covers Spring 5.0

Spring IN ACTION

FIFTH EDITION

Craig Walls

MANNING

MEAP Edition
Manning Early Access Program

Spring in Action

Fifth Edition

Covers Spring 5.0

Version 4

welcome

Thank you for purchasing the MEAP for *Spring in Action, Fifth Edition*.

Can you believe that? The **fifth** edition of Spring in Action! Spring has come a long way over the course of the last four editions and I'm so excited to have a chance to bring the latest and greatest stuff that Spring has to offer in this all new edition. This book should be a valuable resource regardless of whether you're completely new to Spring or are reading this book to brush up on the newest features.

I've attempted to make this edition follow a hands-on, narrative style; leading you through a journey of building an application, starting with initializing the project and going all the way through to how to ready the application for deployment.

We're releasing the first two chapters to start. In chapter 1 you'll learn how to kick start your Spring project leveraging the Spring Initializr and Spring Boot. And before chapter 1 concludes, you'll already have a complete, albeit basic application that is ready to run.

In chapter 2, we'll build on that foundation by using Spring MVC to develop additional browser-based functionality. We'll see how to handle simple web requests, process form submissions, and validate what the user enters.

Throughout the remaining chapters in part 1, we'll see how to use Spring to persist and retrieve data, secure our web application, and then examine how to take advantage of configuration properties to fine-tune how our application components behave.

Looking even further ahead, part 2 of the book will see us integrating our application with other applications. In part 3 we'll dig into Spring 5's new support for reactive programming and revisit some previously developed components to make them more reactive. In part 4 we'll adapt the application to be cloud-native by decomposing it into microservices, leveraging Spring Cloud to bring them together. Finally, in part 5, we'll see how to prepare our application for deployment and see how Spring applications are deployed in a variety of runtime settings.

We hope to have frequent updates to the book, every few weeks, whether that is new chapters or updates to existing chapters. As you are reading, I invite you to visit the Author Online forum to ask questions and leave comments. Your feedback is truly appreciated and I find it valuable in guiding me as I write it.

—Craig Walls

brief contents

PART 1 : FOUNDATIONAL SPRING

1 *Booting Spring*

2 *Developing web applications*

3 *Working with data*

4 *Securing Spring*

5 *Working with configuration properties*

PART 2: INTEGRATED SPRING

6 *Giving Spring some REST*

7 *Consuming REST services*

8 *Messaging*

9 *Working with data in real-time and batch*

PART 3: REACTIVE SPRING

10 *Introducing Reactor*

11 *Working with Reactive REST APIs*

12 *Persisting data reactively*

13 *Securing Reactive Spring*

PART 4: CLOUD-NATIVE SPRING

14 *Discovering services*

15 *Centralizing configuration*

16 *Failing gracefully*

PART 5: DEPLOYED SPRING

17 *Monitoring and instrumenting for production*

18 *Deploying Spring applications*

APPENDIXES

A *Bootstrapping and running Spring projects*

B *Wiring Beans*

1 Booting Spring

This chapter covers:

- Spring and Spring Boot essentials
- Initializing a Spring project
- An overview of the Spring landscape

Although the Greek philosopher Heraclitus wasn't well known as a software developer, he seemed to have a good handle on the subject. He has been quoted to have said "The only constant is change". That statement captures a foundational truth of software development. The way we develop applications today is different than it was a year ago, 5 years ago, 10 years ago, and certainly 15 years ago when a very initial form of the Spring Framework was introduced in Rod Johnson's book, *Expert One-on-One J2EE Design and Development* [1] .

Back then, the most common type of applications being developed were browser-based web applications, backed by relational databases. While that type of development is still relevant, and Spring is well-equipped for those kinds of applications, we're now also interested in developing applications composed of microservices destined for the cloud that persist data in a variety of databases. And a new interest in reactive programming aims to provide greater scalability and improved performance with non-blocking operations.

As software development evolved, the Spring Framework evolved with it to address modern development concerns, including microservices and reactive programming. Spring has also set out to simplify its own development model with the introduction of

[1] www.wiley.com/WileyCDA/WileyTitle/productCd-0764543857.html

Spring Boot.

Whether you're developing a simple database-backed web application or constructing a modern application built around microservices, Spring is the framework that will help you achieve your goals. This chapter is your first step in a journey through modern application development with Spring.

1.1 What is Spring?

I know that you're probably itching to start writing a Spring application, and I assure you that before this chapter ends, we'll have developed a simple Spring application. But first, let me set the stage with a few basic Spring concepts that will help you understand what makes Spring tick.

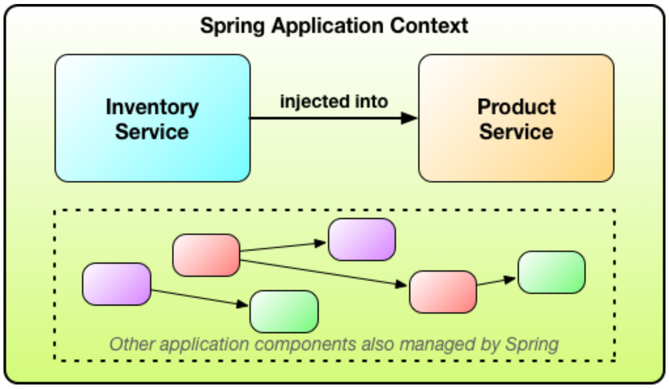
Any non-trivial application is composed of many components, each responsible for their piece of the overall application functionality and coordinating with other components to get their job done. When the application is run, those components somehow need to be created and introduced to each other.

At it's very core, Spring offers a container, often referred to as the Spring application context, that creates and manages application components. These components, or *beans*, are wired together inside of the Spring application context to make a complete application, much like bricks, mortar, timber, nails, plumbing, and wiring are bound together to make a house.

The act of wiring beans together is based on a pattern known as *dependency injection*. Rather than have components create and maintain the lifecycle of other components that they depend upon, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those components into the components that need them, typically through constructor arguments or property accessor methods.

For example, suppose that among an application's many components, there are two which are an inventory service (for fetching inventory levels) and a product service (which provides basic product information). The product service depends on the inventory service to be able to provide a complete set of information about products. Figure 1.1 illustrates the relationships between these beans and the Spring application context.

Figure 1.1. Application components are managed and injected into each other by the Spring application context.


Spring Application Context
Inventory Service
injected into
Product Service
Other application components also managed by Spring

On top of its core container, Spring and a full portfolio of related libraries offer a web framework, a variety of data persistence options, a security framework, integration with other systems, runtime monitoring, microservice support, a reactive programming model, and many other features that are necessary for modern application development.

