Bootiful Asciidoctor

A LOVE STORY

Table of Contents

About Mario Gray

Mario Gray (@mariogray) is a Developer Advocate at Tanzu, a divison of VMWare. `emacs` is *dead*! All hail `emacs`!

Action!

The book is done (for now) but there's still more to learn and do! In this section I'll provide a few things you want to bookmark or buy.

Websites to Bookmark

You should bookmark the following locations for your own reference and edication for later.

Additional reading

Want to learn more and take the next steps? Check out the following resources.

Next Steps

You've come this far. Take some time off, and then go build the next big thing.

Bootcamp

To appreciate reactive programming in the broader ecosystem's context, we'll need to learn about Spring. What is Spring? What is dependency injection? Why do we use Spring Boot over (or in addition to?) Spring Framework? This chapter is a primer on the basics. We don't talk about reactive programming in this chapter at all. If you already know the fundamentals of dependency injection, inversion-of-control, Java configuration, Spring application contexts, aspect-oriented programming (AOP), and Spring Boot auto-configuration, skip this chapter!

For the rest of us, we're going to demonstrate some key concepts by building something. We'll begin our journey, as usual, at our *second* favorite place on the internet, the Spring Initializr - Start dot Spring dot IO. Our goal is to build a trivial application. I specified `sib` in the `Group` field and `bootstrap` in the `Artifact` field, though please feel free to select whatever you'd like. Select the following dependencies using the combo box text field on the bottom right of the page where it says `Search for dependencies`: `DevTools`, `Web`, `H2`, `JDBC`, `Actuator` and `Lombok`.

`Devtools` lets us restart the application by running our IDE's build command eliminating the need to reload the entire JVM process. The result is quicker execution iterations and code-change visibility in the compiled code. It also starts up a Livereload-protocol server. Some excellent browser plugins out there can listen for messages on this server and force the browser page to refresh, giving you a seamless 'WSYSIWYG' type of experience.

> If you're using the Spring Tool Suite, you need only save your code, and you'll see changes here. IntelliJ has no built-in idea of a "save," and so no listener for this event. You'll need to instead "build" the code. Go to `Build` > `Build the project`. On a Mac, you could use Cmd + F9.

`Web` brings in everything you'd need today to build an application based on the Servlet specification and traditional Spring MVC. It brings in form validation, JSON and XML marshaling, WebSocket support, REST- and HTTP-controller support, etc. `H2` is an embedded SQL database that will lose its state on every restart. This is ideal for our first steps since we won't have to install a database (or reset it).

`JDBC` brings in support, like the `JdbcTemplate`, for working with SQL-based databases.

`Lombok` is a compile-time annotation processor that synthesizes things like getters, setters, `toString()` methods, `equals()` methods, and so much more with but a few annotations. Most of these annotations are self-explanatory in describing what they contribute to the class on which they're placed.

`Actuator` contributes HTTP endpoints, under `/actuator/\*`, that give visibility into the application's state.

There are some other choices if we're so inclined. What version of the JVM do you want to target? (I'd

recommend using the version corresponding to the latest OpenJDK build.) What language do you want? We'll explore Kotlin later, but let's use Java for now. Groovy is… well, *groovy*! It's a recommended choice, too. You can leave all the other values at their defaults for this simple application.

For all that we *did* specify, we *didn't* specify Spring itself. Or a logging library. Or any of a dozen sort of ceremonial framework-y bits that we'll need to do the job. Those are implied in the other dependencies when using Spring Boot, so we don't need to worry about them.

Now, I hear you cry, "what about reactive?" Good point! We'll get there, I promise, but nothing we're introducing in this section is reactive simply because we need to have a baseline. These days, it's fair game to assume you've some familiarity with things like JDBC (the Java Database Connectivity API) and the Servlet specification (there's traditionally been very little to be done on the web tier in Java that doesn't involve the Servlet specification).

Scroll to the bottom and select `Generate Project`.

[the Spring Initializr] | *bootstrap/the-spring-initializr.png*

Figure 1. This shows us the selections on the Spring Initializr for a new, reactive application.

This will generate and start downloading a new project in a `.zip` archive, in whatever folder your browser stores downloaded files. Unzip the archive, and you'll see the following layout:

The generated project structure.

```
Unresolved directive in chapters/bootstrap.adoc - include::includes/demo-tree.txt[the
tree of the Maven project generated from the Spring Initializr]
```

This is a stock-standard Maven project. The only thing that might not be familiar is the Maven wrapper - those files starting with `mvnw`. The Maven wrapper provides shell scripts, for different operating systems, that download the Apache Maven distribution required to run this project's build. This is particularly handy when you want to get the build to run as you'd expect it to run in a continuous integration environment. If you're on a UNIX-like environment (macOS, any Linux flavor), you will run `mvnw`. On Windows, you would run `mvnw.cmd`.

Your Maven build `pom.xml` should look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>rsb</groupId>
    <artifactId>bootstrap</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>bootstrap</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
        </project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
```

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

Our Maven build file, `pom.xml`, is pretty plain. Each checkbox selected on the Spring Initializr is represented as a dependency in the `pom.xml` build file. Most of them will have the `groupId` `org.springframework.boot`. We selected `Web`, and that corresponds to a dependency for building web applications with `artifactId` `spring-boot-starter-web`, for example. That explains where at least three of our dependencies came from. But that doesn't explain all of them. Testing is important, and we're in the future, and in the future, everybody tests. (Right!!...?) You'll see at least `spring-boot-starter-test` among the dependencies added to your Maven build. The Spring Initializr often adds other testing libraries where appropriate based on the libraries you've added. There's no "opt-in" for it. The Spring Initializr generates new projects with test dependencies automatically. (You're welcome!)

There's also an empty property file, `src/main/resources/application.properties`. Later, we'll see that we can specify properties to configure the application. Spring can read both `.properties` files and `.yaml` files.

This is a stock-standard Spring Boot application with a `public static void main(String[] args)` entry-point class, `BootstrapApplication.java`. It is an empty class with a `main` method and an annotation, and it is *glorious*! While I'd love to stay here in the land of little, to jump right into bombastic Spring Boot, this wouldn't be much of a *bootstrap* lesson without some background! So, delete `BootstrapApplication.java`. We'll get there. But first, some basics.

A Big ol' Bag o' Beans

Spring Framework, the first project to bear the Spring moniker, is at its core a dependency injection framework. Dependency injection frameworks are simple things with very profound possibilities. The idea is simple: applications change a lot. Decoupling, broadly, helps to reduce the cost of changing our application code and system architecture. So we write code in such a way that it is ignorant of where

dependencies - collaborating objects - originate.

The `CustomerService`

Suppose we've written an interface, `CustomerService`, for which we need to provide an implementation.

```
package sib.bootstrap;

import java.util.Collection;

public interface CustomerService {

    Collection<Customer> save(String... names);

    Customer findById(Long id);

    Collection<Customer> findAll();

}
```

The work of the `CustomerService` implementation itself isn't so interesting as how it's ultimately wired together. The wiring of the implementation - which objects are used to satisfy its dependencies - has an impact on how easy it is to change the implementation later on. This cost grows as you add more types to a system. The long tail of software project costs is in maintenance, so it is *always* cheaper to write maintainable code upfront.

The `JdbcTemplate`, in Spring's core JDBC support, is for many the first thing in the grab bag of utilities that led people to use Spring. It's been around for most of Spring's life. It supports common JDBC operations as expressive one-liners, alleviating most boilerplate (creating and destroying sessions or transactions, mapping results to objects, parameter binding, etc.) involved in working with JDBC.

To keep things simple and distracting discussions around object-relational mappers (ORMs) and the like - a paradigm well supported in Spring itself one way or another - we'll stick to the `JdbcTemplate` in our implementations. Let's look at a base implementation, `BaseCustomerService`, that requires a `DataSource` to do its work and instantiate a new `JdbcTemplate` instance.

