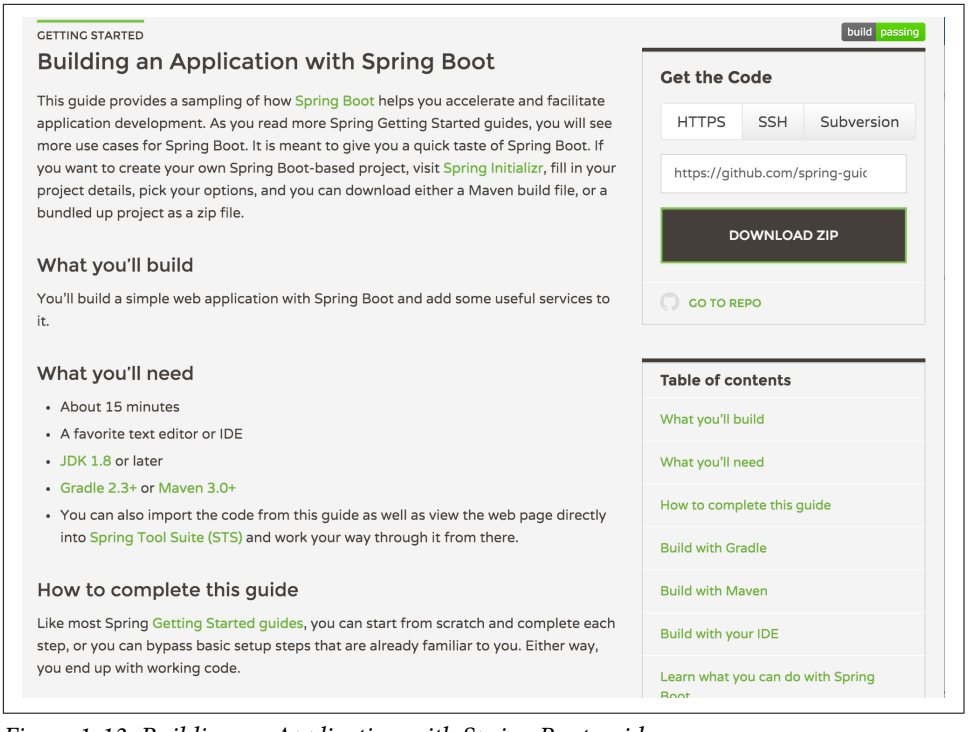


Figure 1-13. Building an Application with Spring Boot guide

In Figure 1-13 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. Each guide features a GitHub repository that contains three folders: `complete`, `initial`, and `test`. The `complete` folder contains the final working example so that you can check your work. The `initial` folder contains a skeletal, almost empty file system that you can use to push past the boilerplate and focus on that which is unique to the guide. The `test` folder has whatever's required to confirm that the `complete` project works.

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.

Following the Guides in STS

If you're using the Spring Tool Suite, you can follow along with the guides from within the IDE. Choose **File** > **New** > **Import Spring Getting Started Content**.

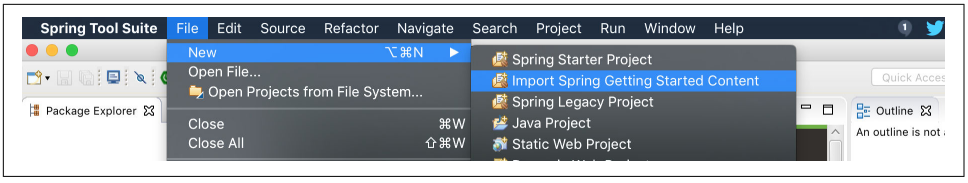


Figure 1-14. Import Spring Getting Started Content

You'll be given the same catalog of guides as you would at spring.io/guides.

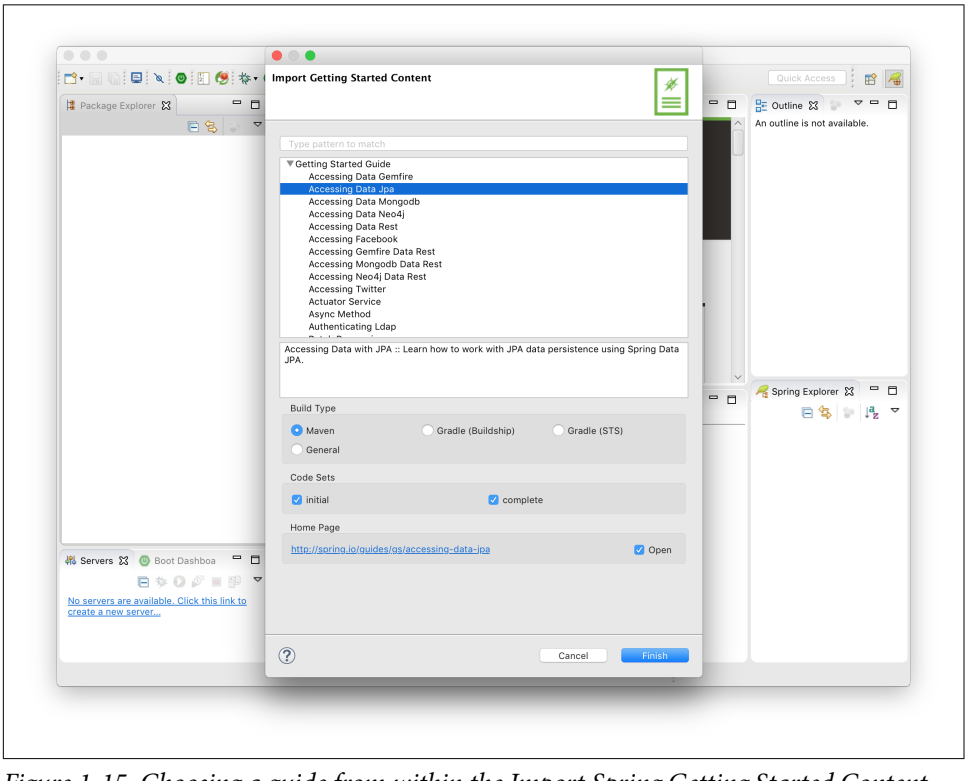


Figure 1-15. Choosing a guide from within the Import Spring Getting Started Content dialog

Select a guide and it'll be displayed within your IDE and the associated code will be shown.

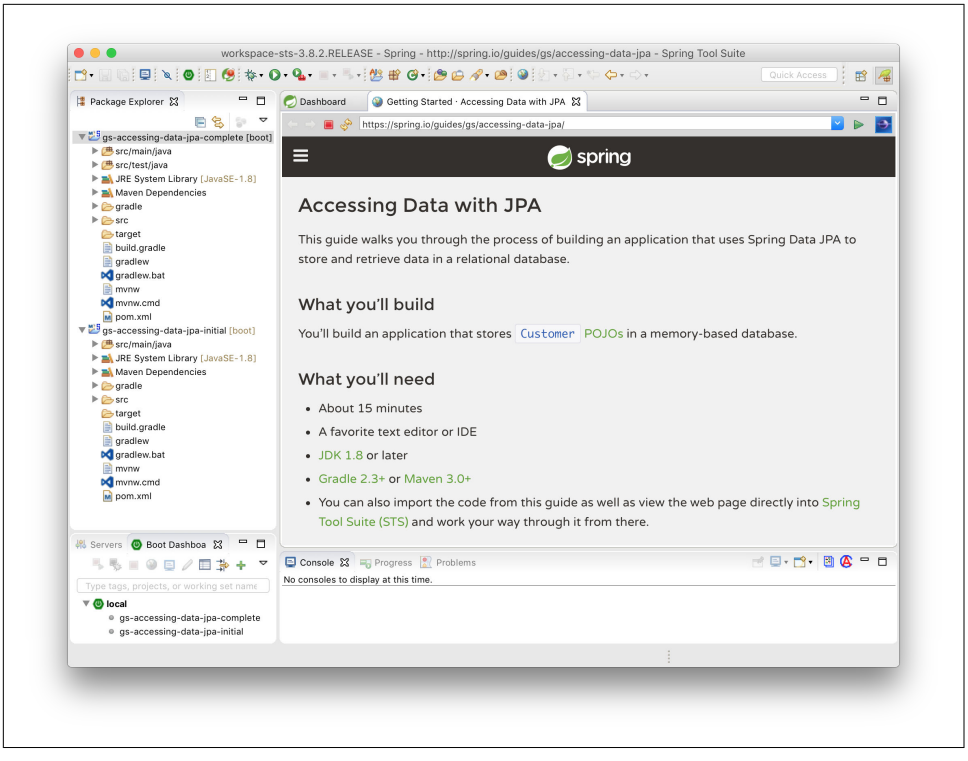


Figure 1-16. The IDE loads the guide and the relevant Git projects

Configuration

At the end of the day, a Spring application is a collection of objects. Spring manages those objects and their relationships for you, providing services to those objects as necessary. In order for Spring to support your objects - *beans* - it needs to be made aware of them. Spring provides a few different (complimentary) ways to describe your beans.

Let's suppose we have a typical layered service, with a service object in turn talking to a `javax.sql.DataSource` to talk to a (in this case) embedded database, H2. We'll need to define a service. That service will need a datasource. We could instantiate that data source in-place, where it's needed, but then we'd have to duplicate that logic whenever we wish to reuse the data source. In other objects. The resource accquisition and initialization is happening at the call site, which means that we can't reuse that logic elsewhere.