Historically, the way you would guide Spring's application context to wire beans together was with one or more XML files that described the components and their relationship to other components. For example, the following XML declares two beans, an `InventoryService` bean and a `ProductService` bean and wires the `InventoryService` bean into the `ProductService` via a constructor argument:

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
  <constructor-arg ref="inventoryService" />
</bean>
```

In more recent versions of Spring, however, a Java-based configuration is more common. The following configuration class is equivalent to the XML configuration:

```
@Configuration
public class ServiceConfiguration {
  @Bean
  public InventoryService inventoryService() {
    return new InventoryService();
  }

  @Bean
  public ProductService productService() {
    return new ProductService(inventoryService());
  }
}
```

The `@Configuration` annotation indicates to Spring that this class is a configuration class which will provide beans to the Spring application context. The configuration class' methods are annotated with `@Bean` indicating that the objects that they return should be added as beans in the application context (where, by default, their respective bean IDs will be the same as the names of the methods that define them).

Java-based configuration offers several benefits over XML-based configuration, including greater type-safety and improved refactorability. Even so, explicit configuration with either Java or XML is only necessary if Spring is unable to automatically configure the components.

Automatic configuration has its roots in a Spring technique known as *auto-wiring* and another technique known as *component-scanning*. With component-scanning, Spring can automatically discover components from the application's classpath and create them as beans in the Spring application context. And using auto-wiring, Spring can automatically inject them with other beans that they depend upon.

More recently, with the introduction of Spring Boot, automatic configuration has gone well beyond component-scanning and auto-wiring. Spring Boot is an extension of the Spring Framework that offers several productivity enhancements to Spring. The most well-known of these enhancements is auto-configuration, where Spring Boot can make reasonable guesses of what components need to be configured and wired together based on entries in the classpath, environment variables, and other factors.

I'd like to show you some example code that demonstrates auto-configuration. But I can't. You see, auto-configuration is much like the wind. You can see the effects of it, but there's no code that I can show you and say "Look! Here's an example of auto-configuration!" Stuff happens, components are enabled, and functionality is provided without writing code. It's this lack of code that is essential to auto-configuration and what makes it so wonderful.

Spring Boot auto-configuration has dramatically reduced the amount of explicit configuration (whether with XML or Java) required to build an application. In fact, by the time we finish the example in this chapter, you'll have a working Spring application that has only a single line of Spring configuration!

Spring Boot offers so much benefit to Spring development that it's hard to imagine developing Spring applications without it. For that reason, this book will treat Spring and Spring Boot as if they were one-in-the-same. We'll leverage Spring Boot as much as possible and use explicit configuration only when it's necessary. And, because Spring XML configuration is the old-school way of working with Spring, we'll focus primarily on Spring's Java-based configuration.

But enough of this chit-chat, yick-yack, and flim-flam. This book's title includes the phrase "in action", so let's get moving and start writing our first application with Spring.

1.2 Initializing a Spring application

Through the course of this book, we're going to create Taco Cloud, an online application for ordering the most wonderful food created by man: tacos. Of course, we're going to use Spring, Spring Boot, and a variety of related libraries and frameworks to achieve this goal.

There are several options for initializing a Spring application. While I could walk you through the steps of manually creating a project directory structure and defining a build specification, that's wasted time—time better spent actually writing application code. Therefore, we're going to lean on the Spring Initializr to bootstrap our application.

The Spring Initializr is both a browser-based web application as well as a REST API that can produce a skeleton Spring project structure that you can flesh out with whatever functionality you want. There are several ways to use Spring Initializr, including the following:

- The web application at start.spring.io
- From the command line using the `curl` command
- From the command line using the Spring Boot Command Line Interface
- When creating a new project in Spring Tool Suite
- When creating a new project in IntelliJ IDEA
- When creating a new project in Netbeans

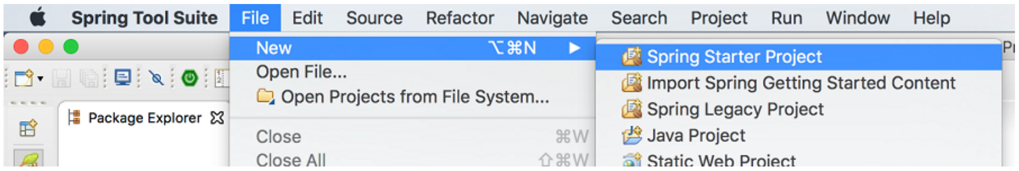
Rather than spend several pages of this chapter talking about each one of these options, I've collected those details in Appendix A. In this chapter, and throughout this book, I'm going to show how to create a new project using my favorite option, the Spring Initializr support in Spring Tool Suite. As its name suggests, Spring Tool Suite is a fantastic Spring development environment. But it also offers a handy Spring Boot Dashboard feature that (at least at the time I write this) isn't available in any of the other IDE options.