```
package sib.bootstrap;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.util.Assert;

import javax.sql.DataSource;
```

```
import java.sql.PreparedStatement;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Objects;

①
public class BaseCustomerService implements CustomerService {

    private final RowMapper<Customer> rowMapper = (rs,
            i) -> new Customer(rs.getLong("id"), rs.getString("NAME"));

    ②
    private final JdbcTemplate jdbcTemplate;

    ③
    protected BaseCustomerService(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
    }

    @Override
    public Collection<Customer> save(String... names) {

        List<Customer> customerList = new ArrayList<>();
        for (String name : names) {
            KeyHolder keyHolder = new GeneratedKeyHolder();
            this.jdbcTemplate.update(connection -> {
                PreparedStatement ps = connection.prepareStatement(
                        "insert into CUSTOMERS (name) values(?)",
                        Statement.RETURN_GENERATED_KEYS);
                ps.setString(1, name);
                return ps;
            }, keyHolder);
            Long keyHolderKey = Objects.requireNonNull(keyHolder.getKey()).longValue();
            Customer customer = this.findById(keyHolderKey);
            Assert.notNull(name, "the name given must not be null");
            customerList.add(customer);
        }
        return customerList;
    }

    @Override
    public Customer findById(Long id) {
        String sql = "select * from CUSTOMERS where id = ?";
        return this.jdbcTemplate.queryForObject(sql, this.rowMapper, id);
    }
```

```
    @Override
    public Collection<Customer> findAll() {
        return this.jdbcTemplate.query("select * from CUSTOMERS", rowMapper);
    }

}
```

① This is a public class because we'll have several different implementations, in different packages, in this chapter. Normally you wouldn't have multiple implementations in different packages, and you should strive to assign a type the least visible modifier. A great majority of my code is package-private (no modifier at all).

② The `JdbcTemplate` reference to talk to our `DataSource`

③ For this implementation to do its work, it needs a `DataSource`

An Inflexible Implementation

The first cut at that implementation might depend on a `java.sql.DataSource` instance that allows it to talk to an RDBMS. It needs that object to be able to transact with the database, the store of record. This database is sure to change locations depending on the environment. It would therefore be a mistake to hard code credentials for the development database in the Java code. More broadly, it would be a mistake to bury the creation of the `DataSource` object in the `CustomerService` implementation itself. It's a terrible idea for a whole host of reasons, not least because of its bad security practice. It's also a bad idea because it couples the code to the data source running on the local machine. I can't switch out the URL to which my database driver should connect.

It's a seriously bad code smell if your development database, testing, and integration testing database, and production database are all the same database instance!

A naive implementation of `CustomerService.java` *- don't do this!*

```
package sib.bootstrap.hardcoded;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import sib.bootstrap.BaseCustomerService;
import sib.bootstrap.DataSourceUtils;

import javax.sql.DataSource;

class DevelopmentOnlyCustomerService extends BaseCustomerService {

    DevelopmentOnlyCustomerService() {
        super(buildDataSource());
    }

    private static DataSource buildDataSource() { ①
        DataSource dataSource = new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2).build();
        return DataSourceUtils.initializeDdl(dataSource);
    }

}
```

① This is brittle. It hardcodes the creation recipe for the `DataSource`, here using an embedded H2 in-memory SQL database, in the `CustomerService` implementation itself.

The biggest pity is that this implementation doesn't do anything except pass in the hardcoded `DataSource` to the base implementation's super constructor. The `BaseCustomerService` is parameterized. It preserves optionality. This subclass almost goes out of its way to remove that optionality by hardcoding this `DataSource`. What a waste. The `DataSource` *needs* to be created *somewhere*, but hopefully, we can agree it shouldn't be implemented. The `DataSource` represents a live connection to a network service whose location may change as we promote the application from one environment (development, QA, staging, etc.) to another. In this silly example, we're using an in-memory and embedded SQL database, but that's not going to the be the common case; you'll typically have a `DataSource` requiring environment-specific URIs, locators, credentials, etc.

The `DataSource` requires some initialization before any consumers can use it. This example collocates that creation and initialization logic in the `CustomerService` implementation itself. If you're curious, here's the initialization logic itself. We'll use this method - `DataSourceUtils#initializeDdl(DataSource)` - in subsequent examples.

```
package sib.bootstrap;

import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.init.DatabasePopulatorUtils;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;

import javax.sql.DataSource;

public abstract class DataSourceUtils {

    public static DataSource initializeDdl(DataSource dataSource) {
        ①
        ResourceDatabasePopulator populator = new ResourceDatabasePopulator(
                new ClassPathResource("/schema.sql")); ②
        DatabasePopulatorUtils.execute(populator, dataSource);
        return dataSource;
    }

}
```

① the `ResourceDatabasePopulator` comes from the Spring Framework. It supports executing SQL statements against a `DataSource` given one or more SQL files. It even has options around whether to fail the initialization if, for example, a database table already exists when trying to run a `CREATE TABLE` operation or whether to continue.

② Spring provides an abstraction, `Resource`, that represents some sort of resource with which we might want to perform input and output. The `ClassPathResource` represents resources found in the classpath of an application.

Here's an example demonstrating how we might wire up such an implementation.

```
package sib.bootstrap.hardcoded;

import sib.bootstrap.Demo;

public class Application {

    public static void main(String[] args) {
        DevelopmentOnlyCustomerService customerService = new
DevelopmentOnlyCustomerService();
        Demo.workWithCustomerService(Application.class, customerService);
    }

}
```

We exercise the resulting implementation with `Demo#workWithCustomerService(CustomerService)`. We'll

use this method in subsequent examples, so let's look at it.

```
package sib.bootstrap;

import lombok.extern.log4j.Log4j2;
import org.springframework.util.Assert;

import java.util.stream.Stream;

@Log4j2
public class Demo {

    public static void workWithCustomerService(Class<?> label,
            CustomerService customerService) {
        ①
        log.info("====================================");
        log.info(label.getName());
        log.info("====================================");

        ②
        Stream.of("A", "B", "C").map(customerService::save)
                .forEach(customer -> log.info("saved " + customer.toString()));

        ③
        customerService.findAll().forEach(customer -> {
            Long customerId = customer.getId();
            ④
            Customer byId = customerService.findById(customerId);
            log.info("found " + byId.toString());
            Assert.notNull(byId, "the resulting customer should not be null");
            Assert.isTrue(byId.equals(customer),
                    "we should be able to query for " + "this result");
        });
    }

}
```

① announce generally what we're doing

② write some data to the database using our implementation

③ find all records in the database

④ confirm that we can find each record by its ID

If that code looks suspiciously like a test… it is! Each example even has a JUnit unit test that exercises the same code path. We'll focus on how to stand up each example in the context of a `public static void main` application, and leave the tests. Suffice it to say, both the test and our demos exercise the

same code.

`DataSource` instances are expensive and typically shared across services. It makes little sense to recreate them everywhere they're used. Instead, let's centralize their creation recipe and write our code so that it doesn't care what reference it's been given.

A Parameterized Implementation

The simplest thing we can do to improve our example, and restore optionality, is to parameterize the `DataSource` through the constructor.