Example 1-5.

```
package com.example.raai;

import com.example.Customer;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.util.Assert;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class CustomerService {

        private final DataSource dataSource =
                        new EmbeddedDatabaseBuilder()
                                        .setName("customers")
                                        .setType(EmbeddedDatabaseType.H2)
                                        .build();

        public Collection<Customer> findAll() {
                List<Customer> customerList = new ArrayList<>();
                try {
                        try (Connection c = dataSource.getConnection()) {
                                Statement statement = c.createStatement();
                                try (ResultSet rs = statement.executeQuery("select
* from CUSTOMERS")) {
                                        while (rs.next()) {
                                                customerList.add(new Customer(
                                                                rs.getLong("ID"),
                                                                rs.get
String("EMAIL")
                                                ));
                                        }
                                }
                        }
                } catch (SQLException e) {
                        throw new RuntimeException(e);
                }
                return customerList;
        }

        public static void main(String argsp[]) throws Throwable {
```

```
                CustomerService customerService = new CustomerService();

                ❶
                DataSource dataSource = customerService.dataSource;
                DataSourceInitializer init = new DataSourceInitializer();
                init.setDataSource(dataSource);
                ResourceDatabasePopulator populator = new ResourceDatabasePopula
tor();
                populator.setScripts(new ClassPathResource("schema.sql"),
                                new ClassPathResource("data.sql"));
                init.setDatabasePopulator(populator);
                init.afterPropertiesSet();

                ❷
                int size = customerService.findAll().size();
                Assert.isTrue(size == 2);

        }
}
```

❶ it's hard to stage the datasource as the only reference to it is buried in a `private final` field in the `CustomerService` class itself. The only way to get access to that variable is by taking advantage of Java's *friend* access, where instances of a given object are able to see other instances private variables.

❷ here we use types from Spring itself, not JUnit, since the JUnit library is `test` scoped, to "exercise" the component. The Spring framework `Assert` class supports design-by-contract behavior, not unit testing!

As we can't plugin a mock datasource, and we can't stage the datasource any other way, we're forced to embed the *test* code in the component itself. This *won't* be easy to test in a proper fashion and it means we need `test`-time dependencies on the CLASSPATH for our main code.

We're using an embedded datasource and so the datasource would be the same in both development and production, but in a realistic example, the configuration of environment-specific details like usernames and hostnames would be parameterizable, otherwise any attempt to exercise the code might execute against a production datasource!

It would be cleaner to centralize the bean definition, outside of the call sites where it's used. How does our component code get access to that centralized reference? We could store them in static variables, but how do we test it since we have static references littered throughout our code? How do we *mock* the reference out? We could also store the references in some sort of shared context, like JNDI (Java Naming and Directory Interface), but we end up with the same problem: it's hard to test this arrangement without mocking out all of JNDI!

Instead of burying resource initialization and accquisition logic code in all the consumers of those resources, we could create the objects and establish their wiring in one place, in a single class. This principle is called *inversion of control*. The wiring of objects is seprate from the components themselves. In teasing these things apart, we're able to build component code that is dependant on base-types and interfaces, not coupled to a particular implementation. This is called *dependency injection*. A component that is ignorant of how and where a pariticular dependency was created won't care if that dependency is a fake (*mock*) object during a unit test.

Instead, let's move the wiring of the objects - the configuration - into a separate class, a *configuration* class. Let's look at how this is supported with Spring's Java configuration.

What about XML? Spring debuted with support for XML-based configuration. XML configuration offers a lot of the same benefits of Java configuration: it's a centralized artifact separate from the components being wired. It's still supported, but it's not the best fit for a Spring Boot application which relies on Java configuration. In this book, we won't use XML-based configuration.

Example 1-6.

```
package com.example.javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

❶
@Configuration
public class ApplicationConfiguration {

        ❷
        @Bean(destroyMethod = "shutdown")
        DataSource dataSource() {
                return new EmbeddedDatabaseBuilder()
                                .setType(EmbeddedDatabaseType.H2)
                                .setName("customers")
                                .build();
        }

        ❸
        @Bean
        CustomerService customerService(DataSource dataSource) {
                return new CustomerService(dataSource);
```

```
        }
}
```

❶ this class is a Spring `@Configuration` class, which tells Spring that it can expect to find definitions of objects and how they wire together in this class.

❷ we'll extract the definition of the `DataSource` into a bean definition. Any other Spring component can see, and work with, this single instance of the `DataSource`. If ten Spring components *depend* on the `DataSource`, then they will *all* have access to the same instance in memory, by default. This has to do with Spring's notion of scope. By default, a Spring bean is *singleton scoped*.

❸ register the `CustomerService` instance with Spring and tell Spring to satisfy the `DataSource` by sifting through the other registered beans in the application context and finding the one whose type matches the bean provider parameter.

Revisit the `CustomerService` and remove the explicit `DataSource` creation logic.

Example 1-7.

```
package com.example.javaconfig;

import com.example.Customer;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class CustomerService {

        private final DataSource dataSource;

        ❶
        public CustomerService(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public Collection<Customer> findAll() {
                List<Customer> customerList = new ArrayList<>();
                try {
                        try (Connection c = dataSource.getConnection()) {
                                Statement statement = c.createStatement();
                                try (ResultSet rs = statement.executeQuery("select
* from CUSTOMERS")) {
```

```
                                        while (rs.next()) {
                                                customerList.add(new Customer(
                                                                rs.getLong("ID"),
                                                                rs.get
String("EMAIL")
                                                ));
                                        }
                                }
                        }
                } catch (SQLException e) {
                        throw new RuntimeException(e);
                }
                return customerList;
        }
}
```

❶ the definition of the `CustomerService` type is markedly simpler, since it now merely depends on a `DataSource`. We've limited its responsibilities, arguably, to things more in scope for this type: interacting with the `dataSource`, not defining the `dataSource` itself.

The configuration is explicit, but also a bit redundant. Spring could do some of the construction for us, if we let it! After all, why should *we* do all of the heavy lifting? We could use Spring's stereotype annotations to mark our own components and let spring instantiate those for us based on convention.

Let's revisit the `ApplicationConfiguration` class and let Spring discover our stereotyped components using *component scanning*. We no longer need to explicitly describe how to construct a `CustomerService` bean, so we'll remove that definition too. The `CustomerService` type is exactly the same as before, except that it has the `@Component` annotation applied to it.

Example 1-8.

```
package com.example.componentscan;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
@ComponentScan ❶
public class ApplicationConfiguration {

        @Bean(destroyMethod = "shutdown")
```

```
        DataSource dataSource() {
                return new EmbeddedDatabaseBuilder()
                                .setType(EmbeddedDatabaseType.H2)
                                .setName("customers")
                                .build();
        }

}
```

❶ this annotation tells Spring to discover other beans in the application context by scanning the current package (or below) and looking for all objects annotated with *stereotype* annotations like `@Component`. This annotation, and others besides that are themselves annotated with `@Component`, act as sorts of markers for Spring. Tags. Spring perceives them on components and creates a new instance of the object on which they're applied. It calls the no-argument constructor by default, or it'll call a constructor with parameters so long as all the parameters themselves are satisfyable with references to other objects in the application context. Spring provides a lot of services as *opt-in* annotations expected on `@Configuration` classes.

The code in our example uses a datasource directly and we're forced to write a lot of low-level boilerplate JDBC code to get a simple result. Dependency injection is a powerful tool, but it's the least interesting aspect of Spring. Let's use one of Spring's most compelling features: the portable service abstractions, to simplify our interaction with the datasource. We'll swap out our manual and verbose JDBC-code and use Spring framework's `JdbcTemplate` instead.

Example 1-9.

```
package com.example.psa;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
@ComponentScan
public class ApplicationConfiguration {

        @Bean(destroyMethod = "shutdown")
        DataSource dataSource() {
                return new EmbeddedDatabaseBuilder()
                                .setType(EmbeddedDatabaseType.H2)
```

```
                                .setName("customers")
                                .build();
        }

        ❶
        @Bean
        JdbcTemplate jdbcTemplate(DataSource dataSource) {
                return new JdbcTemplate(dataSource);
        }

}
```

❶ the `JdbcTemplate` is one of many implementations in the Spring ecosystem of the *template pattern*. It provides convenient utility methods that make working with JDBC a one-liner for common things. It handles resource initialization and acquisition, destruction, exception handling and so much more so that we can focus on the essence of the task at hand.

With the `JdbcTemplate` in place, our revised `CustomerService` is *much* cleaner.

Example 1-10.

```
package com.example.psa;

import com.example.Customer;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.util.Collection;

@Component
public class CustomerService {

        private final JdbcTemplate jdbcTemplate;

        public CustomerService(JdbcTemplate jdbcTemplate) {
                this.jdbcTemplate = jdbcTemplate;
        }

        public Collection<Customer> findAll() {
                ❶
                RowMapper<Customer> rowMapper =
                                (rs, i) -> new Customer(rs.getLong("ID"), rs.get
String("EMAIL"));
                ❷
                return this.jdbcTemplate.query("select * from CUSTOMERS ", rowMap
per);
        }
}
```

❶ there are many overloaded variants of the `query` method, one of which expects a `RowMapper` implementation. It is a callback object that Spring will invoke for you on each returned result, allowing you to map objects returned from the database to the domain object of your systemm. The `RowMapper` interface also lends itself nicely to Java 8 lambas!

❷ the query is a trivial one-liner. Much better!

As we control the wiring in a central place, in the bean configuration, we're able to *substitute* (or *inject*) implementations with different specializations or capabilites. If we wanted to, we could change all the injected implementations to support cross-cutting concerns without altering the consumers of the beans. Suppose we wanted to support logging the time it takes to invoke all methods. We could create a class that subclasses our existing `CustomerService` and then, in method overrides, insert logging functionality before and after we invoke the `super` implementation. The logging functionality is a cross-cutting concern, but in order to inject it into the behavior of our object hierarchy we'd have to override all methods.

Ideally, we wouldn't need to go through so many hoops to interpose trivial cross-cutting concerns over objects like this. Languages (like Java) that only support single-inheritance don't provide a clean way to address this use case for any arbitrary object. Spring suppports an alternative, *aspect-oriented programming* (AOP). AOP is a larger topic than Spring, but Spring provides a very approachable subset of AOP for Spring objects. Spring's AOP support centers around the notion of an *aspect* which codifies cross-cutting behavior. A *pointcut* describes the pattern that should be matched when applying an aspect. The pattern in a pointcut is part of a full-featured pointcut language that Spring supports. The pointcut language lets you describe method invocations for objects in a Spring application. Let's suppose we wanted to create an aspect that will match all method invocations, today and tomorrow, in our `CustomerService` example and interpose logging to capture timing.

Add `@EnableAspectJAutoProxy` to the `ApplicationConfiguration` `@Configuration` class to activate Spring's AOP functionality. Then, we need only extract our cross-cutting functionality into a separate type, an `@Aspect`-annotated object.