If you're not a Spring Tool Suite user, that's fine and we can still be friends. Just hop over to Appendix A and substitute the following instructions with those for the Initializr option that suits you best. Just know that throughout this book, I may occasionally reference features specific to Spring Tool Suite, such as the Spring Boot Dashboard. If you're not using Spring Tool Suite then you'll need to adapt those instructions to fit your IDE.

1.2.1 Initializing a Spring project in Spring Tool Suite

To get started with a new Spring project in Spring Tool Suite, go to the "File" menu and select "New" and then "Spring Starter Project". Figure 1.2 shows the menu structure to look for.

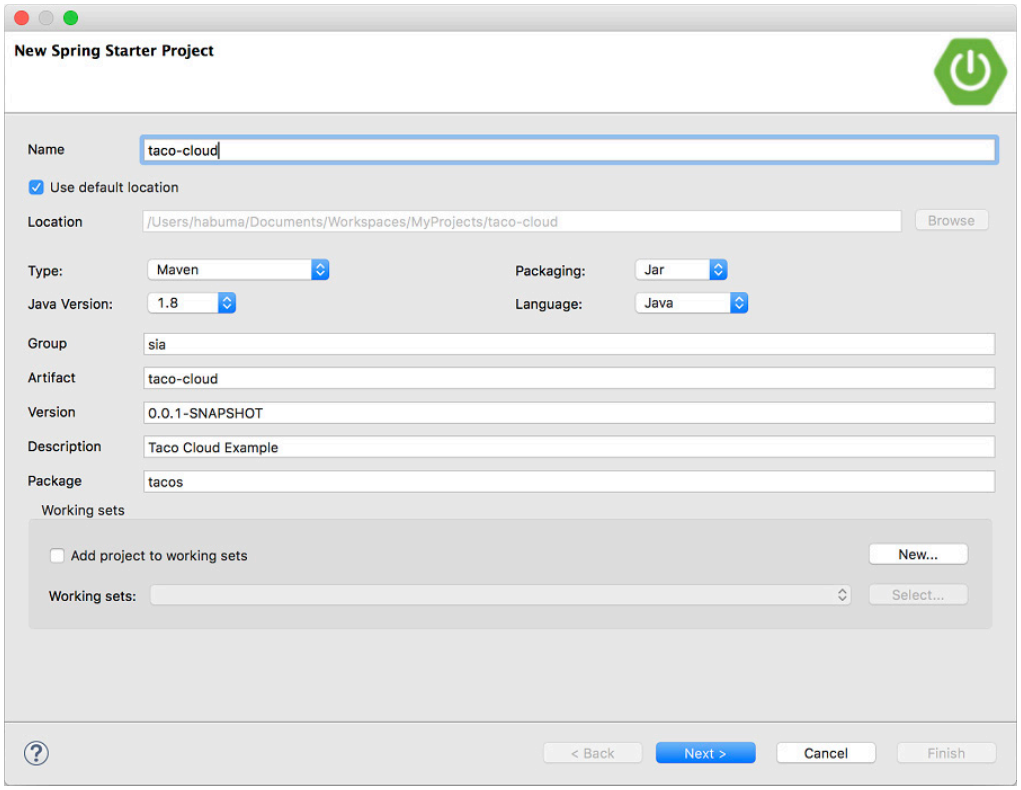
Figure 1.2. Starting a new project with the Intiializr in Spring Tool Suite.



Once you select "Spring Starter Project" a new project wizard dialog Figure 1.3 will appear. The first page in the wizard asks you for some general project information, such as the project name, description, and some other essential information. If you're familiar with the contents of a Maven `pom.xml` file, you'll recognize most of the fields as items that end up in a Maven build specification.

For the Taco Cloud application, fill in the dialog as shown in Figure 1.3 , then click "Next >".

Figure 1.3. Specifying general project information.



New Spring Starter Project

Name	taco-cloud
☑ Use default location	
Location	/Users/habuma/Documents/Workspaces/MyProjects/taco-cloud
Type:	Maven
Packaging:	Jar
Java Version:	1.8
Language:	Java
Group	sia
Artifact	taco-cloud
Version	0.0.1-SNAPSHOT
Description	Taco Cloud Example
Package	tacos

Working sets

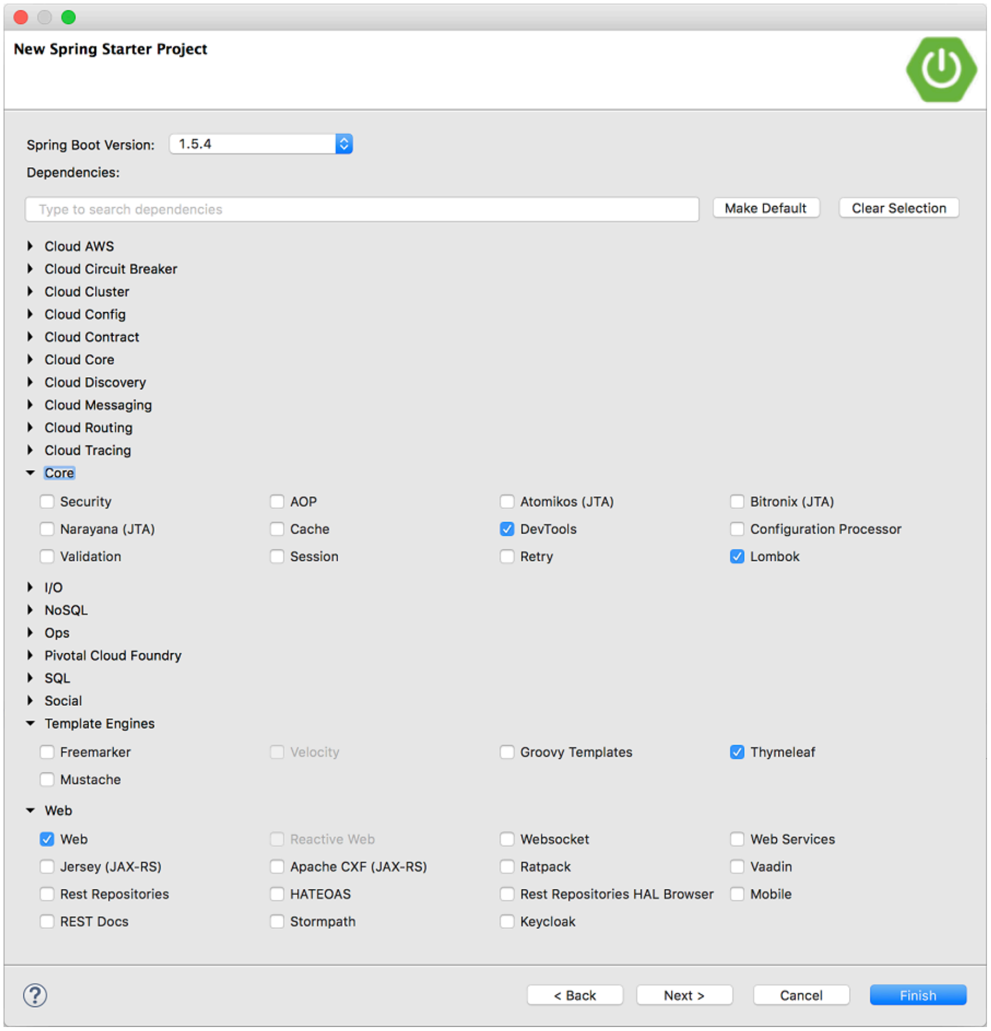
☐ Add project to working sets

Working sets:

The next page in the wizard offers you the chance to select dependencies to add to your project (see Figure 1.4). Near the top of the dialog, you'll notice that you have a chance to select which version of Spring Boot you want to base your project on. This defaults to the most current version available and it's generally a good idea to leave it as-is, unless you need to target a different version.

As for the dependencies themselves, you can either expand the various sections and seek out the desired dependencies manually, or search for them in the search box near the top. For the Taco Cloud application, we're going to start with the dependencies shown in Figure 1.4

Figure 1.4. Choosing starter dependencies.



New Spring Starter Project

Spring Boot Version: 1.5.4

Dependencies:

Type to search dependencies

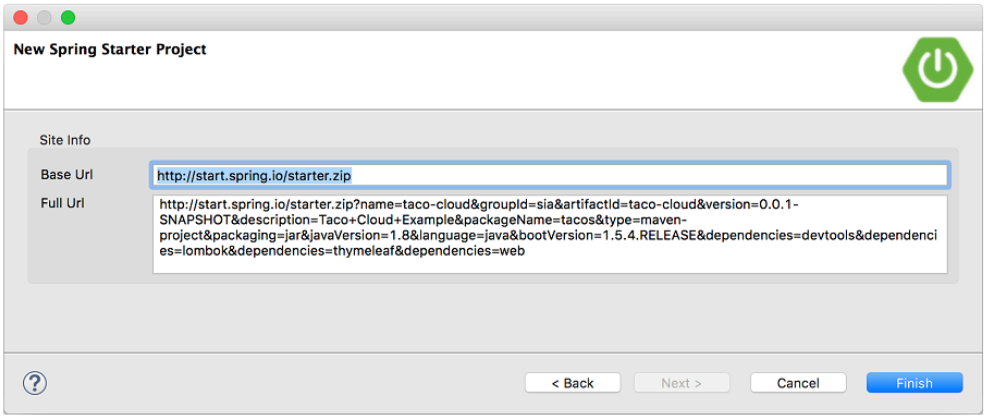
Make Default | Clear Selection

- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
- Cloud Contract
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
  - [ ] Security
  - [ ] AOP
  - [ ] Atomikos (JTA)
  - [ ] Bitronix (JTA)
  - [ ] Narayana (JTA)
  - [ ] Cache
  - [✓] DevTools
  - [ ] Configuration Processor
  - [ ] Validation
  - [ ] Session
  - [ ] Retry
  - [✓] Lombok
- I/O
- NoSQL
- Ops
- Pivotal Cloud Foundry
- SQL
- Social
- Template Engines
  - [ ] Freemarker
  - [ ] Velocity
  - [ ] Groovy Templates
  - [✓] Thymeleaf
  - [ ] Mustache
- Web
  - [✓] Web
  - [ ] Reactive Web
  - [ ] Websocket
  - [ ] Web Services
  - [ ] Jersey (JAX-RS)
  - [ ] Apache CXF (JAX-RS)
  - [ ] Ratpack
  - [ ] Vaadin
  - [ ] Rest Repositories
  - [ ] HATEOAS
  - [ ] Rest Repositories HAL Browser
  - [ ] Mobile
  - [ ] REST Docs
  - [ ] Stormpath
  - [ ] Keycloak

< Back | Next > | Cancel | Finish

At this point you can just click "Finish" to generate the project and add it to your workspace. But if you're feeling slightly adventurous, click "Next >" one more time.

Figure 1.5. Optionally specifying an alternate Initializr address.



Base Url: http://start.spring.io/starter.zip

Full Url: http://start.spring.io/starter.zip?name=taco-cloud&groupId=sia&artifactId=taco-cloud&version=0.0.1-SNAPSHOT&description=Taco+Cloud+Example&packageName=tacos&type=maven-project&packaging=jar&javaVersion=1.8&language=java&bootVersion=1.5.4.RELEASE&dependencies=devtools&dependencies=lombok&dependencies=thymeleaf&dependencies=web

By default, this new project wizard makes a call out to the Spring Initializr at start.spring.io to generate the project.

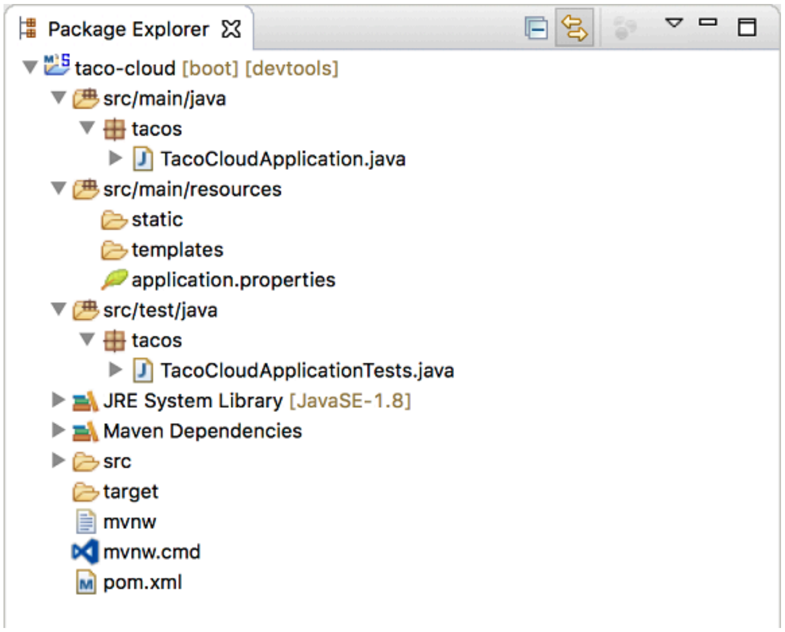
Generally, there's no need to override this default, which is why you could have clicked "Finish" on the 2nd page of the wizard. But if for some reason you're hosting your own clone of Intializr (perhaps a local copy on your own machine or a customized clone running inside your company firewall), then you'll want to change the "Base Url" field to point at your Initializr instance before clicking "Finish".

After you click "Finish", the project will be downloaded from the Initializr and loaded into your workspace. Wait a few moments to give it a chance to load and build and then you'll be ready to start developing the application functionality. But first, let's take a look at what the Initializr gave us.

1.2.2 Examining the Spring project structure

After the project has been loaded in the IDE, expand it to see what it contains. Figure 1.6 shows the expanded Taco Cloud project in Spring Tool Suite.

Figure 1.6. The initial Spring project structure as shown in Spring Tool Suite.