```
package sib.bootstrap.basicdi;

import sib.bootstrap.BaseCustomerService;

import javax.sql.DataSource;

class DataSourceCustomerService extends BaseCustomerService {

    ①
    DataSourceCustomerService(DataSource ds) {
        super(ds);
    }

}
```

① Not much to it! It's a class with a constructor that invokes the super constructor.

Here's the refactored `main` method.

```
package sib.bootstrap.basicdi;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import sib.bootstrap.CustomerService;
import sib.bootstrap.DataSourceUtils;
import sib.bootstrap.Demo;

import javax.sql.DataSource;

public class Application {

    public static void main(String[] args) {
        ①
        DataSource dataSource = new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2).build();
        ②
        DataSource initializedDataSource = DataSourceUtils.initializeDdl(dataSource);
        CustomerService cs = new DataSourceCustomerService(initializedDataSource);
        Demo.workWithCustomerService(Application.class, cs);
    }

}
```

① Our `CustomerService` depends only on a pointer to a `DataSource`. So, if we decide to change this reference tomorrow, we can!

② Much better! Our `CustomerService` only cares that it has a fully-formed `DataSource` reference. It doesn't need to know about initialization logic.

Much better. This implementation supports basic parameterization through generic base types. In this case, our code is ignorant of where the `DataSource` reference comes from. It could be a mock instance in a test or a production-grade connection pooled data source in a production environment.

One thing you'll note is that the code is a little naive with respect to transaction management in that it doesn't handle transactions at all. Our base implementation is, let's say, *optimisitic*! It's all written in such a way that we assume nothing could go wrong. To be fair, the `findById` and `findAll` methods are queries. So, either the query returns the results we've asked for, or it doesn't.

Templates

You're *probably* fine ignoring discussions of atomicity and transactions for those methods that read data since there's only one query. Things get sticky when you look at the `save(String ⋯ names)` method that loops through all of the input parameters and writes them to the database one by one. Sure, we probably should've used SQL batching, but this gives us the ability to have a thought experiment: what happens if there's an exception in processing midway through processing all the `String ⋯ names`

arguments? By that point, we'll have written one or more records to the database, but not all of them. Is that acceptable? In this case, it might be. Some are better than none, sometimes! It could be a more sophisticated example, though. You could be trying to write several related pieces of information to the database. Those related pieces of information would be inconsistent if they weren't all written to the database simultaneously; if their integrity wasn't maintained.

Some middleware, including SQL datastores, support a concept of a transaction. You can enclose multiple correlated things into a unit of work and then commit all those correlated things simultaneously. Either everything in the transaction is written, or everything is rolled back, and the results are as-if you hadn't written anything at all. It's much easier to reason about the system this way. You don't have to guess at what parts of the write succeeded and what didn't.

While we're looking at the concept of a transaction in the context of a SQL-based datastore and the `JdbcTemplate`, they're by no means unique to them. MongoDB supports transactions. So do many of your favorite message queues like RabbitMQ or those supporting the JMS specification. So does Neo4J. This basic workflow of working with a transaction is represented in Spring with the `PlatformTransactionManager`, of which *many* implementations support many different technologies. You can explicitly start some work, commit it or roll it back using a `PlatformTransactionManager`. This is simple enough, but it can be fairly tedious to write the try/catch handler that attempts the unit of work and commits it if there are no exceptions or rolls it back if there, even with a `PlatformTransactionManager`.

So, Spring provides the `TransactionTemplate`, which reduces this to a one-liner. You provide a callback that gets executed in the context of an open transaction. If you throw any exceptions, those result in a rollback. Otherwise, the transaction is committed. Let's revisit our example, this time incorporating transactions.

```
package sib.bootstrap.templates;

import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.BaseCustomerService;
import sib.bootstrap.Customer;

import javax.sql.DataSource;
import java.util.Collection;

public class TransactionTemplateCustomerService extends BaseCustomerService {

    private final TransactionTemplate transactionTemplate; ①

    public TransactionTemplateCustomerService(DataSource dataSource,
            TransactionTemplate tt) {
        super(dataSource);
        this.transactionTemplate = tt;
    }

    @Override
    public Collection<Customer> save(String... names) {
        return this.transactionTemplate.execute(s -> super.save(names));
    }

    @Override
    public Customer findById(Long id) {
        return this.transactionTemplate.execute(s -> super.findById(id));
    }

    @Override
    public Collection<Customer> findAll() {
        return this.transactionTemplate.execute(s -> super.findAll());
    }

}
```

① this class depends on having a `TransactionTemplate` in addition to a `DataSource`

Much better! We only even bother to catch the exception so that we can return a sane result, not because we need to clean up the database. It isn't so difficult to get this all working. Let's look at an application that wires the requisite objects together.

```
package sib.bootstrap.templates;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.DataSourceUtils;
import sib.bootstrap.Demo;

import javax.sql.DataSource;

public class Application {

    public static void main(String args[]) {

        DataSource dataSource = new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2).build();
        DataSource initializedDataSource = DataSourceUtils.initializeDdl(dataSource); ①
        PlatformTransactionManager dsTxManager = new DataSourceTransactionManager(
                initializedDataSource);②
        TransactionTemplate tt = new TransactionTemplate(dsTxManager); ③
        ④
        TransactionTemplateCustomerService customerService = new
TransactionTemplateCustomerService(
                initializedDataSource, tt);
        Demo.workWithCustomerService(Application.class, customerService);
    }

}
```

① initialize a `DataSource`, just as before…

② create an instance of `PlatformTransactionManager` called `DataSourceTransactionManager`.

③ wrap the `PlatformTransactionManager` in a `TransactionTemplate` object.

④ …and the rest is just as before.

Much better! We only even bother to catch the exception so that we can return a sane result, not because we need to clean up the database. The `TransactionTemplate` is just one of many `\*Template` objects - like the `JdbcTemplate` - that we've been using thus far that is designed to encapsulate boilerplate code like transaction management. A template method handles and hides otherwise boilerplate code and lets the user provide the little bit of variable behavior from one run to another. In this case, what we are doing with the database - queries, extraction, and transformation of results, etc. - is unique and so we need to provide that logic. Still, everything else related to using the `PlatformTransactionManager` implementation is not.

You will find that Spring provides a good many `\*Template` objects. The `JmsTemplate` makes working with JMS easier. The `AmqpTemplate` makes working with AMQP easier. The `MongoTemplate` and `ReactiveMongoTemplate` objects make working with MongoDB in a synchronous, blocking fashion and in an asynchronous, non-blocking fashion, respectively, easier. The `JdbcTemplate` makes working with JDBC easier. The `RedisTemplate` makes working with Redis easier. The `RestTemplate` makes creating HTTP client requests easier. There's another dozen or so you'll encounter in day-to-day work and a dozen more that are obscure but nice to have if you need them. One of my favorite, more obscure, examples is the `org.springframework.jca.cci.core.CciTemplate`, which makes working with the client-side of a Java Connector Architecture (JCA) connector through the Common Connector Interface (CCI) easier.

Will you ever need to use this? Statistically? No. Hopefully, never! It's an API that you'll need to integrate enterprise integration systems to your J2EE / Java EE application servers. We won't be anywhere near one of those in this book!

A Context For Your Application

This last example had more moving parts than any previous example. We had to create and configure four different objects to get to the point where we could do anything. Did I say "do"? I meant to exercise the API by passing the resulting, fully configured `CustomerService` as a method invocation parameter. We're still a long way from providing value to a business outcome, though. To do that, we'd need to connect actual clients to this functionality. We'd need to deploy this somewhere. We'd need to create more such services, too. And maybe change out our in-memory embedded SQL databases for a real one whose data endures across restarts. We're going to need to capture and componentize the recipes for these various objects. Right now, everything's buried in our `main(String[] args)` method. We have to copy and paste the code into our tests to confirm that things work as expected.

This isn't going to scale. Let's look at an example that uses Spring to describe the wiring of objects in your application. We'll see that it supports the flexibility we've worked so hard to obtain, thus simplifying our production code and our test code. This won't require a rewrite of the `CustomerService` - virtually everything is identical to before. It is only the wiring that changes.

Spring is ultimately a big bag of beans. It manages the beans and their lifecycles, but we need to tell it what objects are involved. One way to do that is to define objects (called "beans"). In this example, we'll define "beans" in our application class.