Example 1-11.

```
package com.example.aop;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;
```

```
import java.time.LocalDateTime;

@Component
@Aspect ❶
public class LoggingAroundAspect {

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

        ❷
        @Around("execution(* com.example.aop.CustomerService.*(..))")
        public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
                LocalDateTime start = LocalDateTime.now();

                Throwable toThrow = null;
                Object returnValue = null;

                ❸
                try {
                        returnValue = joinPoint.proceed();
                } catch (Throwable t) {
                        toThrow = t;
                }
                LocalDateTime stop = LocalDateTime.now();

                log.info("starting @ " + start.toString());
                log.info("finishing @ " + stop.toString() + " with duration " +
                                stop.minusNanos(start.getNano()).getNano());

                ❹
                if (null != toThrow) throw toThrow;

                ❺
                return returnValue;
        }
}
```

❶ mark this bean as an aspect

❷ declare that this method is to be given a chance to execute *around* - before and afer - the execution of any method that matches the pointcut expression in the `@Around` annotation. There are numerous other annotations, but for now this one will give us a lot of power. For more, you might investigate Spring's support for AspectJ.

❸ when the method matching this pointcut is invoked, our aspect is invoked first, and passed a `ProceedingJoinPoint` which is a handle on the running method invocation. We can choose to interrogate the method execution, to proceed with

it, to skip it, etc. This aspect logs before and after it proceeds with the method invocation.

❹ if an exception is thrown it is cached and rethrown later

❺ if a return value is recorded it is returned as well (assuming no exception's been thrown)

We can use AOP directly, if we need to, but many of the really important cross cutting concerns we're likely going to encounter in typical application development are already extracted out for us in Spring itself. An example is declarative transaction management. In our example, we have one method which is read-only. If we were to introduce another business service method - one that mutated the database multiple times in a single service call - then we'd need to need to ensure that those mutations all happened in a single unit-of-work; either every interaction with the stateful resource (the datasource) succeeds or none of them do. We don't want to leave the system in an inconsistent state. This is an ideal example of a cross-cutting concern: we might use AOP to begin a transactional block before every method invocation in a business service and commit (or rollback) that transaction upon the completion of the invocation. We *could*, but thankfully Spring's declarative transaction support does this for us already, we don't need to write lower-level AOP-centric code to make it work. Add `@EnableTransactionManagement` to a configuration class and then delineate transactional boundries on business services using the `@Transactional` annotation.

We have a service tier, a logical next step might be to build a web application. We could use Spring MVC to create a REST endpoint. We would need to configure Spring MVC itself, and then deploy it an Servlet-compatible application server, and then confgure the application server's interaction with Servlet API. It can be a lot of work before we can take that next step and realize a humble working web application and REST endpoint!

We've used straight JDBC here but we could've elected to use an ORM layer, instead. This would've invited even more complexity. None of it is too much, step by step, but taken together the cognitive load can be overwhelming.

This is where Spring Boot and and its auto-configuration kick in.

In our first example in this chapter, we created an application with an interface, a JPA entity, a few annotations and a `public static void main` entry-point class and.. that's it! Spring Boot started up, and *presto!*, there was a working REST API running on `http://localhost:8080/cats`. The application supported manipulating JPA entities using the Spring Data JPA-based repository. It did a lot of things for which there was, seemingly, no explicit code - *magic*!

If you know much about Spring then you'll no doubt recognize that we were using, among other things, Spring framework and its robust support for JPA, Spring Data JPA to configure a declarative, interface-based repository, Spring MVC and Spring Data REST to serve an HTTP-based REST API, and even then you might be wondering about where the web server itself came from. If you know much about Spring then you'll appreciate that each of these modules requires *some* configuration. Not much, usually, but certainly more than what we did there! They require, if nothing else, an annoation to *opt-in* to certain default behavior.

Historically, Spring has opted to expose the configuration. It's a configuration plane - a chance to refine the behavior of the application. In Spring Boot, the priority is providing a sensible default and supporting easy overrides. It's a full throated embrace of convention-over-configuration. Ultimately, Spring Boot's auto-configuration is the same sort of configuration that you could write by hand, with the same annotations and the same beans you might register, along with common sense defaults.

Spring supports the notion of a service loader to support registering custom contributions to the application without changing Spring itself. A service loader is a mapping of types to Java configuration class names that are then evaluated and made available to the Spring application later on. The registration of custom contributions happens in the `META-INF/spring.factories` file. Spring Boot looks in the `spring.factories` file for, among other things, all classes under the `org.springframework.boot.autoconfigure.EnableAutoConfiguration` entry. In the `spring-boot-autoconfigure .jar` that ships with the Spring Boot framework itself there are *dozens* of different configuration classes here, and Spring Boot will try to evaluate them all! It'll try, but *fail*, to evaluate all of them, at least, thanks to various conditions - guards, if you like - that have been placed on the configuration classes and the `@Bean` definitions therein. This facility, to conditionally register Spring components, is from Spring framework itself, on which Spring Boot is built. These conditions are wide-ranging: they test for the availability of beans of a given type, the presence of environment properties, the availability of certain types on the CLASSPATH, and more.

Let's review our `CustomerService` example. We want to build a Spring Boot-version of the application that uses an embedded database, the Spring framework `JdbcTemplate` and then support building a web application. Spring Boot will do all of that for us. Revisit the `ApplicationConfiguration` and turn it into a Spring Boot application accordingly:

Example 1-12. `ApplicationConfiguration.java` class

```
package com.example.boot;

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

```
@SpringBootApplication
public class ApplicationConfiguration {
        ❶
}
```

❶ Look ma, no configuration! Spring Boot will contribute all the things that we contributed ourselves, manually, and so much more.

Let's introduce a Spring MVC-based controller to expose a REST endpoint to respond to HTTP `GET` requests at `/customers`.

Example 1-13. `CustomerRestController.java` class

```
package com.example.boot;

import com.example.Customer;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collection;

@RestController ❶
public class CustomerRestController {

        private final CustomerService customerService;

        public CustomerRestController(CustomerService customerService) {
                this.customerService = customerService;
        }

        ❷
        @GetMapping("/customers")
        public Collection<Customer> readAll() {
                return this.customerService.findAll();
        }
}
```

❶ `@RestController` is another stereotype annotation, like `@Component`. It tells Spring that this component is to serve as a REST controller.

❷ We can use Spring MVC annotations that map to the domain of what we're trying to do, in this case, to handle HTTP `GET` requests to a certain endpoint with `@GetMapping`

You could create an entry point in the `ApplicationConfiguration` class and then run the application and visit `http://localhost:8080/customers` in your browser. Cognitively, it's much easier to reason about what's happening in our business logic *and* we've done more in less!

Example 1-14.

```
//...

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

//...
```

We got more for less, but we know that somewhere something is doing the same configuration as we did explicitly, before. Where is the `JdbcTemplate` being created? It's created in an auto-configuration class called `JdbcTemplateAutoConfiguration`, whose definition is (roughly):

Example 1-15. `JdbcTemplateAutoConfiguration`

```
@Configuration ❶
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class }) ❷
@ConditionalOnSingleCandidate(DataSource.class) ❸
@AutoConfigureAfter(DataSourceAutoConfiguration.class) ❹
public class JdbcTemplateAutoConfiguration {

        private final DataSource dataSource;

        public JdbcTemplateAutoConfiguration(DataSource dataSource) {
                this.dataSource = dataSource;
        }