```
taco-cloud [boot] [devtools]
  src/main/java
    tacos
      TacoCloudApplication.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    tacos
      TacoCloudApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

You may recognize this as a typical Maven or Gradle project structure, where application source code is placed under `src/main/java`, test code is placed under `src/test/java`, and non-Java resources are placed under `src/main/resources`. Within that project structure, you'll want to take note of the following items:

- `mvnw` and `mvnw.cmd` : These are Maven wrapper scripts. You can use these to build your project even if you don't have Maven installed on your machine.
- `pom.xml` : This is the Maven build specification. We'll look deeper into this in a moment.
- `TacoCloudApplication.java` : This is the Spring Boot main class that bootstraps the project. We'll take a closer look at this class in a moment.
- `application.properties` : This file is initially empty, but offers a place where you can specify configuration properties. We'll tinker with this file a little in this chapter, but will defer a detailed explanation of configuration properties to chapter 5.
- `static` : This folder is where you can place any static content (images, stylesheets, Javascript, etc) that you want to be able to serve to the browser. It is initially empty.
- `templates` : This folder is where you'll place template files that will be used to render content to the browser. It's initially empty, but we'll add a Thymeleaf template soon.
- `TacoCloudApplicationTests.java` : This is a very simple test class that ensures that the Spring application context will load successfully. We'll certainly add

more tests to the mix as we develop the application.

As the Taco Cloud application grows, we'll most certainly fill in this barebones project structure with Java code, images, stylesheets, tests, and other collateral that will make our project more complete. But in the meantime, let's dig a little deeper into a few of the items that Spring Initializr provided to us.

EXPLORING THE BUILD SPECIFICATION

When we filled out the form at start.spring.io, we specified that our project should be built with Maven. Therefore, the Spring Initializr gave us a `pom.xml` file, already populated with the choices we made. Listing 1.1 shows the entire `pom.xml` file provided by the Initializr.

Listing 1.1. The initial Maven build specification.

```
<?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>sia</groupId>
  <artifactId>taco-cloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>                        ❶

  <name>taco-cloud</name>
  <description>Taco Cloud Example</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.M3</version>                     ❷
    <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>                                             ❸
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
    </dependency>

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

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

  <repositories>
    <repository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>

</project>
```

❶ JAR packaging
❷ Spring Boot version
❸ Starter dependencies
❹ Spring Boot plugin

The first noteworthy item in the `pom.xml` file is the `<packaging>` element. We chose to build our application as an executable JAR file, as opposed to a WAR file. This is probably one of the most curious choices we'll make, especially for a web application. After all, traditional Java web applications are packaged as WAR files, leaving JAR files the packaging of choice for libraries (and the occasional desktop user-interface application).

The choice of JAR packaging is a cloud-minded choice. Whereas WAR files are perfectly suitable for deploying to a traditional Java application server, they are an unnatural fit for most cloud platforms. While some cloud platforms (such as CloudFoundry) are capable of deploying and running WAR files, all Java cloud platforms are capable of running an executable JAR file. Therefore, the Spring Initializr will default to JAR packaging unless you tell it to do otherwise.

If you'll need to deploy your application to a traditional Java application server, then you'll need to choose WAR packaging and include a web initializer class. We'll look at how to build WAR files in more detail in chapter 2.

Next, take note of the `<parent>` element and, more specifically, its `<version>` child. This specifies that our project has `spring-boot-starter-parent` as its parent POM. Among other things, this parent POM will provide dependency management for several dependency libraries commonly used in Spring projects. For those libraries covered by the parent POM, we will not have to specify a version, as it will be inherited from the parent. The version, `2.0.0.RELEASE` indicates that we're using Spring Boot 2.0.0 and thus will inherit dependency management as defined by that version of Spring Boot.

While we're on the subject of dependencies, you'll see that there are three dependencies declared under the `<dependencies>` element. The first two should look somewhat familiar to you. They correspond directly to the "web" and "thymeleaf" dependencies that we selected before clicking the "Finish" button in the Spring Tool Suite new project wizard. The third dependency is one that provides a lot of helpful testing capabilities. We didn't have to check a box for it to be included because the Spring Initializr will assume (hopefully correctly) that you will be writing tests.

You may also notice that all three dependencies have the word "starter" in their artifact ID. Spring Boot starter dependencies are kind of special in that they typically don't have any library code themselves, but instead will transitively pull in other libraries. These starter dependencies offer three primary benefits:

- Your build file will be significantly smaller and easier to manage because you won't need to declare a dependency on every library you might need.
- You are able to think of your dependencies in terms of what capabilities they provide, rather than in terms of library names. If you're developing a web application, you add the web starter dependency, rather than a laundry list of individual libraries that enable you to write a web application.
- You are freed from the burden of worry about library versions. You can trust that for a given version of Spring Boot, the versions of the libraries brought in transitively will be compatible. You only need to worry about which version of Spring Boot you're using.