```
package sib.bootstrap.context;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
```

```
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.CustomerService;
import sib.bootstrap.DataSourceConfiguration;
import sib.bootstrap.Demo;
import sib.bootstrap.SpringUtils;
import sib.bootstrap.templates.TransactionTemplateCustomerService;

import javax.sql.DataSource;

①
@Configuration
@Import(DataSourceConfiguration.class) ②
public class Application {

    ③
    @Bean
    PlatformTransactionManager transactionManager(DataSource ds) {
        return new DataSourceTransactionManager(ds);
    }

    @Bean
    TransactionTemplateCustomerService customerService(DataSource ds,
            TransactionTemplate tt) {
        return new TransactionTemplateCustomerService(ds, tt);
    }

    @Bean
    TransactionTemplate transactionTemplate(PlatformTransactionManager tm) {
        return new TransactionTemplate(tm);
    }

    public static void main(String args[]) {
        ④
        ApplicationContext ac = SpringUtils.run(Application.class, "prod");

        ⑤
        CustomerService cs = ac.getBean(CustomerService.class);
        Demo.workWithCustomerService(Application.class, cs);
    }

}
```

① the `Application` class is *also* a `@Configuration` class. It is a class that has methods annotated with `@Bean` whose return values are objects to be stored and made available for other objects in the application context

② The definition of our `DataSource` changes depending on whether we're running the application in a development context or production. We have stored those definitions in another configuration class, which we import here. We'll review those definitions momentarily.

③ Each method in a `@Configuration`-annotated class annotated with `@Bean` is a *bean provider method*.

④ I've hidden the complexity of creating a Spring `ApplicationContext` in this method, `SpringUtils.run`. There are a half dozen interesting implementations of the `ApplicationContext` interface. Usually, we won't need to care which and when we use what because the creation of that object in Spring Boot, which we'll use later, is hidden from us. To arrive at a working instance of the Spring `ApplicationContext`, we need to furnish both the configuration class and a *profile*, a label, `prod`. We'll come back to labels momentarily.

⑤ The `ApplicationContext` is the heart of a Spring application. It's the thing that stores all our configured objects. We can ask it for references to any bean, by the beans class type (as shown here) or its bean name.

Those `@Bean` provider methods are important. This is how we define objects and their relationships for Spring. Spring starts up, invokes the methods, and stores the resulting objects to be made available for other objects that need those references as collaborating objects. When Spring provides a reference to the dependency, we say it has "injected" the dependency. If any other code anywhere in the application needs an object of the type (or types, if interfaces are expressed on the resulting type) returned from the method, they'll be given a reference to the single instance returned from this method the first time it was invoked.

If a bean needs a reference to another to do its work, it expresses that *dependency* as a parameter in the bean provider method. Spring will look up any beans of the appropriate definition, and it will provide them as parameters when it invokes our method.

We can recreate the entire application by recreating the `ApplicationContext`.. In this first example, we're using plain ol' Spring Framework. Nothing special about it. Let's look at how we create our `ApplicationContext` instance, but keep in mind that we'll not need this boilerplate code later.

```
package sib.bootstrap;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.util.StringUtils;

public abstract class SpringUtils {

    public static ConfigurableApplicationContext run(Class<?> sources, String profile) {
        ①
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext();

        ②
        if (StringUtils.hasText(profile)) {
            applicationContext.getEnvironment().setActiveProfiles(profile);
        }

        ③
        applicationContext.register(sources);
        applicationContext.refresh();

        ④
        applicationContext.start();
        return applicationContext;
    }

}
```

① we're using an `ApplicationContext` instance that can handle annotation-centric configuration, also known as "Java configuration".

② You can tell Spring to create, or not create, objects based on various conditions. One condition is, "does this bean have a profile associated with it?" A profile is a label, or a tag, attached to an object's definition. We haven't seen one yet, but we will. By setting an active profile, we're saying: "create all objects that have no profile associated with them, *and* create all objects associated with the specific profiles that we make active."

③ In this case, we're registering a configuration class. In other contexts, the "sources" might be other kinds of input artifacts.

④ finally, we launch Spring, and it, in turn, goes about triggering the creation of all the objects contained therein.

Let's look at how the `DataSource` definitions are handled in a separate class, `DataSourceConfiguration`. I've extracted these definitions out into a separate class to more readily reuse their definition in subsequent examples. I want to centralize all the complexity of constructing the `DataSource` into a single place in the codebase. We're going to take advantage of profiles to create two `DataSource`

definitions. One that results in an in-memory H2 `DataSource` and another configuration that produces a `DataSource` when given a driver class name, a username, a password, and a JDBC URL. These parameters are variable and may change from one developer's machine to another and one environment to another.

Spring has an `Environment` object that you can inject anywhere you want it, that acts like a dictionary for configuration values - just keys and values associated with those keys. Values may originate anywhere - property files, YAML files, environment variables, databases, etc. You can teach the `Environment` about new sources for configuration values by contributing an object of type `PropertySource` to the `Environment`. Spring has a convenient annotation, `@PropertySource`, that takes any configuration values from a file and adds them to the `Environment`. Once in the `Environment`, you can have those values injected into configuration parameters in your bean provider methods with the `@Value` annotation.