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

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

}
```

❶ this is a normal `@Configuration` class

❷ the configuration class should *only* be evaluated if the type `DataSource.class` and `JdbcTemplate.class` are somewhere on the classpath, otherwise this would no doubt fail with an error like `ClassNotFoundException`

❸ we only want this configuration class if, somewhere in the application context, a `DataSource` bean has been contributed.

❹ we know that `DataSourceAutoConfiguration` will contribute an embedded H2 datasource if we let it, so this annotation ensures that this configuration happens **after** that `DataSourceAutoConfiguration` has run. If a database is contributed, then this configuration class will evaluate.

❺ we want to contribute a `JdbcTemplate` but only so long as the user (that's to say, so long as you and I) haven't already defined a bean of the same type in our own configuration classes.

Spring Boot's auto-configuration greatly reduces the actual code count and the cognitive load associated with that code. It frees us to focus on the essence of the business logic, leaving the tedium to the framework. If we want to exert control over some aspect of the stack we're free to contribute beans of certain types and those will be plugged in to the framework for us. Spring Boot is an implementation of the open-closed principal: it's open for extension but closed for modification. You don't *need* to recompile Spring or Spring Boot in order to override parts of the machine. You may see your Spring Boot application exhibit some odd behavior that you want to override or customize. It's important to know how to customise the application and how to debug it. Specify the `--Ddebug=true` flag when the application starts up and Spring Boot will print out the Debug Report, showing you all the conditions that have been evaluated and whether they're a positive or negative match. From there, it's easy to examine what's happening in the relevant auto-configuration class to ascertain its behavior.

Cloud Foundry

Spring Boot lets us focus on the essence of the application itself, but what good would all that newfound productivity be if we then lost it all struggling to move the application to production? Operationalizing an application is a daunting task, but one that must not be ignored. Our goal is to continuously deploy our application into a production-like environment, to validate in integration and acceptance tests, that things will work when they're deployed to production. It's important to get to production as early and often as possible for that is the only place where the customer, the party most invested in the outcome of a team's deliverable, can validate that it works. If moving to production is arduous, inevitably, a software team will come to fear the

process and hesitate, increasing the delta between development and production pushes. This increases the backlog of work that hasn't been moved to production, which means that each push becomes more risky because there's more business value in each release. In order to de-risk the move to production, reduce the batch of work and increase the frequency of deployment. If there's a mistake in production, it should be as cheap as possible to fix it and deploy that fix.

The trick is to automate away everything that can be automated in the value chain from product management to production that doesn't add value to the process. Deployment is not a business differentiating activity; it adds no value to the process. It should be completely automated. You get velocity through automation.

Cloud Foundry is a cloud platform that wants to help. It's a Platform-as-a-service (PaaS). Cloud Foundry focuses not on hard disks, RAM, CPU, Linux installations and security patches as you would with an Infrastructure-as-a-service (IaaS) offering, or on containers as you might in a container-as-a-service offering, but instead on applications and their services. Operators focus on applications and their services, nothing else. We'll see Cloud Foundry throughout this book, so far now let's focus on deploying the Spring Boot application we iteratively developed when looking at Spring's configuration support.

There are multiple implementations of Cloud Foundry, all based on the open-source Cloud Foundry code. For this book we've run and deployed all applications on Pivotal Web Services. Pivotal Web Services offers a subset of the functionality of Pivotal's on-premise Pivotal Cloud Foundry offering. It's hosted on Amazon Web Services, in the AWS East region. If you want to take your first steps with Cloud Foundry, PWS is an affordable and approachable option that serves projects large and small every day and is maintained by the operations team at Pivotal who turns that know-how that gets driven back into the product.

Go to the main page. You'll find a place to sign in an existing account, or register new accounts.

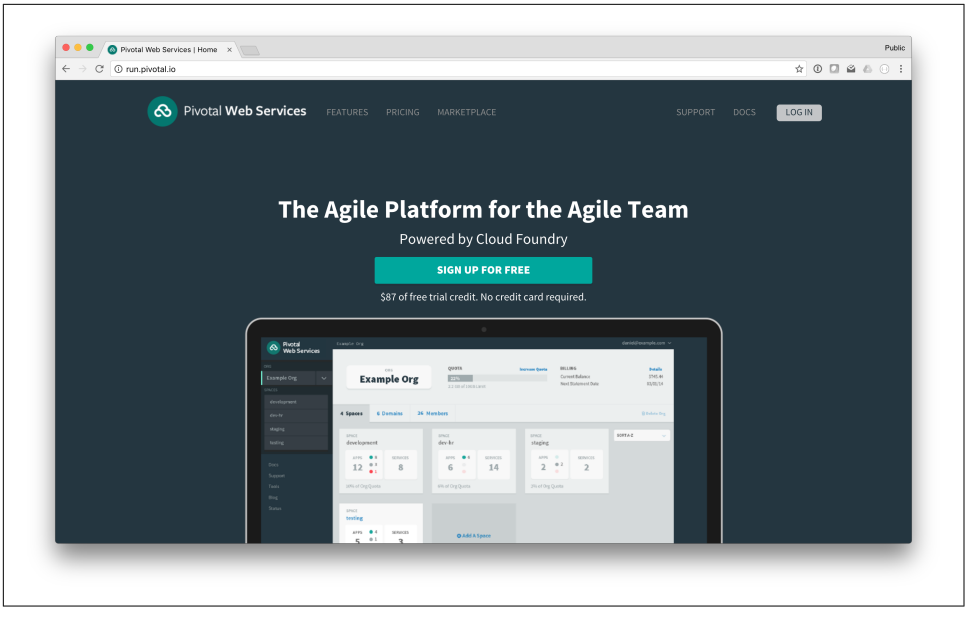


Figure 1-17. The PWS main page

Once logged in you can interact with your account and get information about deployed applications.

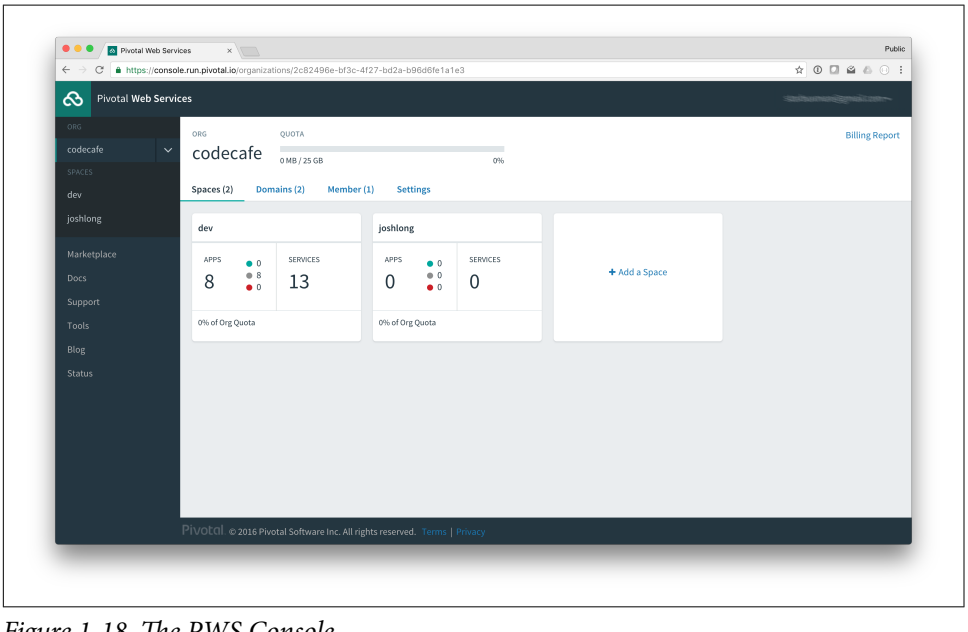


Figure 1-18. The PWS Console

Once you have an account, make sure you have the `cf` CLI. You'll need to login if you haven't before.

Target the appropriate Cloud Foundry instance using `cf target api.run.pivotal.io` and then login using `cf login`.

Example 1-16. authenticating and targeting a Cloud Foundry organization and a space

```
   cf login

API endpoint: https://api.run.pivotal.io

Email> email@gmail.com

Password>
Authenticating...
OK

Select an org (or press enter to skip):
1. marketing
2. sales
2. back-office

Org> 1
Targeted org back-office

Targeted space development

API endpoint:   https://api.run.pivotal.io (API version: 2.65.0)
User:           email@gmail.com
Org:            back-office     # 1
Space:          development    # 2
```

❶ in Cloud Foundry, there might be many organizations to which a given user has access.

❷ within that organization, there might be multiple environments (for example, `development`, `staging`, and `integration`).

Now that we have a valid account and are logged in, we can deploy our existing application, in `target/configuration.jar`. We'll need to provision a database using the `cf create-service` command.

Example 1-17. create a MySQL database, bind it to our application and then start the application

```
cf create-service p-mysql 100mb bootcamp-customers-mysql            # 1

cf push -p target/configuration.jar bootcamp-customers \
```

```
 --random-route --no-start                                    # 2

cf bind-service bootcamp-customers bootcamp-customers-mysql   # 3

cf start bootcamp-customers                                   # 4
```

❶ first we need to provision a MySQL database from the MySQL service (called `p-mysql`) at the `100mb` plan. Here, we'll assign it a logical name, `bootcamp-customers-mysql`. Use `cf marketplace` to enumerate other services in a Cloud Foundry instance's service catalog.

❷ then push the application's artifact, `target/configuration.jar`, to Cloud Foundry. We assign the application and a random route (a URL, in this case under the PWS `cfapps.io` domain), but we don't want to start it yet: it still needs a database!

❸ the database exists, but nobody can talk to it unless we bind it to an application. Ultimately, *binding* results in environment variables being exposed in the application with the relevant connection information in the environmental variables.

❹ now that the application is bound to a database, we can finally start it up!

When you push the application to Cloud Foundry, you're giving it an application binary, a `.jar`, not a virtual machine or a Linux container (though, you *could* give it a Linux container). When Cloud Foundry receives the `.jar` it will try to determine what the nature of the application is - is it a Java application? A Ruby application? A .NET application? It'll eventually arrive at a Java application. It will pass the `.jar` to the Java buildpack. A buildpack is a directory full of scripts invoked according to a well-known lifecycle. You can override the default buildpack used by specifying its URL, or you can let the default buildpack kick in. There are buildpacks for all manner of different languages and platforms including Java, .NET, Node.js, Go, Python, Ruby, and a slew more. The Java buildpack will realize that the application is an executable `main(String [] args)` method, and that it is self-contained. It will pull down the latest OpenJDK version, and specify that our application is to run. All of this configuration is packaged into a Linux container which Cloud Foundry's scheduler will then deploy across a cluster. Cloud Foundry can run hundreds of thousands of containers in a single environment.

In no time at all, the application will spin up and report a URL on the console at which the application has been exposed. *Congratulations*! Our application is now deployed to a Cloud Foundry instance.

In our application, we have one datasource bean and so Cloud Foundry re-mapped it automatically to the single, bound MySQL service, overwriting the default embedded H2 datasource definition with one that points to a MySQL datasource.

There are a lot of aspects to the deployed application that we may want to describe on each deployment. In our first run, we used various `cf` incantations to configure the application, but that could get tedious, quickly. Instead, let's capture our application's configuration in a Cloud Foundry manifest file, typically named `manifest.yml`. Here's a `manifest.yml` for our application that will work so long as we've already got a MySQL datasource provisioned with the same name as specified earlier.

Example 1-18. a Cloud Foundry manifest