Finally, the build specification ends with the Spring Boot plugin. This plugin performs a few important functions:

- It provides a Maven goal that will enable you to run the application using Maven. We'll try this goal out in section "Building and running the application".
- It ensures that all dependency libraries are included within the executable JAR file and available on the runtime classpath.
- It produces a manifest file in the JAR file that causes the bootstrap class (`TacoCloudApplication` in our case) is the main class for the executable JAR.

Speaking of the bootstrap class, let's open it up and take a closer look.

BOOTSTRAPPING THE APPLICATION

Since we'll be running the application from an executable JAR, it's important to have a main class that will be executed when that JAR file is run. We'll also need at least a minimal amount of Spring configuration to bootstrap the application. That's what you'll find in the `TacoCloudApplication` class, shown in Listing 1.2 .

Listing 1.2. The Taco Cloud bootstrap class.

```
package tacos;

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

@SpringBootApplication    (1)
public class TacoCloudApplication {

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

}
```

(1) This is a Spring Boot application
(2) Run the application

Although there's very little code in `TacoCloudApplication`, what's there packs quite a punch. One of the most powerful lines of code is also one of the shortest lines of code. The `@SpringBootApplicationannotation` clearly signifies that this is a Spring Boot application. But there's more to `@SpringBootApplication` than meets the eye.

`@SpringBootApplication` is a composite application that combines three other annotations:

- `@SpringBootConfiguration` - Designates this class as a configuration class. Although there's not much configuration in the class at this time, we can add Java-based Spring Framework configuration to this class if we need to. This annotation is, in fact, a specialized form of the `@Configuration` annotation.
- `@EnableAutoConfiguration` - Enables Spring Boot automatic configuration. We'll talk more about auto-configuration later. For now just know that this annotation

tells Spring Boot to automatically configure any components that it thinks we need.

- `@ComponentScan` - Enables component scanning. This lets us declare other classes with annotations like `@Component`, `@Controller`, `@Service`, and others to have Spring automatically discover them and register them as components in the Spring application context.

The other important piece of `TacoCloudApplication` is the `main()` method. This is the method that will be run when the executable JAR file is run. For the most part, this method is boilerplate code; every Spring Boot application you write will have a method very similar or identical to this one (class name differences notwithstanding).

The `main()` calls a static `run()` method on the `SpringApplication` class which performs the actual bootstrapping of the application, creating the Spring application context. The two parameters passed to the `run()` method are a configuration class and the command line arguments. Although it's not necessary that the configuration class passed to `run()` be the same as the bootstrap class, this is the most convenient and typical choice.

Chances are you won't ever need to change anything in the bootstrap class. For simple applications, you might find it convenient to configure one or two other components in the bootstrap class, but for most applications you're better off creating a separate configuration class for anything that isn't auto-configured. We'll be defining several configuration classes throughout the course of this book, so stay tuned for details.

TESTING THE APPLICATION

Testing is a very important part of software development. Recognizing this, the Spring Initializr has given us a test class to get started. Listing 1.3 shows the baseline test class.

Listing 1.3. A baseline application test.

```
package tacos;

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

@RunWith(SpringRunner.class)     ❶
@SpringBootTest                  ❷
public class TacoCloudApplicationTests {

  @Test                          ❸
  public void contextLoads() {
  }

}
```

❶ Use the Spring runner

2. This is a Spring Boot test
3. The test method

There's not much to be seen in `TacoCloudApplicationTests`. Even the one test method in the class is empty. Even so, this test class does perform an essential check to be sure that the Spring application context can be loaded successfully. If we make any changes that prevent the Spring application context from being created, this test will fail and we can react by trying to fix the problem.

The class annotated with `@RunWith(SpringRunner.class)`. `@RunWith` is a JUnit annotation which provides a test runner that guides JUnit in running a test. Think of it as applying a plugin to JUnit to provide custom testing behavior. In this case, it is given `SpringRunner`, which is a Spring-provided test runner that provides for the creation of a Spring application context that the test will run against.

Note

A test runner by any other name...

If you're already familiar with writing Spring tests or are maybe looking at some existing Spring-based test classes, you may have seen a test runner named `SpringJUnit4ClassRunner`. `SpringRunner` is an alias for `SpringJUnit4ClassRunner` that was introduced in Spring 4.3 to remove the association with a specific version of JUnit (e.g., JUnit 4). And there's no denying that it's easier to read and type.

The `@SpringBootTest` tells JUnit to bootstrap the test with Spring Boot capabilities. For now, it's enough to think of this as the test class equivalent of calling `SpringApplication.run()` in a `main()` method. Over the course of this book, we'll see `@SpringBootTest` several times and uncover some of it's power.

Finally, there's the test method itself. While `@RunWith(SpringRunner.class)` and `@SpringBootTest` are tasked to load the Spring application context for the test, they won't have anything to do if there aren't any test methods. Even without any assertions or code of any kind, this empty test method will prompt the two annotations to do their job and load the Spring application context. If there are any problems in doing so, the test will fail.

At this point, we've concluded our review of the code provided by the Spring Initializr. We've seen some of the boilerplate foundation that we can develop a Spring application on, but we still haven't written a single line of code for ourselves. Now it's time to fire up your IDE and dust off your keyboard and add some custom code to the Taco Cloud application.

1.3 Writing a Spring application

Since we're just getting started, we're going to start off with a relatively small change to the Taco Cloud application, but one that will demonstrate a lot of Spring goodness. It seems appropriate that as we are just starting out, the first feature we should add to the Taco Cloud application is a home page.

As we add the home page, we're going to create two code artifacts:

- A controller class that handles requests for the home page.
- A view template that will define what the home page looks like.

And because testing is important, we'll also write a simple test class to test the home page.

But first things first…let's write that controller.

1.3.1 Handling web requests

Spring comes with a powerful web framework known as Spring MVC. At the center of Spring MVC is the concept of a controller, a class that handles requests and responds with information of some sort. In the case of a browser-facing application, a controller responds by optionally populating model data and passing the request on to a view to produce HTML that is returned to the browser.

We're going to learn a lot about Spring MVC in chapter 2. But for now, we're just going to write a simple controller class that handles requests for the root path (e.g., `/`) and forwards those requests to the home page view without populating any model data. Listing 1.4 shows our simple controller class.

