you that, and showing it again wouldn't help us get the reading-list application written any quicker.

Instead of wasting time talking about Spring configuration, knowing that Spring Boot is going to take care of that for us, let's see how taking advantage of Spring Boot auto-configuration keeps us focused on writing application code. I can think of no better way to do that than to start writing the application code for the reading-list application.

DEFINING THE DOMAIN

The central domain concept in our application is a book that's on a reader's reading list. Therefore, we'll need to define an entity class that represents a book. Listing 2.5 shows how the `Book` type is defined.

Listing 2.5 The `Book` class represents a book in the reading list

```
package readinglist;

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

@Entity
public class Book {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String reader;
  private String isbn;
  private String title;
  private String author;
  private String description;

  public Long getId() {
    return id;
  }

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

  public String getReader() {
    return reader;
  }

  public void setReader(String reader) {
    this.reader = reader;
  }

  public String getIsbn() {
    return isbn;
  }

  public void setIsbn(String isbn) {
    this.isbn = isbn;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

}
```

As you can see, the `Book` class is a simple Java object with a handful of properties describing a book and the necessary accessor methods. It's annotated with `@Entity` designating it as a JPA entity. The `id` property is annotated with `@Id` and `@GeneratedValue` to indicate that this field is the entity's identity and that its value will be automatically provided.

DEFINING THE REPOSITORY INTERFACE

Next up, we need to define the repository through which the `ReadingList` objects will be persisted to the database. Because we're using Spring Data JPA, that task is a simple matter of creating an interface that extends Spring Data JPA's `JpaRepository` interface:

```
package readinglist;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ReadingListRepository extends JpaRepository<Book, Long> {

  List<Book> findByReader(String reader);

}
```

By extending JpaRepository, ReadingListRepository inherits 18 methods for performing common persistence operations. The JpaRepository interface is parameterized with two parameters: the domain type that the repository will work with, and the type of its ID property. In addition, I've added a findByReader() method through which a reading list can be looked up given a reader's username.

If you're wondering about who will implement ReadingListRepository and the 18 methods it inherits, don't worry too much about it. Spring Data provides a special magic of its own, making it possible to define a repository with just an interface. The interface will be implemented automatically at runtime when the application is started.

CREATING THE WEB INTERFACE

Now that we have the application's domain defined and a repository for persisting objects from that domain to the database, all that's left is to create the web front-end. A Spring MVC controller like the one in listing 2.6 will handle HTTP requests for the application.

Listing 2.6 A Spring MVC controller that fronts the reading list application

```
package readinglist;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.List;

@Controller
@RequestMapping("/")
public class ReadingListController {

  private ReadingListRepository readingListRepository;

  @Autowired
  public ReadingListController(
             ReadingListRepository readingListRepository) {
    this.readingListRepository = readingListRepository;
  }

  @RequestMapping(value="/{reader}", method=RequestMethod.GET)
  public String readersBooks(
      @PathVariable("reader") String reader,
      Model model) {

    List<Book> readingList =
        readingListRepository.findByReader(reader);
    if (readingList != null) {
      model.addAttribute("books", readingList);
    }
    return "readingList";
  }
  @RequestMapping(value="/{reader}", method=RequestMethod.POST)
  public String addToReadingList(
            @PathVariable("reader") String reader, Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/{reader}";
  }

}
```

`ReadingListController` is annotated with `@Controller` in order to be picked up by component-scanning and automatically be registered as a bean in the Spring application context. It's also annotated with `@RequestMapping` to map all of its handler methods to a base URL path of "/".

The controller has two methods:

- `readersBooks()`—Handles HTTP `GET` requests for /{reader} by retrieving a `Book` list from the repository (which was injected into the controller's constructor) for the reader specified in the path. It puts the list of `Book` into the model under the key "books" and returns "readingList" as the logical name of the view to render the model.
- `addToReadingList()`—Handles HTTP `POST` requests for /{reader}, binding the data in the body of the request to a `Book` object. This method sets the `Book` object's `reader` property to the reader's name, and then saves the modified `Book` via the repository's `save()` method. Finally, it returns by specifying a redirect to /{reader} (which will be handled by the other controller method).

The `readersBooks()` method concludes by returning "readingList" as the logical view name. Therefore, we must also create that view. I decided at the outset of this project that we'd be using Thymeleaf to define the application views, so the next step is to create a file named readingList.html in src/main/resources/templates with the following content.