```
---
applications:
- name: bootcamp-customers                                          # 1
  buildpack: https://github.com/cloudfoundry/java-buildpack.git     # 2
  instances: 1
  random-route: true
  path: target/configuration.jar                                    # 3
  services:
    - bootcamp-customers-mysql                                      # 4
  env:
    DEBUG: "true"
    SPRING_PROFILES_ACTIVE: cloud                                   # 5
```

❶ we provide a logical name for the application

❷ specify a buildpack

❸ specify which binary to use

❹ specify a dependency on provisioned Cloud Foundry services

❺ specify environment variables to override properties to which Spring Boot will respond. `--Ddebug=true` (or `DEBUG: true`) enumerates the conditions in the auto-configuration and `--Dspring.profiles.active=cloud` specifies which profiles, or logical groupings with arbitrary names, should be activated in a Spring application. This configuration says to start all Spring beans without any profile as well as those with the `cloud` profile.

Now, instead of writing all those `cf` incantations, just run `cf push -f manifest.yml`. Soon, your application will be up and running and ready for inspection.

Next Steps

In this chapter we looked every so briefly at getting started with Spring Boot and supporting tools like the Spring Tool Suite, creating Java configuration, and then moving that application to a cloud environment. We've automated the deployment of the code

to a production environment, and it wouldn't be hard to standup a continuous integration flow using Jenkins or Bamboo. In the next chapters, we'll look more heavily at all things Spring Boot and Cloud Foundry.

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, at every level, from development to operations.

Amazon's Story

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, tablets and watches 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 boundries 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 cloud native applications share: the idea that teams own what they 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 oppor-

tunity 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.

the Promise of a Platform

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.

Platforms are able to automate the tasks that are not essential to supporting the business requirements of an application. 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.

Any team that has written shell scripts or stamped containers or virtual machines to automate deployment has built a platform, of sorts. The question is: what promises can that platform keep? How much work would it take to support the majority (or even all) requirements for continuously delivering new software?

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.

- **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.

Ideas:

- *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 reduce the time spent doing 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 quickly building and operating. 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.

Scalability

To develop software faster, we are required to think about scale at all levels. Scale, in a most general sense, is a function of cost that produces value. The level of unpredictability that reduces that value is called risk. We are forced to frame scale in this context because building software is fraught with risks. The risks that are being created by software developers are not always known to operators. By demanding that developers deliver features to production at a faster pace, we are adding to the risks of its operation without having a sense of empathy for its operators.

The result of this is that operators grow distrustful of the software that developers produce. The lack of trust between developers and operators creates a blame culture

that tends to confuse the causes of failures that impact the creation of value for the business.

To alleviate the strain that is being put on traditional structures of an IT organization, we need to rethink how software delivery and operations teams communicate. Communication between operations and developers can affect our ability to scale, as the goals of each party tends to become misaligned over time. To succeed at this requires a shift towards a more reliable kind of software development, one that puts emphasis on the experience of an operations team inside the software development process; one that fosters shared learning and improvement.

Reliability

The expectations that are created between teams, be it operations, development, or user experience design, we can think of these expectations as contracts. The contracts that are created between teams imply some level of service is provided or consumed. By looking at how teams provide services to one another in the process of developing software, we can better understand how failures in communication can introduce risk that lead to failures down the road.

Service agreements between teams are created in order to reduce the risk of unexpected behavior in the overall functions of scale that produce value for a business. A service agreement between teams is made explicit in order to guarantee that behaviors are consistent with the expected cost of operations. In this way, services enable units of a business to maximize its total output. The goal here for a software business is to reliably predict the creation of value through cost. The result of this goal is what we call *reliability*.

The service model for a business is the same model that we use when we build software. This is how we guarantee the reliability of a system, whether it be in the software that we produce to automate a business function or in the people that we train to perform a manual operation.

Agility

We are beginning to find that there is no longer only one way to develop and operate software. Driven by the adoption of agile methodologies and a move towards *Software as a Service* business models, the enterprise application stack is becoming increasingly distributed. Developing distributed systems is a complex undertaking. The move towards a more distributed application architecture for companies is being fueled by the need to deliver software faster and with less risk of failure.

The modern day software-defined business is seeking to restructure their development processes to enable faster delivery of software projects and continuous deploy-

ment of applications into production. Not only are companies wanting to increase the rate in which they develop software applications, but they also want to increase the number of software applications that are created and operated to serve the various business units of an organization.

Software is increasingly becoming a competitive advantage for companies. Better and better tools are enabling business experts to open up new sources of revenue, or to optimize business functions in ways that lead to rapid innovation.

At the heart of this movement is *the cloud*. When we talk about the cloud, we are talking about a very specific set of technologies that enable developers and operators to take advantage of web services that exist to provision and manage virtualized computing infrastructure.

Companies are starting to move out of the data center and into public clouds. One such company is the popular subscription-based streaming media 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.

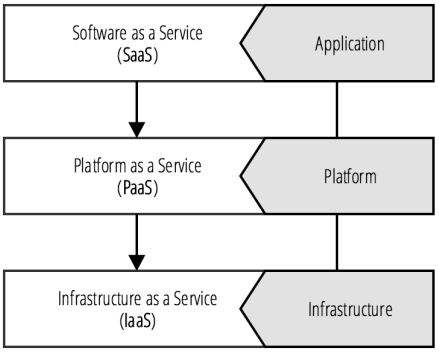
Furthermore, Izrailevsky stated that the benefits of scalability in the cloud became more evident when looking at its rapid global expansion. "In order to give our European customers a better low-latency experience, we launched a second cloud region in Ireland. Spinning up a new data center in a different 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.

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.

Cloud Native Java

Netflix has provided the software industry with a wealth of knowledge as a result of their move to become a cloud native company. This book is going to focus taking the learnings and open source projects by Netflix and apply them as a set of patterns with two central themes:

- Building resilient distributed systems using Spring and Netflix OSS
- Using continuous delivery to operate cloud native applications with Cloud Foundry

The first stop on our journey will be to understand a set of terms and concepts that we will use throughout this book to describe building and operating cloud native applications.

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.

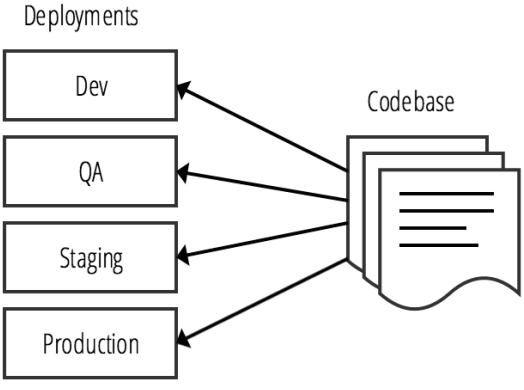
Outside of the 12-factor website—which covers each of the twelve factors in detail—there are full books that have been written that expand even greater details on each constraint. The twelve-factor methodology is now used in some application frameworks to help developers comply with some, or even all,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

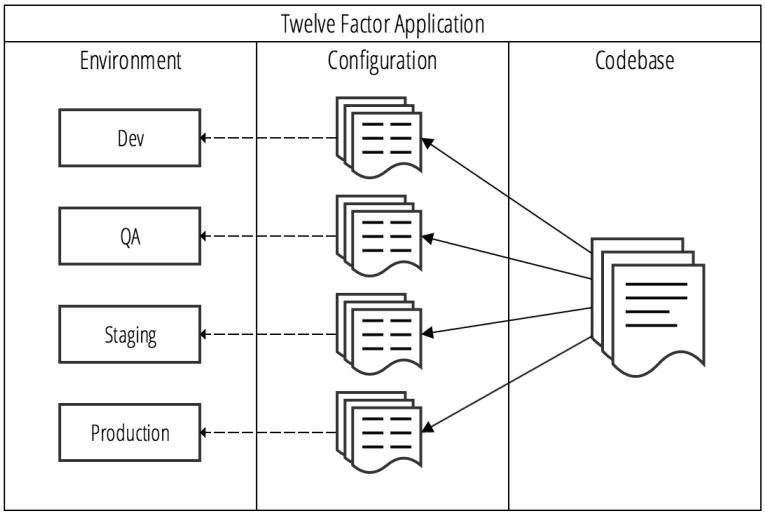
Config

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.

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
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 communciation 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 utiilization 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 steps which in turn have optional `ItemReader`, `ItemProcessor` and `ItemWriter`.

Figure 3-1. 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. I am processor is typically something that you would provide. Prosser 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 3-1. 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);
        }
}
```

❶ we use the `StepBuilderFactory` to define a lone step with a `ItemReader` and `ItemWriter` and a chunk size of 5. Every 5 records read will be written in a single chunk.

❷ this step will skip recordswhere an `InvalidEmailException` is thrown. In this case, the `processor` throws that exception if the email doesn't validate.

❸ this code uses a third party API to validate emails. If the service call should fail (resulting in a `HttpStatusCodeException`), the step will retry the processing up to two times before aborting.

❹ we use the `JobBuilderFactory` to define a simple `Job` with only one step.

❺ the `FlatFileItemReader` reads data from a file, delegating a configured `LineMapper<T>` to properly parse the lines. Note that the `ItemReader` is annotated with `@StepScope`. Beans of this scope are recreated each time the `Step` is evaluated, in contrast to the default Spring bean scope, `singleton`.

❻ the item processor attempts to validate the email by calling a potentially shaky service-to-service call.

❼ The `JdbcBatchItemWriter` 's `BeanPropertyItemSqlParameterSourceProvider` uses convention to line up database columns with JavaBean properties on the `Customer` POJO to create new instances of that type.

❽ Our data is comma-separated. Each column is delimited with a comma (`,`) and in well-known columns that line up to JavaBean properties on the `Customer` POJO.

Spring Batch is stateful; it keeps medatata 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 paritcularly 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 3-2. Spring Retry helps us retry, with increasing backoff periods between attempts, to invoke potentially flaky downstream services.

```
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.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Recover;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;
```

```
import java.util.Date;
import java.util.Map;

@Component
public class RetryableGreetingClient implements GreetingClient {

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

        private final RestTemplate restTemplate;

        private final String serviceUri;

        @Autowired
        public RetryableGreetingClient(RestTemplate restTemplate,
                                       @Value("${greeting-service.domain:
127.0.0.1}") String domain,
                                       @Value("${greeting-service.port:8080}") int
port) {
                this.restTemplate = restTemplate;
                this.serviceUri = "http://" + domain + ":" + port + "/hi/{name}";
        }

        ❶
        @Retryable(include = Exception.class,
                        maxAttempts = 4,
                        backoff = @Backoff(multiplier = 5))
        @Override
        public String greet(String name) {
                long time = System.currentTimeMillis();

                Date now = new Date(time);

                this.log.info("attempting to call the greeting-service "
                                + time + "/" + now.toString());

                ParameterizedTypeReference<Map<String, String>> ptr =
                                new ParameterizedTypeReference<Map<String,
String>>() { };

                return this.restTemplate
                                .exchange(this.serviceUri, HttpMethod.GET, null,
ptr, name)
                                .getBody()
                                .get("greeting");
        }

        ❷
        @Recover
        public String recoverForGreeting(Exception e) {
                return "OHAI";
        }
```

```
}
```