Listing 1.4. The home page controller

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller                ❶
public class HomeController {

  @GetMapping("/")         ❷
  public String home() {
    return "home";         ❸
  }

}
```

❶ This is a controller
❷ Handle requests for "/"
❸ Return the view name

As you can see, this class is annotated with `@Controller`. On its own, `@Controller` doesn't do much. Its primary purpose in this class is to identify this class as a component for purposes of component-scanning. That is to say that because `HomeController` is annotated with `@Controller`, Spring component-scanning will automatically discover it and create an instance of `HomeController` as a bean in the Spring application context.

There are, in fact, a handful of other annotations that serve a similar purpose to @Controller, including @Component, @Service, and @Repository. We could have just as effectively annotated HomeControllerwith any of those other annotations and it would have still worked the same. The choice of @Controller is, however, more descriptive of this component's role in the application.

The home() method is as simple as controller methods come. It is annotated with @GetMapping to indicate that if an HTTP GET request is received for "/", then this method should handle that request. It does so by doing nothing more than returning a String value of "home".

This value is interpreted as the logical name of a view. How that view is implemented, depends on a few factors, but since we have Thymeleaf in our classpath, we can define that template with Thymeleaf.

Note

Why Thymeleaf?

You may be wondering why I chose Thymeleaf for a template engine? Why not JSP? Why not Freemarker? Why not one of several other options?

Put simply, I had to choose something and I like Thymeleaf and generally prefer it over those other options. And even though JSP may seem like an obvious choice, there are some challenges to overcome when using JSP with Spring Boot and I didn't want to go down that rabbit hole in chapter 1.

Hang tight. We'll look at other template options, including JSP, in chapter 2.

The template's name is derived from the logical view name by prefixing it with "/templates/" and postfixing it with ".html". The resulting path for the template is "/templates/home.html". Therefore, we'll need to place the template in our project at `/src/main/resources/templates/home.html'. Let's create that template now.

1.3.2 Defining the view

In the interest of keeping our home page simple, I've decided it should do nothing more than welcome users to the site. Listing 1.5 shows the basic Thymeleaf template that defines the Taco Cloud home page.

Listing 1.5. The Taco Cloud home page template.

```
<!DOCTYPE html>
<html
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Welcome to...</h1>
    <img th:src="@{/images/TacoCloud.png}"/>
  </body>
</html>
```

There's not much to discuss with regard to this template. The only notable line of code is the one with the `<img>` tag to display the Taco Cloud logo. It uses a Thymeleaf `th:src` attribute and an `@{…}`expression to reference the image with a context-relative path. Aside from that, it's not much more than a "Hello World" page.

But let's talk about that image a bit more. I'll leave it up to you to define a Taco Cloud logo that you like. You'll just need to make sure you place it at the right place within the project.

The image is referenced with a context-relative path of "/images/TacoCloud.png". As you'll recall from our review of the project structure, static content such as images are to be kept in `/src/main/resources/static`. That means that our Taco Cloud logo image must reside within our project at `/src/main/resources/static/images/TacoCloud.png`.

Now that we have a controller to handle requests for the home page and a view template to render the home page, we're almost ready to fire up the application and see it in action. But first, let's see how we can write a test against the controller.

1.3.3 Testing the controller

Testing web applications can be tricky, especially when making assertions against the content of an HTML page. Fortunately, Spring comes with some powerful test support that makes testing a web application easy.

For the purposes of the home page, we'll write a test that's comparable in complexity to the home page itself. Our test will simply perform an HTTP `GET` request for "/" and expect a successful result where the view name is "home" and that the resulting content contains the phrase "Welcome to…". Listing 1.6 should do the trick.

Listing 1.6. A test for the home page controller.

```
package tacos;

import static org.hamcrest.Matchers.containsString;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@RunWith(SpringRunner.class)
@WebMvcTest(HomeController.class)                ❶
public class HomeControllerTest {
  @Autowired
  private MockMvc mockMvc;                         ❷

  @Test
  public void testHomePage() throws Exception {
    mockMvc.perform(get("/"))                      ❸

      .andExpect(status().isOk())                  ❹

      .andExpect(view().name("home"))              ❺

      .andExpect(content().string(                 ❻
          containsString("Welcome to...")));
  }