```
package sib.bootstrap;

import org.h2.Driver;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.context.annotation.PropertySource;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
public class DataSourceConfiguration {

    ①
    @Configuration
    @Profile("prod") ②
    @PropertySource("application-prod.properties") ③
    public static class ProductionConfiguration {

        @Bean
        DataSource productionDataSource(@Value("${spring.datasource.url}") String url, ④
                @Value("${spring.datasource.username}") String username,
                @Value("${spring.datasource.password}") String password,
                @Value("${spring.datasource.driver-class-name}") Class<Driver>
driverClass ⑤
        ) {
```

```
            DriverManagerDataSource dataSource = new DriverManagerDataSource(url,
                    username, password);
            dataSource.setDriverClassName(driverClass.getName());
            return dataSource;
        }

    }

    @Configuration
    @Profile("default") ⑥
    @PropertySource("application-default.properties")
    public static class DevelopmentConfiguration {

        @Bean
        DataSource developmentDataSource() {
            return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2).build(
);
        }

    }

    @Bean
    DataSourcePostProcessor dataSourcePostProcessor() {
        return new DataSourcePostProcessor();
    }

    ⑦
    private static class DataSourcePostProcessor implements BeanPostProcessor {

        @Override
        public Object postProcessAfterInitialization(Object bean, String beanName)
                throws BeansException {

            if (bean instanceof DataSource) {
                DataSourceUtils.initializeDdl(DataSource.class.cast(bean));
            }
            return bean;
        }

    }

}
```

① `@Configuration` classes can act as a container for other configuration classes. When we import the `DataSourceConfiguration` class, Spring also resolves any nested configuration classes.

② This configuration class is only meant to be active when the `prod` profile is active.

③

Tell Spring that we want configuration values from the `application-prod.properties` property file, located at `src/main/resources/application-prod.properties`, to be loaded into the `Environment`.

④ inject values from the configuration file by their key using the `@Value` annotation

⑤ Spring can convert string values in property files to more complex types, like `Class<T>` literals, because it delegates to another subsystem in Spring called the `ConversionService`. You can customize this object, too!

⑥ the `default` profile is a special profile. It is active *only* when no other profile is active. So, if you specifically activate the `prod` profile, then `default` won't be active. If you don't activate any profile, then `default` will kick in. Thus, all objects in this `default` profile will be contributed by default. Here, we contribute an in-memory H2 embedded SQL database.

⑦ In previous examples, we used our initializer, `DataSourceUtils.initializeDdl`, to ensure that the `DataSource` had the required DDL run against it before it was put into service. Now, we have two places where we define a `DataSource`. We could simply duplicate that logic across both locations, but that violates the DRY (Don't Repeat Yourself) principle. Instead, we will configure an object of type `BeanPostProcessor`. `BeanPostProcessor` is a callback interface. It provides two (default) methods. Here, our class overrides `postProcessAfterInitialization` which is invoked for every object in the Spring application. We interrogate the type of the object, confirm it is a `DataSource` of some sort, and then initialize it. This way, it doesn't matter how or where the `DataSource` is created. The `BeanPostProcessor` ensures it is properly initialized. Spring has other lifecycle events and associated callback interfaces that might be interesting that you'll see from time to time. `BeanPostProcessor` is one of the more common.

In the `Application` class, we explicitly pass in the profile, `prod`. This is not the only way to configure the profile, though. It's a limiting approach, too. Consider that the profile, so specified, is hardcoded into the application logic. In a normal workflow, you would promote the application binary from the environment to another without recompiling to not risk inviting variables into the builds. So, you'd want some way to change the profile without recompilation. Spring supports a command-line argument, `--spring.profiles.active=prod`, when running `java` to start the application. You could also specify the property in your main class's `main` method - say `System.setProperty("spring.profiles.active", "prod")` right before the `SpringApplication.run` call. The `prod` profile consults properties in its property file, `application-prod.properties`.

```
spring.datasource.url=jdbc:h2:mem:rsb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
```

Naturally, you could change any of these property values. If you were using PostgreSQL or MySQL or Oracle, or whatever, it'd be a simple matter to update these values accordingly.

Component Scanning

The configuration class approach has the advantage of being explicit; we need only to inspect the class to see every object in the application and its wiring. It's also another artifact we need to evolve as we move forward in time, as we add new objects to the application. Each time we add a new service, we need to configure it in this configuration class. Spring supports a ton of different types of objects - components are the simplest of the hierarchy. It also supports controllers, services, repositories, and any of several other types of objects. For each object you add to a Spring application, you need to have a corresponding entry in the configuration class. Is that, in its way, a violation of DRY, as well?

Spring can *implicitly* learn the wiring of objects in the application graph if we let it perform component scanning when the application starts up. If we added the `@ComponentScan` annotation to our application, Spring would discover any objects in the current package, or beneath it, that had identifying marker - or "stereotype" - annotations. This would complement the use of Java configuration nicely. In this scenario, Spring's component scanning would discover all Spring objects that we, the developer define, things like our services and HTTP controllers. At the same time, we'd leave things like the `DataSource` and the `TransactionTemplate` to the Java configuration. Put another way; if you have access to the source code and can annotate it with Spring annotations, then you might consider letting Spring discover that object through component scanning.

When Spring finds such an annotated object, it'll inspect the constructor(s). If it finds no constructor, it'll instantiate an instance of the application using the default constructor. If it finds a single constructor with no arguments, it'll instantiate that. Suppose it finds a constructor argument whose values may be satisfied by other objects in the Spring application (the same way that they might for our bean definition provider methods). In that case, Spring will provide those collaborating objects. If it finds multiple, ambiguous constructors, you can tell Spring which constructor to use by annotating that constructor with the `@Autowired` annotation to disambiguate it from other constructors.

Let's rework our application, ever so slightly, in the light of component scanning.

```
package sib.bootstrap.scan;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.CustomerService;
import sib.bootstrap.DataSourceConfiguration;
import sib.bootstrap.Demo;
import sib.bootstrap.SpringUtils;

import javax.sql.DataSource;

@Configuration
@ComponentScan ①
@Import(DataSourceConfiguration.class)
public class Application {

    @Bean
    PlatformTransactionManager transactionManager(DataSource ds) {
        return new DataSourceTransactionManager(ds);
    }

    @Bean
    TransactionTemplate transactionTemplate(PlatformTransactionManager tm) {
        return new TransactionTemplate(tm);
    }

    public static void main(String args[]) {
        ConfigurableApplicationContext applicationContext = SpringUtils
                .run(Application.class, "prod");
        CustomerService customerService = applicationContext
                .getBean(CustomerService.class);
        Demo.workWithCustomerService(Application.class, customerService);
    }

}
```

① The only thing of note here is that we've enabled component scanning using the `@ComponentScan` annotation and that we don't have a bean provider method for the `CustomerService` type since Spring will now automatically detect that type in the component scan.

It will discover the `CustomerService` type in the component scan, *if*, that is, we annotate it to be

discovered. Let's create a new type that does nothing but host a stereotype annotation, `@Service`.

```
package sib.bootstrap.scan;

import org.springframework.stereotype.Service;
import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.templates.TransactionTemplateCustomerService;

import javax.sql.DataSource;

①
@Service
class DiscoveredCustomerService extends TransactionTemplateCustomerService {

    ②
    DiscoveredCustomerService(DataSource dataSource, TransactionTemplate tt) {
        super(dataSource, tt);
    }

}
```

① the `@Service` annotation is a stereotype annotation. This class exists in the same package as `Application.java`, only to allow us to annotate it so that it can be discovered. Most codebases will have much shallower hierarchies. The stereotype annotation will live on the class with the business logic's implementation, not just a subclass of some other thing extracted into another package just for improvement as in this book!

② the `Application` class defines instances of these types, so we know that Spring can satisfy these dependencies.

Declarative Container Services with `@Enable*` Annotations

We introduced Spring and added a class to support our configuration. We extricated the objects' wiring from the `main(String[])` method and into this new configuration class. Our code has more moving parts, not less. It's not hard to see Spring could make things simpler - particularly when using component scanning - as we add even more objects to our application. The configuration, amortized over the cost of more objects in the application, is almost nothing at all. That's a win in the near term but doesn't apply to us right now. What can we do to simplify the code right now? Our service now uses the `TransactionTemplate` to manage transactions. We use the `TransactionTemplate` to demarcate transactional boundaries for our core functionality. For every method we add, we need to wrap the transaction demarcation logic in the same way. This cross-cutting concern - transaction demarcation - is one thing with which we'll have to often contend, even though its a fairly straightforward requirement. It shouldn't require constant code to support it, either. Let's look at how we can simplify the code with declarative transaction demarcation.