❶ 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`

❷ 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 3-3. this circuit breaker gives our downstream service time to breathe

```
package demo;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
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.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Date;
import java.util.Map;

@Component
public class CircuitBreakerGreetingClient implements GreetingClient {

        private Log log = LogFactory.getLog(getClass());
        private final RestTemplate restTemplate;
        private final String serviceUri;
```

```
        @Autowired
        public CircuitBreakerGreetingClient(RestTemplate restTemplate,
                        @Value("${greeting-service.domain:127.0.0.1}") String
domain,
                        @Value("${greeting-service.port:8080}") int port) {
                this.restTemplate = restTemplate;
                this.serviceUri = "http://" + domain + ":" + port + "/hi/{name}";
        }

        @Override
        @HystrixCommand(fallbackMethod = "fallback")
        public String greet(String name) {
                long time = System.currentTimeMillis();
                Date now = new Date(time);
                this.log.info("attempting to call " + "the greeting-service " + time
                                + "/" + now.toString());

                ParameterizedTypeReference<Map<String, String>> ptr = new Parameteri
zedTypeReference<Map<String, String>>() {
                };

                return this.restTemplate
                                .exchange(this.serviceUri, HttpMethod.GET, null,
ptr, name)
                                .getBody().get("greeting");
        }

        public String fallback(String name) {
                return "OHAI";
        }

}
```

❶ 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`

❷ 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 Hystix 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 invevitable 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 3-4.

```
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(taskExecu
tion.toString()));
        }

        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 workoads 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 capabiltiies, 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. Conceptu-

ally, the signup process is a simple affair: the user submitsa 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 sophiticated 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.

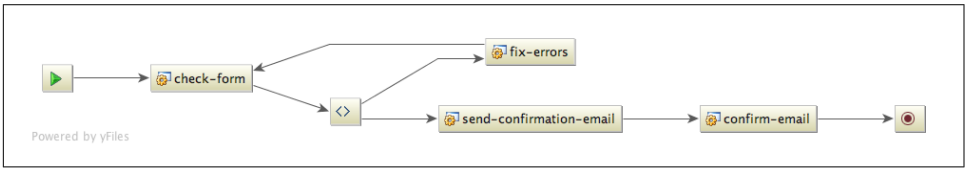


Figure 3-2. a signup process modeled using BPMN 2.0 as viewed from IntelliJ IDEA's yFiles-powered BPMN 2.0 preview.

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 defintional. 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 applcation, by default. Let's take a look at the markup for the BPMN 2.0 `signup` process.

Example 3-5. 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">
        ❶
        <startEvent id="start"/>

        ❷
        <sequenceFlow sourceRef="start" targetRef="check-form"/>

        ❸
        <serviceTask id="check-form" name="check-form" activiti:expression="#{check
Form.execute(execution)}"/>

        <sequenceFlow sourceRef="check-form" targetRef="form-completed-decision-
gateway"/>

        ❹
        <exclusiveGateway id="form-completed-decision-gateway"/>

        <sequenceFlow name="formOK" id="formOK" sourceRef="form-completed-decision-
gateway" targetRef="send-confirmation-email">
            <conditionExpression xsi:type="tFormalExpression">${formOK == true}</
conditionExpression>
        </sequenceFlow>

        <sequenceFlow  id="formNotOK" name="formNotOK" sourceRef="form-completed-
decision-gateway" targetRef="fix-errors">
            <conditionExpression  xsi:type="tFormalExpression">${formOK == false}</
conditionExpression>
        </sequenceFlow>

        ❺
        <userTask name="fix-errors" id="fix-errors">
            <humanPerformer>
                <resourceAssignmentExpression>
                    <formalExpression>customer</formalExpression>
                </resourceAssignmentExpression>
            </humanPerformer>
        </userTask>

        <sequenceFlow sourceRef="fix-errors" targetRef="check-form"/>

        ❻
        <serviceTask id="send-confirmation-email" name="send-confirmation-email"
activiti:expression="#{sendConfirmationEmail.execute(execution)}"/>

        <sequenceFlow sourceRef="send-confirmation-email" targetRef="confirm-
email"/>

        ❼
        <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>
```

❶ the first state is the `startEvent`. All processes have a defined `start` and `end`.

❷ 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.

❸ 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.

❹ 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 elesewhere. This process expects one input process variable, `customerId`.

❺ 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

❻ 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.

❼ 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 3-6. 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 = "customerId";

        private final RuntimeService runtimeService;
        private final TaskService taskService;
        private final CustomerRepository customerRepository;
        private Log log = LogFactory.getLog(getClass());

        ❶
        @Autowired
        public SignupRestController(RuntimeService runtimeService,
                        TaskService taskService, CustomerRepository repository) {
```

```
                this.runtimeService = runtimeService;
                this.taskService = taskService;
                this.customerRepository = repository;
        }

        ❷
        @RequestMapping(method = RequestMethod.POST)
        public ResponseEntity<?> startProcess(@RequestBody Customer customer) {
                Assert.notNull(customer);
                Customer save = this.customerRepository.save(new Customer(customer
                                .getFirstName(), customer.getLastName(), cus
tomer.getEmail()));

                String processInstanceId = this.runtimeService
                                .startProcessInstanceByKey(
                                                "signup",
                                                Collections.singletonMap(CUS
TOMER_ID_PV_KEY,

Long.toString(save.getId()))).getId();
                this.log.info("started sign-up. the processInstance ID is "
                                + processInstanceId);

                return ResponseEntity.ok(save.getId());
        }

        ❸
        @RequestMapping(method = RequestMethod.GET, value = "/{customerId}/signup/
errors")
        public List<String> readErrors(@PathVariable String customerId) {
                return this.taskService.createTaskQuery().active()
                                .taskName("fix-errors").includeProcessVariables()
                                .processVariableValueEquals(CUSTOMER_ID_PV_KEY, cus
tomerId)
                                .list().stream().map(TaskInfo::getId)
                                .collect(Collectors.toList());
        }

        ❹
        @RequestMapping(method = RequestMethod.POST, value = "/{customerId}/signup/
errors/{taskId}")
        public void fixErrors(@PathVariable String customerId,
                        @PathVariable String taskId, @RequestBody Customer fixedCus
tomer) {

                Customer customer = this.customerRepository.findOne(Long
                                .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(CUSTOMER_ID_PV_KEY, cus
tomerId)
                                .list()
                                .forEach(
                                                t -> {
                                                        log.info("fixing customer#
" + customerId
                                                                        + " for tas
kId " + taskId);
                                                        taskService.com
plete(t.getId(),
                                                                        Collec
tions.singletonMap("formOK", true));
                                                });
        }

        ❺
        @RequestMapping(method = RequestMethod.POST, value = "/{customerId}/signup/
confirmation")
        public void confirm(@PathVariable String customerId) {
                this.taskService.createTaskQuery().active().taskName("confirm-
email")
                                .includeProcessVariables()
                                .processVariableValueEquals(CUSTOMER_ID_PV_KEY, cus
tomerId)
                                .list().forEach(t -> {
                                        log.info(t.toString());
                                        taskService.complete(t.getId());
                                });
                this.log.info("confirmed email receipt for " + customerId);
        }
}
```

❶ 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.

❷ 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 specificed 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.

❸ 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.

❹ 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.

❺ 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 3-7. 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; ❶
        private final CustomerRepository customerRepository;
        private final EmailValidationService emailValidationService;

        @Autowired
        public CheckForm(EmailValidationService emailValidationService,
                        RuntimeService runtimeService, CustomerRepository customerRe
pository) {
                this.runtimeService = runtimeService;
                this.customerRepository = customerRepository;
                this.emailValidationService = emailValidationService;
        }

        ❷
        public void execute(ActivityExecution e) throws Exception {
                Long customerId = Long.parseLong(e.getVariable("customerId",
                                String.class));
                Map<String, Object> vars = Collections.singletonMap("formOK",
                                validated(this.customerRepository.findOne(custom
erId)));
                this.runtimeService.setVariables(e.getId(), vars); ❸
        }

        private boolean validated(Customer customer) {
                return !isEmpty(customer.getFirstName())
                                && !isEmpty(customer.getLastName())
                                && this.emailValidationService
                                                .isEmailValid(customer.getEmail());
        }

}
```

❶ inject the Activiti `RuntimeService`

❷ the `ActivityExecution` is a context object. You can use it to access process variables, the process engine itself, and other interesting services.

❸ then use it to artifculate 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 int he 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 int erms 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 3-8. 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 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 fowarded 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 3-9. 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.MessageChannels;
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/in}") File direc
tory) {
                // @formatter:off
                return IntegrationFlows
                                (1)
                                .from(Files.inboundAdapter(directory).autoCreateDir
ectory(true),
                                                consumer -> consumer.poller(poller -
> poller
                                                                .fixedRate(1000)))
                                (2)
                                .handle(File.class, (file, headers) -> {
                                        log.info("we noticed a new file, " + file);
                                        return file;
                                })
                                (3)
                                .routeToRecipients(
                                                spec -> spec
                                                                .recipient(csv(),
                                                                                msg
-> hasExt(msg.getPayload(), ".csv"))
                                                                .recipient(txt(),
                                                                                msg
-> hasExt(msg.getPayload(), ".txt")))
                                .get();
                // @formatter:on
        }

        boolean hasExt(Object f, String ext) {
                File file = File.class.cast(f);
                return file.getName().toLowerCase().endsWith(ext.toLowerCase());
        }

        (4)
        @Bean
        MessageChannel txt() {
                return MessageChannels.direct().get();
        }

        (5)
        @Bean
        MessageChannel csv() {
                return MessageChannels.direct().get();
        }
```

```
        ❻
        @Bean
        IntegrationFlow txtFlow() {
                return IntegrationFlows.from(txt()).handle(File.class, (f, h) -> {
                        log.info("file is .txt!");
                        return null;
                }).get();
        }