}
```

❶ Web test for HomeController
❷ Inject MockMvc
❸ Perform GET /
❹ Expect HTTP 200
❺ Expect "home" view
❻ Expect "Welcome to…"

The first thing you might notice about this test is that it differs slightly from the `TacoCloudApplicationTests` class with regard to the annotations applied to it. Instead of a `@SpringBootTest` annotation, `HomeControllerTest` is annotated with `@WebMvcTest`. This annotation is a special test annotation provided by Spring Boot that arranges for the test to run in the context of a Spring MVC application. More specifically, in this case, it arranges for `HomeController` to be registered in Spring MVC so that we can throw requests against it.

`@WebMvcTest` also sets up Spring's support for testing Spring MVC. Although it could be made to start an actual server, mocking the mechanics of Spring MVC is sufficient for our purposes. The test class is injected with a `MockMvc` object for the test to drive the mock.

The `testHomePage()` method defines the test we want to perform against the home page. It starts by the `MockMvc` object to perform an HTTP `GET` request for "/". And from that request, it sets the following expectations:

- The response should have an HTTP 200 (OK) status.
- The view should have a logical name of "home".
- The rendered view should contain the text "Welcome to…"

If, after performing the request, any of those expectations are not met, then the test will fail. But our controller and view template are written to satisfy those expectations, so the test should pass with flying colors—or at least with some shade of green indicating a passing test.

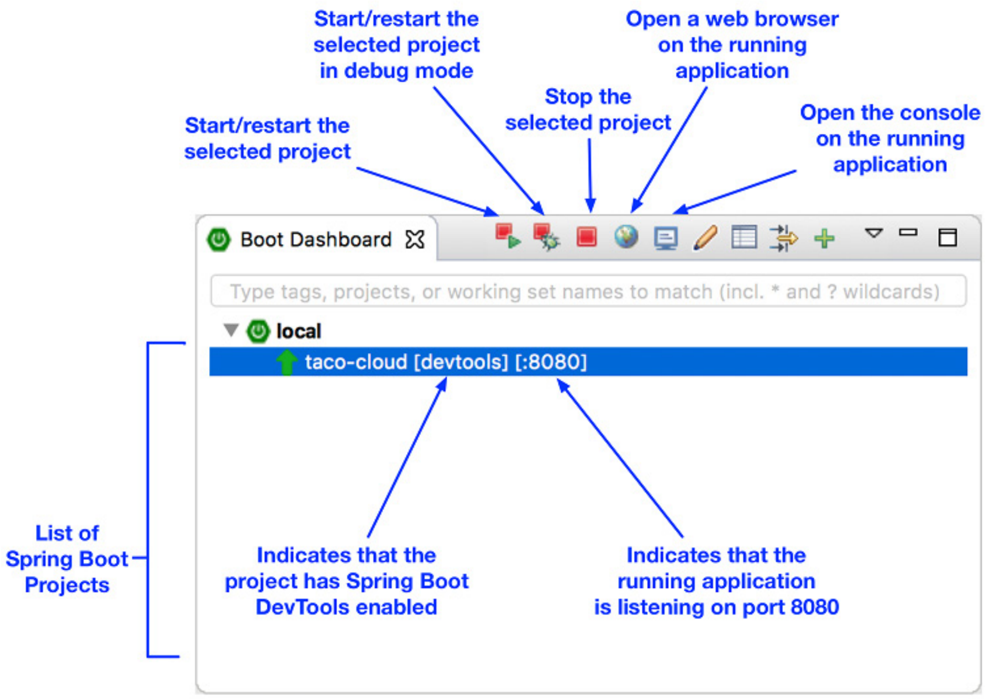
The controller has been written, the view template created, and we have a passing test. It seems that we've implemented the home page successfully. But even though our test passes, there's something slightly more satisfying with seeing the results in a browser. After all, that's how Taco Cloud customers are going to see it. So let's build the application and run it.

1.3.4 Building and running the application

Just as there are several ways to initialize a Spring application, there are several ways to run a Spring application. You can flip over to Appendix A to read about some of the more common ways to run a Spring Boot application.

Since I chose to use Spring Tool Suite to initialize and work on the project, I have a handy feature called the Spring Boot Dashboard available to help me run my application inside of the IDE. The Spring Boot Dashboard appears as a tab, typically near the bottom left of the IDE window. Figure 1.7 shows an annotated screenshot of the Spring Boot Dashboard.

Figure 1.7. Highlights of the Spring Boot Dashboard.


Start/restart the selected project in debug mode
Open a web browser on the running application
Stop the selected project
Start/restart the selected project
Open the console on the running application
Boot Dashboard
Type tags, projects, or working set names to match (incl. * and ? wildcards)
local
taco-cloud [devtools] [:8080]
List of Spring Boot Projects
Indicates that the project has Spring Boot DevTools enabled
Indicates that the running application is listening on port 8080

I don't want to spend much time going over everything that the Spring Boot Dashboard does, although Figure 1.7 covers some of the most useful details. The important thing to know right now is how to use it to run the Taco Cloud application. Just make sure that the "taco-cloud" application is highlighted in the list of projects (it's the only application shown in Figure 1.7) and then click the start button (the left-most button with both a green triangle and a red square). The application should start right up.

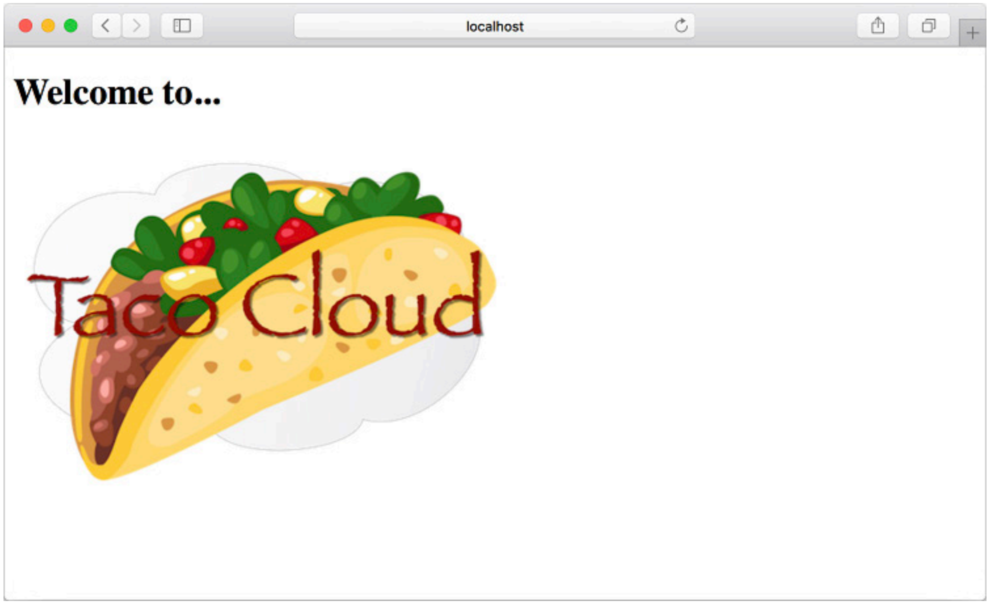
As the application starts up, you’ll see some Spring ASCII-art fly by in the console, followed by some log entries describing the steps as the application starts up. Just before the logging stops, you’ll see a log entry saying "Tomcat started on port(s): 8080 (http)", which means that we’re ready to point our web browser at the home page to see the fruits of our labor.

Wait a minute. Tomcat started? When did we deploy the application to Tomcat?

Spring Boot applications tend to bring everything they need with them and don’t need to be deployed to some application server. We never deployed our application to Tomcat…Tomcat is a part of our application! (I’ll describe the details of how Tomcat became part of our application in section “Let’s review”.)

Now that the application has started, just point your web browser to localhost:8080 (or click the globe button in the Spring Boot Dashboard) and you should see something like Figure 1.8 .

Figure 1.8. The Taco Cloud home page



Your results may be different if you designed your own logo image. But it shouldn’t vary much from what you see in Figure 1.8 .

It may not be much to look at. But this isn’t exactly a book on graphic design. The humble appearance of the home page is more than sufficient for now. And it provided us a solid start on getting to know Spring.

One thing I’ve glossed over up until now is DevTools. We selected it as a dependency when intializing our project. It appears as a dependency in the produced `pom.xml` file. And the Spring Boot Dashboard even shows that the project has DevTools enabled. But what is DevTools and what does it do for us? Let’s take a quick survey of a couple of DevTools' most useful features.

1.3.5 Getting to know Spring Boot DevTools

As its name suggests, DevTools provides Spring developers with some handy development-time tools. Among those are:

- Automatic application restart when code changes.
- Automatic browser refresh when browser-destined resources (such as templates, JavaScript, stylesheets, etc) change.
- Automatic disable of template caches.
- Built in H2 Console if the H2 database is in use.

It’s important to understand that DevTools isn’t an IDE plugin, nor does it require that you use a specific IDE. It works equally well in Spring Tool Suite, IntelliJ IDEA, and Netbeans. Furthermore, because it is only intended for development purposes, it’s smart enough to disable itself when deploying in a production setting. (We’ll discuss how it does this when we get around to deploying our application in chapter 17.)

For now, let’s focus on the most useful features of Boot DevTools, starting with automatic application restart.

AUTOMATIC APPLICATION RESTART

With DevTools as part of your project, you’ll be able to make changes to Java code and properties files in the project and see those changes applied after a brief moment. DevTools monitors for changes and when it sees something has changed, it will automatically restart the application.

More precisely, when DevTools is in play, the application is loaded into two separate class loaders in the Java virtual machine (JVM). One class loader is loaded with your Java code, property files, and pretty much anything that’s in the `src/main/` path of the project. These are items that are likely to change frequently. The other class loader is loaded with dependency libraries, which are not as likely to change often.

When a change is detected, DevTools reloads only the classloader containing your project’s code and restarts the Spring application context, but leaves the other class loader and the JVM intact. Although subtle, this strategy affords a small reduction in the time that the application starts.

The downside of this strategy is that changes to dependencies will not be available in automatic restarts. That’s because the class loader containing dependency libraries is not automatically reloaded. This means that any time you add, change, or remove a dependency in your build specification, you’ll need to do a hard restart of the application for those changes to take effect.

AUTOMATIC BROWSER REFRESH AND TEMPLATE CACHE DISABLE

By default, template options such as Thymeleaf and Freemarker are configured to cache the results of template parsing so that the templates don't need to be reparsed with every request that they serve. This is great in production, as it buys a little bit of performance benefit.

Cached templates, however, are not so great at development time. Cached templates make it impossible to make changes to the templates while the application is running and see the results after refreshing the browser. Even if you've made changes, the cached template will still be in use until you restart the application.

DevTools addresses this issue by automatically disabling all template caching. Make as many changes as you want to your templates and know that you're only a browser refresh away from seeing the results.

But if you're like me, you don't even want to be burdened with the effort of clicking the browser's refresh button. It'd be much nicer if you could simply make the changes and witness the results in the browser immediately. Fortunately, DevTools has something special for those of us who are too lazy to click a refresh button.

When DevTools is in play, it automatically enables a LiveReload[2] server along with your application. By itself, the LiveReload server isn't very useful. But when coupled with a corresponding LiveReload browser plugin, it can cause your browser to automatically refresh when changes are made to templates, images, stylesheets, JavaScript…almost anything that ends up being served to your browser.

LiveReload has browser plugins for Google Chrome, Safari, and Firefox browsers. (Sorry Internet Explorer fans.) Visit livereload.com/extensions/ to find information on how to install LiveReload for your browser.

BUILT IN H2 CONSOLE