```
package sib.bootstrap.enable;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.support.TransactionTemplate;
import sib.bootstrap.CustomerService;
import sib.bootstrap.DataSourceConfiguration;
import sib.bootstrap.Demo;
import sib.bootstrap.SpringUtils;

import javax.sql.DataSource;

@Configuration
@EnableTransactionManagement ①
@ComponentScan
@Import(DataSourceConfiguration.class)
public class Application {

    @Bean
    PlatformTransactionManager transactionManager(DataSource ds) {
        return new DataSourceTransactionManager(ds);
    }

    @Bean
    TransactionTemplate transactionTemplate(PlatformTransactionManager tm) {
        return new TransactionTemplate(tm);
    }

    public static void main(String args[]) {
        ConfigurableApplicationContext applicationContext = SpringUtils
                .run(Application.class, "prod");
        CustomerService customerService = applicationContext
                .getBean(CustomerService.class);
        Demo.workWithCustomerService(Application.class, customerService);
    }

}
```

① the *only* difference now is that we're enabling declarative transaction management.

This is otherwise the same as any other example, except that we have the one extra annotation. Now,

we can reimplement the `CustomerService` implementation. Well, we don't have to reimplement it. Simply declare it and annotate it with `@Transactional`. All `public` methods in the implementation class then have transactions demarcated automatically.

```
package sib.bootstrap.enable;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import sib.bootstrap.BaseCustomerService;
import sib.bootstrap.Customer;

import javax.sql.DataSource;
import java.util.Collection;

@Service
@Transactional ①
public class TransactionalCustomerService extends BaseCustomerService {

    public TransactionalCustomerService(DataSource dataSource) {
        super(dataSource);
    }

    @Override
    public Collection<Customer> save(String... names) {
        return super.save(names);
    }

    @Override
    public Customer findById(Long id) {
        return super.findById(id);
    }

    @Override
    public Collection<Customer> findAll() {
        return super.findAll();
    }

}
```

① this is the only thing that we have to do - decorate the Spring bean with `@Transactional`.

That's it! The `@Transactional` annotation has attributes that we can use that give us some of the flexibility that we have in explicitly managing the transactions with the `TransactionTemplate`. Not all, but most. As written, we get default transaction demarcation for all public methods. We could explicitly configure transaction demarcation per-method by annotating each method with the `@Transactional` annotation and overriding the class-level configuration there.

A "Bootiful" Application

So far, we've built a working application that talks to a database, manages declarative transactions, and we've streamlined the code as much as possible. The code is svelter because Spring can handle a lot of boilerplate code for you. So far, though, we've focused *only* on creating a service that talks to a database. We're nowhere near ready for production! There's a ton of things to sort out before we have a working REST API to which clients might connect. And we have considerably more still to do before we have a client of any sort - HTML 5, Android, iOS, whatever - to connect to our REST API. We've come a long way! And yet… we're still nowhere. We need to stand up a web server, configure a web framework, set up security, etc.

Spring has us covered here. Indeed, Spring has us covered… anywhere! We could use Spring MVC for Servlet-based web applications. We could use Spring Data and its numerous modules supporting data access across SQL and NoSQL datastores. We could use Spring Security to integrate authentication and authorization in our application. We could use Spring Integration to build out messaging-centric integration flows, talking to technologies like Apache Kafka, RabbitMQ, FTP, IMAP, JMS, etc. We could use Spring Batch to support batch processing for large, sequential data access jobs. And microservices? Yah, there is a *lot* to absorb for that, too!

We also need to care about observability - something needs to articulate the application's health so that we can run it with confidence in production and so that it can be effectively monitored. Monitoring is critical; monitoring gives us the ability to measure and, with those measurements, advance. Let's see if we can kick things up a notch.

The Rise of the Convention over Configuration Web Frameworks

We need to be more productive; there's (a lot) more to do, and it would be nice if we could exert less energy to do it. This isn't a new problem. There have been many efforts in the Java community, and in other communities, to support more productive application development.

Ruby on Rails, a web framework that debuted in 2004 and became white-hot popular in the early 2000s, is owed a large credit here. It was the first project to support what it called "convention over configuration," wherein the framework was optimized for common sense scenarios that it made trivial. In those days, there was a lot of discussion of web applications "babysitting databases." What else do you need? An HTML-based frontend that talked behind the scenes to a SQL database described 80% of the applications at the time. Ruby on Rails optimized for this particular outcome, but it was *really* optimized! The Rails team had the famous 5-minute demo. In the space of five actual minutes, they initialized a new application and integrated data access and a user interface that supported manipulating that data. It was a *really* quick way to build user interfaces that talked to SQL databases. Ruby on Rails was driven by code generation and buoyed by the Ruby language's dynamic nature. It coupled the representation of database state to the HTML forms and views with which users would interact. It made heavy use of code generation to get there; users interact with the command-line shell to generate new entities mapped to state in the SQL database; they used the shell to generate "scaffolding" with the views and entities. The approach is a considerable improvement over the then prolific technologies of the day.

Ruby on Rails critics would say that it was very difficult to unwind the underlying assumptions. As most of a Ruby on Rails application is generated code and highly opinionated runtime code, it is very difficult to unwind the choices made by the framework without having to rewrite entire verticals of the framework. Either the code it generated for you worked the way you wanted it to, or you'd have to scrap everything. The use cases average web developers face evolved, too. If you wanted to have a user interface that, in turn, manipulated two different database entities, then things got difficult. If you wanted to build an HTTP-based REST API, then you were out of luck. If you wanted to integrate a NoSQL datastore, then you were on your own. All of these things were ultimately added to Ruby on Rails; it's evolved, certainly. But the criticisms linger. Now, closer to 2020 than 2000, Ruby on Rails is optimized for the wrong thing. Most applications are not web applications babysitting a database anymore. They're client-service based architectures. The clients run in different logical tiers and different physical tiers, in Android runtimes, on iOS, and in rich and capable HTML 5 browsers like Google Chrome and Mozilla Firefox.

The Spring team has had an interesting history here, too. There were two efforts of note with which the Spring team was involved. The first, Spring Roo, is a code-generation approach to Java development. The premise behind Spring Roo is that there were a ton of moving parts in a working Java application circa 2008 that weren't Java code. XML deployment descriptors. `.JSP`-based views. Hibernate mapping configuration XML files. Just a ton of things! Spring Roo took a very Ruby on Rails-centric approach to code generation, with the same fatal flaws. Assumptions were too difficult to unwind. It was optimized for one type of application.

Grails leaned more heavily on runtime configuration. It had code generation, but most of its dynamic nature came from the Groovy. Both Spring Roo and Grails built on Spring. Spring Roo generated a whole bunch of Spring-based code that could be changed if needed, but the effort could be an uphill battle. Grails instead supported metaprogramming and hooks to override runtime assumptions. Grails is far and away from the most successful of the two options. I'd even argue it was the most successful of all the convention-over-configuration web frameworks after Ruby on Rails itself!

What's missing? Grails is a Groovy-centric approach to building applications. If you didn't want to use the Groovy programming language (why wouldn't you? It's *fantastic*!), then Grails is probably not the solution for you. Grails was for most of its life optimized for the web applications babysitting database use case. And, finally, while Java and regular Spring could never hope to support the kind of metaprogramming that is uniquely possible in the Groovy language, they were both pretty dynamic and could offer more.

Cloud Native Applications and Services

The shape of modern software has changed with architecture changes. No longer do we build web applications babysitting databases. Instead, clients talk to services. *Lots* of services. Small, singly focused, independently deployable, autonomously updatable, reusable, bounded contexts. *Microservices*. Microservices are the software manifestation of a new paradigm, called *continuous delivery*, where organizations optimize for continuous feedback loops by embracing software that can be easily, continuously, and incrementally updated. The goal is to shorten feedback loops to learn from iterations in production. Kenny Bastani and I look at this paradigm in O'Reilly's epic tome *Cloud Native*

Java. One knock-on effect of this architectural paradigm is that software is constantly being moved from development to production; that change is constant. The things that you only worry about when you're close to production become things you need to worry about when you're just starting. In a continuous delivery pipeline, you could see software pushed to production due to every single `git push`! How will you handle load-balancing? Security? Monitoring and observability? DNS? HTTPS certificates? Failover? Racked-and-stacked servers? VMs? Container security?

Our frameworks need to be optimized for production. They must simplify as much of this undifferentiated heavy lifting as possible.

Spring Boot

In 2013, we released Spring Boot to the world. Spring Boot is a different approach to building Java applications. It offers strong opinions, loosely held. Spring Boot integrates the best-of-breed components from the Spring and larger Java ecosystems into one cohesive whole. It provides default configurations for any of some scenarios but, and this part is key; it provides a built-in mechanism to undo or override those default configurations. There is *no* code generation required. A working Spring application is primarily Java code and a build artifact (a `build.gradle` or a `pom.xml`). Whatever dynamic behavior is needed can be provided at runtime using Java and metaprogramming.

Let's revisit our example, this time with Spring Boot. Spring Boot is just Spring. It's Spring configuration for all the stuff that you *could* write, but that you don't need to. I like to say that Spring Boot is your chance to pair-program with the Spring team.

First, let's look a the `Application` class.