        ❼
        @Bean
        IntegrationFlow csvFlow() {
                return IntegrationFlows.from(csv()).handle(File.class, (f, h) -> {
                        log.info("file is .csv!");
                        return null;
                }).get();
        }
}
```

❶ 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.

❷ this method announces that we've received a file and then forward the payload onward

❸ route the request to one of two possible integration flows, derived from the extension of the file, through well-known `MessageChannel` instances

❹ the channel through which all files with the `.txt` extension will travel

❺ the channel through which all files with the `.csv` extension will travel

❻ the `IntegrationFlow` to handle files with `.txt`

❼ 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 `JobParameters` 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 3-10. 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.JobLaunchRequest;
import org.springframework.batch.integration.launch.JobLaunchingGateway;
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.MessageChannels;
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/in}") File direc
tory,
                        JobLauncher launcher, Job job) {
                // @formatter:off
                return IntegrationFlows
                                .from(Files.inboundAdapter(directory).autoCreateDir
ectory(true),
                                                consumer -> consumer.poller(poller -
> poller
                                                                .fixedRate(1000)))
                                .handle(File.class,
                                                (file, headers) -> {

                                                        ❷
                                                        JobParameters parameters =
new JobParametersBuilder()
                                                                        .addParame
ter(

"file",

new JobParameter(file

.getAbsolutePath()))
                                                                        .toJobParame
ters();

                                                        return MessageBuilder
                                                                        .withPay
load(

new JobLaunchRequest(job,

parameters))
                                                                        .setHeader(F
ileHeaders.ORIGINAL_FILE,

file.getAbsolutePath())
                                                                        .copyHeader
sIfAbsent(headers).build();
                                                })
                                ❸
                                .handle(new JobLaunchingGateway(launcher))
                                .routeToRecipients(
```

```
                                                spec -> spec.recipient(invalid(),
                                                                ms -> !jobFin
ished(ms.getPayload()))
                                                                .recipient(comple
ted(),
                                                                                ms -
> jobFinished(ms.getPayload())))
                                .get();
                // @formatter:on
        }

        ❹
        private boolean jobFinished(Object payload) {
                return JobExecution.class.cast(payload).getExitStatus()
                                .equals(ExitStatus.COMPLETED);
        }

        @Bean
        IntegrationFlow finishedJobsFlow(
                        @Value("${completed-directory:${HOME}/Desktop/completed}")
File finished,
                        JdbcTemplate template) {
                // @formatter:off
                return IntegrationFlows
                                .from(completed())
                                .handle(JobExecution.class,
                                                (je, headers) -> {
                                                        String ogFileName =
String.class.cast(headers
                                                                        .get(File
Headers.ORIGINAL_FILE));
                                                        File file = new File(ogFile
Name);
                                                        mv(file, finished);
                                                        List<Contact> contacts = tem
plate.query(
                                                                        "select *
from CONTACT",
                                                                        (rs, i) ->
new Contact(rs
.getString("full_name"), rs
.getString("email"), rs
.getLong("id")));
                                                        contacts.forEach(log::info);
                                                        return null;
                                                }).get();
                // @formatter:on
        }
```

```
        @Bean
        IntegrationFlow invalidFileFlow(
                        @Value("${error-directory:${HOME}/Desktop/errors}") File
errors) {
                // @formatter:off
                return IntegrationFlows
                                .from(this.invalid())
                                .handle(JobExecution.class,
                                                (je, headers) -> {
                                                        String ogFileName =
String.class.cast(headers
                                                                        .get(File
Headers.ORIGINAL_FILE));
                                                        File file = new File(ogFile
Name);
                                                        mv(file, errors);
                                                        return null;
                                                }).get();
                // @formatter:on
        }

        private void mv(File in, File out) {
                // @formatter:off
                try {
                        Assert.isTrue(out.exists() || out.mkdirs());
                        try (InputStream inStream = new BufferedInputStream(
                                        new FileInputStream(in));
                                        OutputStream outStream = new BufferedOutput
Stream(
                                                        new FileOutputStream(out)))
{
                                StreamUtils.copy(inStream, outStream);
                        }
                } catch (Exception e) {
                        throw new RuntimeException(e);
                }
                // @formatter:on
        }
}
```

❶ this `IntegrationFlow` starts off the same as in the previous example

❷ setup some `JobParameters` for our Spring Batch job using the `JobParameters Builder`

❸ forward the job and associated parameters to a `JobLaunchingGateway`

❹ 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 anomolous 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 neeed 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 brok-

ers with known interaction modes. We could explicitly configure inbound and outbound RabbitMQ or Apache Kafka adapers, of course. Instead, let's move up the stack a little bit, to simplify our work and remove the congnitive 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 unadulturated 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 3-11. a simple `MessageChannel`-centric greetings producer

```
package stream.producer;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface ProducerChannels {

        (1)
        String DIRECT = "directGreetings";
        String BROADCAST = "broadcastGreetings";

        @Output(DIRECT)
        (2)
        MessageChannel directGreetings();

        @Output(BROADCAST)
        MessageChannel broadcastGreetings();
}
```

(1) 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 3-12. a simple `MessageChannel`-centric greetings producer

```
(1)
spring.cloud.stream.bindings.broadcastGreetings.destination = greetings-pub-sub
spring.cloud.stream.bindings.directGreetings.destination = greetings-p2p

(2)
spring.rabbitmq.addresses=localhost
```

❶ 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.

❷ we're using Spring Boot's auto-configuration to create a RabbitMQ `ConnectionFactory` 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 3-13. 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);
        }
}

@RestController
class GreetingProducer {

        private final MessageChannel broadcast, direct;

        ❷
        @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 + "!";

                ❸
                this.direct.send(MessageBuilder.withPayload("Direct: " + message)
                                .build());

                this.broadcast.send(MessageBuilder.withPayload("Broadcast: " + mes
sage)
                                .build());
                return ResponseEntity.ok(message);
        }
}
```

❶ the `@EnableBinding` annotation activates Spring Cloud Stream

❷ 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}`.

❸ 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 workign 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 3-14. 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.IntegrationComponentScan;
```

```
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)
❶
@IntegrationComponentScan
❷
public class StreamProducer {

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

❸
@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;

        ❹
        @Autowired
        GreetingProducer(GreetingGateway gateway) {
                this.gateway = gateway;
        }

        @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
        ResponseEntity<?> hi(@PathVariable String name) {
                String message = "Hello, " + name + "!";
                this.gateway.directGreet("Direct: " + message);
                this.gateway.broadcastGreet("Broadcast: " + message);
                return ResponseEntity.ok(message);
        }
}
```

❶ the `@EnableBinding` annotation activates Spring Cloud Stream, as before

❷ 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

❸ 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.

❹ 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 3-15. 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";

        ❶
        @Input(DIRECTED)
        SubscribableChannel directed();

        @Input(BROADCASTS)
        SubscribableChannel broadcasts();

}
```

❶ 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 3-16. the `application.properties` for our consumer

❶
```
spring.cloud.stream.bindings.broadcasts.destination = greetings-pub-sub
```

❷
```
spring.cloud.stream.bindings.directed.destination = greetings-p2p
spring.cloud.stream.bindings.directed.group = greetings-p2p-group
spring.cloud.stream.bindings.directed.durableSubscription = true

server.port=0

spring.rabbitmq.addresses=localhost
```

❶ this should look fairly familiar given what we just covered in the producer

❷ 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 3-17. 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);
        }


❷

private IntegrationFlow incomingMessageFlow(SubscribableChannel incoming,

String prefix) {


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


// @formatter:off

return IntegrationFlows

.from(incoming)

.transform(String.class, String::toUpperCase)

.handle(String.class,

(greeting, headers) -> {

log.info("greeting received in IntegrationFlow ("

+ prefix + "): " + greeting);

return null;

}).get();

// @formatter:on

}

        @Bean
        IntegrationFlow direct(ConsumerChannels channels) {
                return incomingMessageFlow(channels.directed(), "directed");
        }

        @Bean
        IntegrationFlow broadcast(ConsumerChannels channels) {
```

```
            return incomingMessageFlow(channels.broadcasts(), "broadcast");
        }
}
```

❶ as before, we see `@EnableBinding` activates the consumer channels.

❷ 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 ineteroperate 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 3-18. 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.EnableDataFlowServer;

❶
@EnableDataFlowServer
@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 3-19. importing the RabbitMQ-bound modules.

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabbit-maven
```

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`.

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.

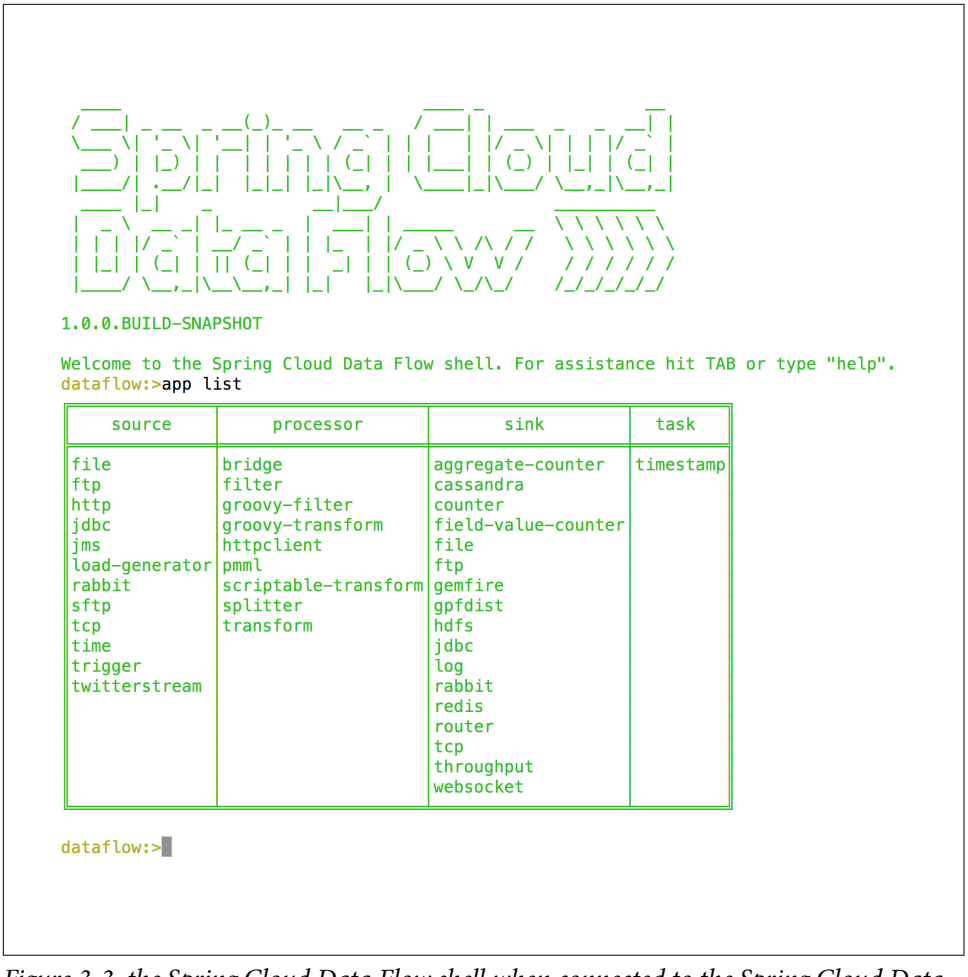


Figure 3-3. the Spring Cloud Data Flow shell when connected to the Spring Cloud Data Flow server with the default modules in place.

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 3-20.

❶