Although our project doesn't yet leverage a database, that will change soon in chapter 3. If you choose to use the H2 database for development, DevTools will also automatically enable an H2 Console that you can access from your web browser. You only need to point your web browser to localhost:8080/h2-console to gain insight into the data your application is working with.

At this point, we've written a complete, albeit simple, Spring application. We'll expand on it throughout the course of the book. But now is a good time to step back and review what we have accomplished and how Spring played a part.

1.3.6 Let's review

Think back on how we got to this point. In short, these are the steps we've taken to build our Spring-based Taco Cloud application:

1. We created an initial project structure using Spring Initializr.

[2] livereload.com/

2. We wrote a controller class to handle the home page request.
3. We defined a view template to render the home page.
4. We wrote a simple test class to prove out our work.

Seems pretty straightforward, doesn't it? With the exception of the first step to bootstrap the project, each action we've taken has been keenly focused on achieving the goal of producing a home page.

In fact, almost every line of code we've written is aimed toward that same goal. Not counting Java `import` statements, I count only 2 lines of code in our controller class that are specific to Spring and no lines in the view template that are Spring specific. And while the bulk of the test class utilizes Spring's testing support, it seems a little less invasive in the context of a test.

That's an important benefit of developing with Spring. You can focus on the code that meets the requirements of an application rather than on satisfying the demands of a framework. Although you'll no doubt need to write some framework-specific code from time to time, it will usually be only a small fraction of your codebase. As I said before, Spring (with Spring Boot) can be considered the "framework-less framework".

So how does this even work? What is Spring doing behind the scenes to make sure that your application's needs are met?

To understand what Spring is doing, let's start by looking at the build specification. In the `pom.xml` file, we declared a dependency on the "web" and "thymeleaf" starters. These two dependencies transitively brought in a handful of other dependencies, including:

- Spring's MVC framework
- Embedded Tomcat
- Thymeleaf and the Thymeleaf layout dialect

It also brings Spring Boot's auto-configuration library along for the ride. When the application starts up, Spring Boot auto-configuration detects those libraries and automatically…

- …configures the beans in the Spring application context to enable Spring MVC.
- …configures the embedded Tomcat server in the Spring application context.
- …configures a Thymeleaf view resolver for rendering Spring MVC views with Thymeleaf templates.

In short, auto-configuration does all of the grunt work, leaving you to focus on writing code implements your application's functionality.

That's a pretty sweet arrangement if you ask me!

Our Spring journey has just begun. The Taco Cloud application has only touched on a small portion of what Spring has to offer. Before we take our next step, let's survey the Spring landscape and see what landmarks we'll encounter on our journey./

1.4 Surveying the Spring landscape

To get an idea of what the Spring landscape looks like, you can look no further than the enormous list of checkboxes on the full version of the Spring Initializr's web form. There are over 100 dependency choices listed, so I won't even bother trying to list them all here or provide a screenshot. But I encourage you to take a look. In the meantime, I'll mention of few of the highlights.

1.4.1 The Core Spring Framework

As you might expect, the core Spring Framework is the foundation of everything else in the Spring universe. It provides the core container and dependency injection framework. But it also provides a few other essential features.

Among those is Spring MVC, Spring's web framework. We've already seen how to use Spring MVC to write a controller class to handle web requests. What you've not seen yet, however, is that Spring MVC can also be used create REST APIs that produce non-HTML output. We're going to dig more into Spring MVC in chapter 2 and then take another look at how to use it to create REST APIs in chapter 6.

The core Spring Framework also offers some elemental data persistence support, specifically template-based JDBC support. We'll see how to use `JdbcTemplate` in chapter 3.

In the most recent version of Spring (5.0.0), Spring has added support for reactive style programming, including a new reactive web framework called Spring WebFlux that borrows heavily from Spring MVC. We'll look at Spring's reactive programming model in Part 3 and Spring WebFlux specifically in chapter 10.

1.4.2 Spring Boot

We've already seen many of the benefits of Spring Boot, including starter dependencies and auto-configuration. Be certain that we'll leverage as much of Spring Boot as possible throughout this book and avoid any form of explicit configuration unless it's absolutely necessary.

But in addition to starter dependencies and auto-configuration, Spring Boot also offers a handful of other useful features:

- The Actuator provides runtime insight into the inner workings of an application, including metrics, thread dump information, application health, and environment properties available to the application.
- Flexible specification of environment properties.
- Additional testing support on top of testing support found in the core framework.

What's more, Spring Boot offers an alternative programming model based on Groovy scripts called the Spring Boot CLI (Command Line Interface). With the Spring Boot CLI, you can write entire applications as a collection of Groovy scripts and run them from the command line. We won't spend much time with the Spring Boot CLI, but we will touch upon it on occasion when it befits our needs.

Spring Boot has become such an integral part of Spring development and I can't imagine developing a Spring application without it. Consequently, this book will take a very Spring Boot-centric view and you might catch me using the work "Spring" when I am referring to something that Spring Boot is doing.

1.4.3 Spring Data

Although the core Spring Framework comes with basic data persistence support, Spring Data provides something that is quite amazing: The ability to define your application's data repositories as simple Java interfaces, using a naming convention when defining methods to drive how data is stored and retrieved.

What's more, Spring Data is capable of working with a several different kinds of databases, including relational (JPA), document (Mongo), graph (Neo4j), and others.

We'll leverage Spring Data to help create repositories for the Taco Cloud application in chapter 3.

1.4.4 Spring Security

Application security has always been an important topic, and it seems to become more important every day. Fortunately, Spring has a robust security framework in Spring Security.

Spring Security addresses a broad range of application security needs, including authentication, authorization, and API security.

Although the scope of Spring Security is too large to be properly covered in this book, we'll touch on some of the most common use cases in chapters 4 and 12.

1.4.5 Spring Integration and Spring Batch

At some point, most applications will need to integrate with other applications or even with other components of the same application. There are several patterns of application integration that have emerged to address these needs. Spring Integration and Spring Batch provide the implementation of these patterns for Spring-based applications.

Spring Integration addresses real-time integration, where data is processed as it is made available. In contrast, Spring Batch addresses batched integration where data is allowed to collect for a time until some trigger (perhaps a time trigger) signals that it is time for the batch of data to be processed.

We'll explore both Spring Batch and Spring Integration in chapter 8.

1.4.6 Spring Cloud

As I'm writing this, the application development world is entering a new era where we will no longer develop our applications as single deployment unit monoliths and will instead compose applications from several individual deployment units known as microservices.

Microservices are a hot topic and they do address several practical development and runtime concerns. In doing so, however, they bring to fore their own challenges. Those challenges are met head-on by Spring Cloud, a collection of projects for developing cloud-native applications with Spring.

Spring Cloud covers a lot of ground and it'd be impossible to cover it all in this book. We will look at some of the most common components of Spring Cloud in chapters 13, 14, and 15. For a more complete discussion of Spring Cloud, I suggest taking a look at *Spring Microservices in Action* by John Carnell[3] .

1.5 Summary

- Spring aims to make make developer challenges—such as creating web applications, working with databases, securing applications, and microservices—easy.
- Spring Boot builds on top of Spring to make Spring even easier with simplified dependency management, automatic configuration, and runtime insights.
- Spring applications can be initialized using the Spring Initializr, which is web-based and supported natively in most Java development environments.
- The components, commonly referred to as beans, in a Spring application context can be declared explicitly with Java or XML, discovered by component-scanning, or automatically configured with Spring Boot auto-configuration.

[3] www.manning.com/books/spring-microservices-in-action