```
package sib.bootstrap.bootiful;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;
import sib.bootstrap.CustomerService;
import sib.bootstrap.Demo;

①
@SpringBootApplication
public class Application {

    public static void main(String args[]) {
        ②
        System.setProperty("spring.profiles.active", "prod");
        ③
        SpringApplication.run(Application.class, args);
    }

}

④
@Profile("dev")
@Component
class DemoListener {

    private final CustomerService customerService;

    DemoListener(CustomerService customerService) {
        this.customerService = customerService;
    }

    ⑤
    @EventListener(ApplicationReadyEvent.class)
    public void exercise() {
        Demo.workWithCustomerService(getClass(), this.customerService);
    }

}
```

① `@SpringBootApplication` is an annotation that itself is meta-annotated with a few other annotations including `@Configuration`, `@ComponentScan` and `@EnableAutoConfiguration`. The first two should be familiar, but we'll come back to the last one in a moment.

② This application has code that runs under different profiles. Spring can take a hint from the environment as to which profile should be active in several different ways, including environment variables or Java `System` properties. I'd *normally* stick with just using the environment variables.

③ `SpringApplication.run(…)` is standard Spring Boot. This is part of every single application. It comes provided with the framework and does everything that our trivial `SpringUtils.run` method did (and considerably more).

④ In previous examples, constructing the application in production was different from the recipe for testing it. So we had to duplicate code. Here, Spring Boot behaves the same in both a test and in the production code, so we keep the call to `Demo.workWithCustomerService(CustomerService)` in a bean that is only active if the `dev` Spring profile is active.

⑤ Spring is a big bag of beans. Components can talk to each other through the use of `ApplicationEvent` instances. In this case, our bean listens for the `ApplicationReadyEvent` that tells us when the application is just about ready to start processing requests. This event gets called as late as possible in the startup sequence as possible.

The event listener mechanism is nice because it means we no longer have to muddy our `main(String [] args)` method; it is the same from one application to another.

The `@EnableAutoConfiguration` annotation, though not seen, is arguably the most important part of the code we just looked at. It activates Spring Boot's auto-configuration. Auto-configuration classes are run by the framework at startup time and given a chance to contribute objects to the resulting object graph. Specifically, when Spring Boot starts up it inspects the text file, `META-INF/spring.factories`, in all `.jar` artifacts on the CLASSPATH. Spring Boot itself provides one, but your code could as well. The `spring.factories` text file enumerates keys and values associated with those keys.

Therein, Spring Boot inspects a line that begins with `org.springframework.boot.autoconfigure.EnableAutoConfiguration`. This line has as its value a *yuuuuge* list of fully-qualified class names. These class names are Java configuration classes. They're classes that, in turn, contribute objects to the Spring object graph. Objects that do the things we'd much rather not do. Like, configure a web server. Or a `DataSource`. Or a `PlatformTransactionManager` and `TransactionTemplate`. Did you notice that we didn't import the `sib.bootstrap.DataSourceConfiguration` class as we did in previous examples? That we didn't define anything, really, except the bean required to exercise our `CustomerService` implementation.

```
package sib.bootstrap.bootiful;

import org.springframework.stereotype.Service;
import sib.bootstrap.enable.TransactionalCustomerService;

import javax.sql.DataSource;

①
@Service
class BootifulCustomerService extends TransactionalCustomerService {

    BootifulCustomerService(DataSource dataSource) {
        super(dataSource);
    }

}
```

① This implementation simply extends the existing `@Transactional`-annotated example. We could've defined that bean using a `@Bean` provider method, too, I suppose.

Thus far, the properties we've been using - from `application.properties` - are automatically read in when the application starts up. We've been using keys that start with `spring.datasource` all this time because those are the keys that Spring Boot is expecting. It'll even load the right property file based on which Spring profile is active!

This auto-configuration is helpful already, but we can do a whole lot more. Let's build a REST API.

```
package sib.bootstrap.bootiful;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import sib.bootstrap.Customer;
import sib.bootstrap.CustomerService;

import java.util.Collection;

①
@RestController
public class BootifulRestController {

    private final CustomerService customerService;

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

    ②
    @GetMapping("/customers")
    Collection<Customer> get() {
        return this.customerService.findAll();
    }

}
```

① This is another stereotype annotation, quite like `@Component` and `@Service`. It is itself meta-annotated with `@Component`. It tells Spring that this class is also a `@Component` but that it is specialized; it exposes handlers that it expects Spring MVC, the web framework in play here, to map to incoming HTTP requests

② Spring MVC knows which HTTP requests to match to which handler based on the handler methods' mapping annotations. Here, this handler method is mapped to HTTP GET requests for the URL `/customers`.

Run the application and inspect the logs. It will have announced `Tomcat started on port(s): ⋯`. Note the port. I'm just going to assume it's `8080`. You should replace it with the noted port. Then visit `http://localhost:8080/customers` in your browser to see the resulting JSON output! You could also request that resource using `curl`. Spring Boot not only configured a working web framework for us, it also configured a webserver! Not just any webserver. It auto-configured Apache Tomcat, the world's leading Java web server, that powers the very large majority of all Java web applications.

Spring Boot will run the application, by default, on port 8080. However, we can easily customize that and so many other things about Spring Boot applications' running behavior by specifying the relevant properties in a `application.properties` or `application.yml` file. Let's look at the configuration file.