```
dataflow:>stream create --name ticktock --definition "time | log"
Created new stream 'ticktock'
```

❷

```
dataflow:>stream deploy --name ticktock
Deployed stream 'ticktock'
```

❶ this defines a stream that takes produced times and then pipes them to the logs

❷ 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 3-21. 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.servlet.Dispatch-
erServlet        : FrameworkServlet 'dispatcherServlet': initialization completed
in 11 ms
2016-06-07 01:37:51.244  INFO 24156 --- [nio-9393-exec-3] o.s.c.d.spi.local.Local-
AppDeployer       : deploying app ticktock.log instance 0
   Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-cloud-
dataflow-3828195950927318450/ticktock-1465277871225/ticktock.log
2016-06-07 01:37:51.259  INFO 24156 --- [nio-9393-exec-3] o.s.c.d.spi.local.Local-
AppDeployer       : deploying app ticktock.time instance 0
   Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-cloud-
dataflow-3828195950927318450/ticktock-1465277871251/ticktock.time
...
```

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 3-22. 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, outputChannel = Pro
cessor.OUTPUT)
        public Message<String> process(Message<String> in) {
                return MessageBuilder.withPayload("{" + in.getPayload() + "}") ❸
                                .copyHeadersIfAbsent(in.getHeaders()).build();
        }

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

❶ marks this class as a Spring Integration messaging endpoint

❷ activates the Spring Cloud Stream binding that connects our `input` and `output` channels to the broker

❸ 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 3-23. using the shell to register and deploy a stream

```
❶
dataflow:>app register --name brackets --type processor --uri maven://cnj:stream-
example:jar:1.0.0-SNAPSHOT
```

```
Successfully registered application 'processor:brackets'
```

❷

```
dataflow:>stream create --name bracketedticktock --definition "time | brackets |
log"
Created new stream 'bracketedticktock'
```

❸

```
dataflow:>stream list
╔═══════════════════╤═══════════════════════════╤═══════════╗
║   Stream Name     │  Stream Definition        │  Status   ║
╠═══════════════════╪═══════════════════════════╪═══════════╣
║bracketedticktock│time | brackets | log│undeployed║
╚═══════════════════╧═══════════════════════════╧═══════════╝
```

❹

```
dataflow:>stream deploy --name bracketedticktock
Deployed stream 'bracketedticktock'
```

❶ we must first register the application so that Spring Cloud Data Flow is aware of it

❷ create a stream that stitches the `time`, `log` and the newly-registered `bracket` component together.

❸ confirm that the stream's been registered

❹ 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 3-24.

❶

```
dataflow:>task create --name whattimeisit --definition timestamp
Created new task 'whattimeisit'
```

❷

```
dataflow:>task list
╔═══════════╤═══════════════╤═══════════╗
║ Task Name  │Task Definition│Task Status║
╠═══════════╪═══════════════╪═══════════╣
║whattimeisit│timestamp      │unknown    ║
```

```
╚═══════════╧═══════════════╧═══════════╝
```

❸

```
dataflow:>task launch --name whattimeisit
Launched task 'whattimeisit'
```

❹

```
dataflow:>task list
╔═══════════╤═══════════════╤═══════════╗
║ Task Name │Task Definition│Task Status║
╠═══════════╪═══════════════╪═══════════╣
║whattimeisit│timestamp      │complete   ║
╚═══════════╧═══════════════╧═══════════╝
```

❶ create a task that launches the very simple Spring Cloud Task called timestamp and prints out the value

❷ enumerate the existing tasks

❸ launch the task and give it a logical name by which to refer to it later

❹ 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 3-25. 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.EnableBatchProcess
ing;
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
❶
@EnableBatchProcessing
❷
@SpringBootApplication
public class BatchTaskExample {

        @Bean
        Job hello(JobBuilderFactory jobBuilderFactory,
                        StepBuilderFactory stepBuilderFactory) {

                // @formatter:off

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

                TaskletStep step = stepBuilderFactory
                                .get("one")
                                .tasklet((stepContribution, chunkContext) -> { ❸
                                                log.info("Hello, world");
                                                log.info("parameters: ");
                                                chunkContext
                                                                .getStepCon
text()
                                                                .getJobPara
meters()
                                                                .entrySet()
                                                                .forEach(

e -> log.info(e.getKey() + ':'

+ e.getValue()));
                                                return RepeatStatus.FIN
ISHED;
                                        }).build();

                return jobBuilderFactory.get("hello").start(step).build();

                // @formatter:on
        }

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

❶ activate Spring Cloud Task

❷ activate Spring Batch

❸ create a simple Spring Batch job that has one `Step` which uses a Spring Batch Tasklet instead of specifying a chunked `ItemReader` , `ItemProcessor` and `Item`

`Writer`, 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 3-26. using the shell to register and launch a Batch task

```
❶
dataflow:>app register --name args --type task --uri maven://cnj:task-example:jar:
1.0.0-SNAPSHOT
Successfully registered application 'task:args'

❷
dataflow:>task create --definition "args --p1=1 --p2=2" --name args
Created new task 'args'

❸
dataflow:>task launch --name args
Launched task 'args'

❹
dataflow:>task list
╔═══════════╤═══════════════╤═══════════╗
║Task Name  │Task Definition│Task Status║
╠═══════════╪═══════════════╪═══════════╣
║args       │args           │complete   ║
╚═══════════╧═══════════════╧═══════════╝
```

❶ we must first register the application so that Spring Cloud Data Flow is aware of it

❷ create a (potentially) parameterized instance of the application.

❸ launch an instance of the task

❹ 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 process-

ing of a Spring Batch `job` across a clsuter using Spring Cloud Stream as the messaging fabric.

CHAPTER 4

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 forklifiting* - 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  custom-app
```

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 ServletException,
IOException {

                response.setContentType("text/html");

                response.getWriter().print(
                                "<html><body><h3>Hello Cloud</h3></body></
html>");
        }
}
```

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-buildpack.git
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: {heap: 85,
stack: 10}}]'
```

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 ceate-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.HttpInvokerServiceExporter;

@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")
        ❷
        HttpInvokerServiceExporter httpMessageService() {
                HttpInvokerServiceExporter http = new HttpInvokerServiceEx
porter();
                http.setServiceInterface(MessageService.class);
                http.setService(this.messageService());
                return http;
        }
}
```

❶ the implementation itself needs to implement, at a minimum, the service interface specified in the `HttpInvokerServiceExporter`

❷ 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.AnnotationConfigApplicationCon
text;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean;

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

public class DemoApplicationTests {

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

        @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 = new Annota
tionConfigApplicationContext(
                                DemoApplicationClientConfiguration.class);

                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 HttpInvokerProx
yFactoryBean();
                        client.setServiceUrl("http://localhost:8080/messageSer
vice");
                        client.setServiceInterface(MessageService.class);
                        return client;
                }
        }

}
```

❶ the test first stands up the HTTP Invoker service

❷ ..then stands up the configuration for the client

❸ ..then interacts with the service through the shared interface

❹ 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;
        }

}
```

(1) 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 you're 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 Cloud Foundry
to mount the filesystem.

@SpringBootApplication
public class DemoApplication {

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

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.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

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 filessystem-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;

@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;

        ❶
        @RequestMapping(method = RequestMethod.POST)
        String createOrUpdate(@RequestParam MultipartFile file) throws Excep
tion {
                String name = file.getOriginalFilename();
                maybeLoadFile(name).ifPresent(
                                p -> gridFsTemplate.delete(getFilename
Query(name)));
                gridFsTemplate
                                .store(file.getInputStream(), name, file.getCon
tentType())
                                .save();
                return "redirect:/";
        }

        ❷
        @RequestMapping(method = RequestMethod.GET)
        @ResponseBody
```

```
        List<String> list() {
                return getFiles().stream().map(GridFSDBFile::getFilename)
                                .collect(Collectors.toList());
        }

        ❸
        @RequestMapping(value = "/{name:.+}", method = RequestMethod.GET)
        ResponseEntity<?> get(@PathVariable String name) throws Exception {
                Optional<GridFSDBFile> optionalCreated = maybeLoadFile(name);
                if (optionalCreated.isPresent()) {
                        GridFSDBFile created = optionalCreated.get();
                        try (ByteArrayOutputStream os = new ByteArrayOutput
Stream()) {
                                created.writeTo(os);

                                HttpHeaders headers = new HttpHeaders();
                                headers.add(HttpHeaders.CONTENT_TYPE, cre
ated.getContentType());
                                return new ResponseEntity<byte[]>(os.toByteAr
ray(), headers,
                                                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(getFilename
Query(name));
                return Optional.ofNullable(file);
        }

        private static Query getFilenameQuery(String name) {
                return Query.query(GridFsCriteria.whereFilename().is(name));
        }
}
```

❶ the `/files` endpoint accepts multipart file uploaded data and writes it to MongoDB's GridFS

❷ the `/files` endpoint simply returns a listing of the files in GridFS

❸ 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;

        ❶
        @RequestMapping("/email")
        SendGrid.Response email(@RequestParam String message) throws Exception {
```

```
                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 (message trunca
ted)");
                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.

❷ The auto-configuration expects a valid SendGrid username (`spring.sendgrid.username`) and password (`spring.sendgrid.password`). Remember that Spring Boot normalizes properties, and these values can be provided as environment variables: `SPRING_SENDGRID_USERNAME`, `SPRING_SENDGRID_PASSWORD`, etc.

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.