```
①
spring.jmx.enabled = false
②
server.port=0
③
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

① Do we want Spring Boot to export information about the application over the JMX protocol?

② On what port do we want the web application run? `server.port=0` tells Spring Boot to run the application on any unused port. This is doubly convenient

③ These last two properties tell Spring Boot to expose all the Actuator endpoints and expose the Actuator Health endpoint's details.

What is the Actuator? Glad you asked! Our application is intended for production, and in production, no one can hear your application scream… unless it is visible from an HTTP JSON endpoint that your monitoring systems can inspect. The Spring Boot Actuator library, which is on your CLASSPATH, auto-configures a standard set of HTTP endpoints that articulate the application's state to support observability and operations. Want to know what the application's health is? Visit `http://localhost:8080/actuator/health`. Want to see metrics? Visit `http://localhost:8080/actuator/metrics`. There are many other endpoints, all of which are accessible at the root endpoint, `http://localhost:8080/actuator`.

"Hold on a tick!" I hear you exclaim. "Where did these Actuator endpoints, and the web server that serves them up, come from?" Recall that, way back at the beginning of this journey, we visited the Spring Initializr, where we selected some dependencies, including `Web` and `Actuator`. This resulted in build artifacts `spring-boot-starter-web` and `spring-boot-starter-actuator` being added to the resulting project's build file. Those artifacts, in turn, contained code that our Spring Boot auto-configuration picks up and configures. It detects the classpath classes and, because those classes are on the classpath, it configures the objects required.

Pretty *bootiful*, right?

But What If…

You know, I wasn't sure how long this chapter was going to take or if it was even worth doing. I halfway thought that I could take for granted that many people already knew Spring and Spring Boot basics. I wasn't going to do it. I wasn't going to do it until one day; a student approached me with a fairly trivial question in one of my classes. She had used Spring Boot for a couple of years. She'd come to the Spring Boot ecosystem from the Node.js ecosystem and had no other Java background. She had joined a team, and they'd been successful in building a monolithic application with Spring Boot because the happy path cases were friction-free and worked as advertised. Her team had built a pretty sophisticated application and took it to production, entirely by adhering to the happy-paths. She was stuck, however, and wondered if I could help. We spent ten minutes answering the wrong questions

when it dawned on me she had no familiarity with Spring Framework itself! She'd never had to use it before Spring Boot. Here was this very competent engineer (she had successfully built React-based applications! She was more sophisticated than I am…) who was stuck because she lacked a foundational understanding of the technologies behind Spring Boot! That is our fault. We on the Spring Boot team are always trying to do better here. I hope that this chapter is a straight line from basics to "bootiful" and that you have an idea of, roughly, what's supposed to happen in a Spring Boot application. Hopefully, this clears up a few things. Hopefully, you've got a clear picture of what to do next.

Sometimes you won't, though! Sometimes things break, and it's hard to be sure why. It can be daunting to debug an application if you don't know where to start, so let's talk about some first-line-of-defense tricks for understanding Spring Boot applications.

- **Use the Debug Switch**: Spring Boot will log all the auto-configuration classes that were evaluated, and in what way, if you flip the debug switch. The easiest way might be to set an environment variable, `export DEBUG=true`, before running the application. You can also run the Java program using `--debug=true`. Your IDE, like IntelliJ IDEA Ultimate edition or Spring Tool Suite, might have a checkbox you can use when running the application to switch the debug flag on
- **Use the Actuator**: the Actuator, as configured in our example, will have many useful endpoints that will aid you. `/actuator/beans` will show you all the objects and how they are wired together. `/actuator/configprops` will show you the properties that you could put in the `application.(yml|properties)` file that can be used to configure the running application. `/actuator/conditions` shows you the same information as you saw printed to the console when the application started. `/actuator/threaddump` and `/actuator/heapdump` will give you thread dumps and heap dumps. Very useful if you're trying to profile or debug race conditions.
- **@Enable Annotations Usually Import Configuration Classes**: You can command or ctrl-click on a `@Enable-` annotation in your IDE, and you'll be shown the source code for the annotation. You'll very commonly see that the annotation has `@Import(⋯)` and brings in a configuration class that might explain how a given thing is coming to life.

These are some first-cut approaches to problems you might encounter, but they're by no means your only recourse. If you're still stuck, you can depend on the largest community in the Java ecosystem to help you out! We're always happy to help. There is, of course, the chatrooms - `spring-projects` [http://Gitter.im/spring-projects] and `spring-cloud`[http://gitter.im/Spring-Cloud] - where you need only your Github ID to post questions and to chat with the folks behind the projects. The Spring team frequent those chat rooms, so drop on in and say hi! Also, we monitor several different tags on Stackoverflow so be sure to try that, too!

Deployment

We've got an application up and running, and we've even got Actuator in there. At this point, it's time to figure out how to deploy it. The first thing to keep in mind is that Spring Boot is deployed as a so-called "fat" `.jar` artifact. Examine the `pom.xml` file, and you'll find that the `spring-boot-maven-plugin` has been configured. When you go to the root of the project and run `mvn clean package`, the plugin will

attempt to bundle up your application code and all the relevant dependencies into a single artifact you can run using `java -jar ⋯`. In our particular case, it will fail because it won't be able to unambiguously resolve the single class with the `main(String ⋯ args)` method to run. To get it to run, configure the Maven plugin's `mainClass` configuration element, pointing it to the last `Application` instance we created.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <mainClass>sib.bootstrap.bootiful.Application</mainClass>
    </configuration>
</plugin>
```

Now, when you run `mvn clean package`, you'll get a working `.jar` in the `target` directory. You can do any of some things with it from there. You might deploy it to a cloud platform, like Cloud Foundry.

Deploying the application to a Cloud Foundry foundation

```
cf push -p target/my-jar.jar my-new-app
```

This will upload the artifact to the platform, which will assign it a `PORT` and a load-balanced URI to announce on the shell.

You could also containerize the application and deploy that to Cloud Foundry, or a Kubernetes distribution like PKS.

Next Steps

In this chapter, we've examined the logical evolution of an application from basic to "bootiful." We've hopefully established that Spring Boot is a force multiplier. Hopefully, you're clear on how it could be used to build an application that's destined for production. Hopefully, you're clear on the basic workflow. You start at the Spring Initializr, you add the relevant "starter" dependencies to activate desired functionality, and then configure the application, if necessary, by changing properties and contributing objects. From here, the world is your oyster! There are dozens of Pivotal-sponsored projects supporting Spring and far more besides maintained by third parties. Want to work with Microsoft Azure? Use their Spring Boot support. Want to work with Alibaba's Cloud? Use Spring Cloud Alibaba. Want to work with Google Cloud Platform? Check out Spring Cloud GCP. Want to work on Amazon Web Services? Check out Spring Cloud AWS! The list of interesting third party APIs goes on and on.

Thus far, we've focused entirely on synchronous and blocking, input and output because I presume that this is a model with which you're already familiar, and we could focus in this chapter on Spring itself. The general workflow of building reactive, non-blocking, asynchronous Spring Boot applications

is the same as introduced in this chapter. We've just left those specifics for later.

So, let's get to it!

Acknowledgements

I want to thank myself.

Some Code

You know what would be nice right about now? An include. Yah. An include. I'm all about inclusivity, inclusion, and code includes.

```
Unresolved directive in chapters/acknowledgements.adoc - include::/home/runner/Desktop
/root/code/bootcamp/README.md[]
```

Not bad, eh?