Listing 2.7 The Thymeleaf template that presents a reading list

```
<html>
  <head>
    <title>Reading List</title>
    <link rel="stylesheet" th:href="@{/style.css}"></link>
  </head>

  <body>
    <h2>Your Reading List</h2>
    <div th:unless="${#lists.isEmpty(books)}">
      <dl th:each="book : ${books}">
        <dt class="bookHeadline">
          <span th:text="${book.title}">Title</span> by
          <span th:text="${book.author}">Author</span>
          (ISBN: <span th:text="${book.isbn}">ISBN</span>)
        </dt>
        <dd class="bookDescription">
          <span th:if="${book.description}"
                th:text="${book.description}">Description</span>
          <span th:if="${book.description eq null}">
                No description available</span>
        </dd>
      </dl>
    </div>
    <div th:if="${#lists.isEmpty(books)}">
      <p>You have no books in your book list</p>
    </div>

    <hr/>

    <h3>Add a book</h3>
    <form method="POST">
      <label for="title">Title:</label>
        <input type="text" name="title" size="50"></input><br/>
      <label for="author">Author:</label>
        <input type="text" name="author" size="50"></input><br/>
      <label for="isbn">ISBN:</label>
        <input type="text" name="isbn" size="15"></input><br/>
      <label for="description">Description:</label><br/>
        <textarea name="description" cols="80" rows="5">
        </textarea><br/>
      <input type="submit"></input>
    </form>

  </body>
</html>
```

This template defines an HTML page that is conceptually divided into two parts. At the top of the page is a list of books that are in the reader's reading list. At the bottom is a form the reader can use to add a new book to the reading list.

For aesthetic purposes, the Thymeleaf template references a stylesheet named style.css. That file should be created in src/main/resources/static and look like this:

```
body {
    background-color: #cccccc;
    font-family: arial,helvetica,sans-serif;
}

.bookHeadline {
    font-size: 12pt;
    font-weight: bold;
}

.bookDescription {
    font-size: 10pt;
}

label {
    font-weight: bold;
}
```

This stylesheet is simple and doesn't go overboard to make the application look nice. But it serves our purposes and, as you'll soon see, serves to demonstrate a piece of Spring Boot's auto-configuration.

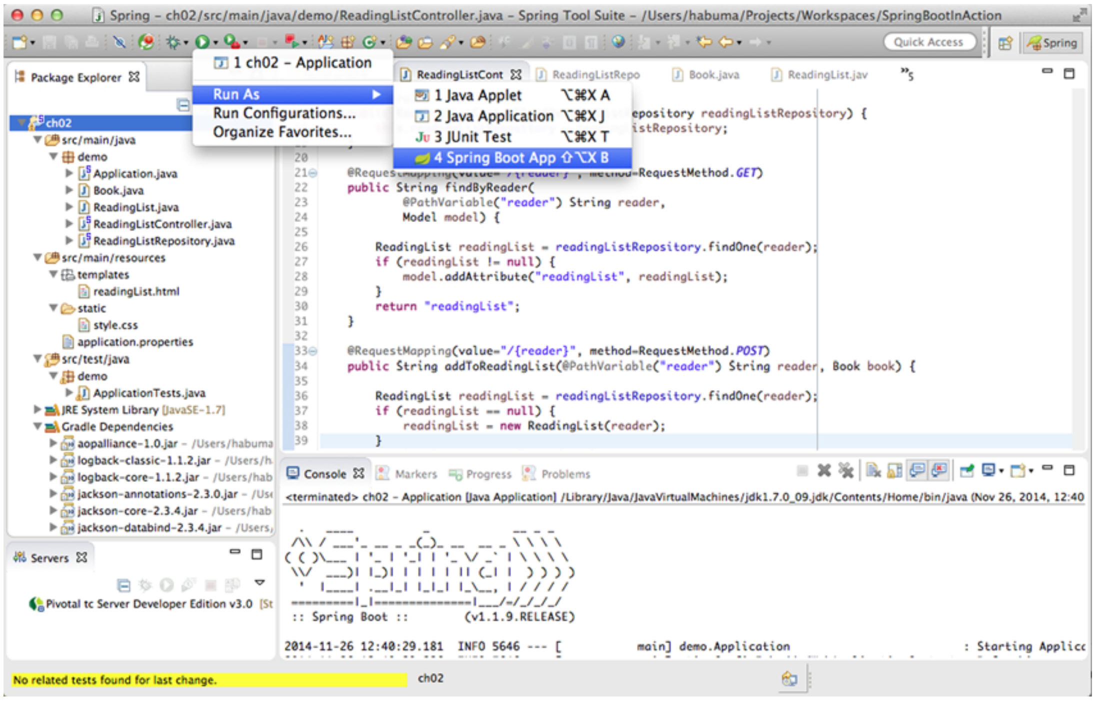
Believe it or not, that's a complete application. Every single line has been presented to you in this chapter. Take a moment, flip back through the previous pages, and see if you can find any configuration. In fact, aside from the three lines of configuration in listing 2.1 (which essentially turn on auto-configuration), you didn't have to write any Spring configuration.

Despite the lack of Spring configuration, this complete Spring application is ready to run. Let's fire it up and see how it looks.

2.3.2 Running the application

There are several ways to run a Spring Boot application. Earlier, in section 2.5, we discussed how to run the application via Maven and Gradle, as well as how to build and run an executable JAR. Later, in chapter 8 you'll also see how to build a WAR file that can be deployed in a traditional manner to a Java web application server such as Tomcat.

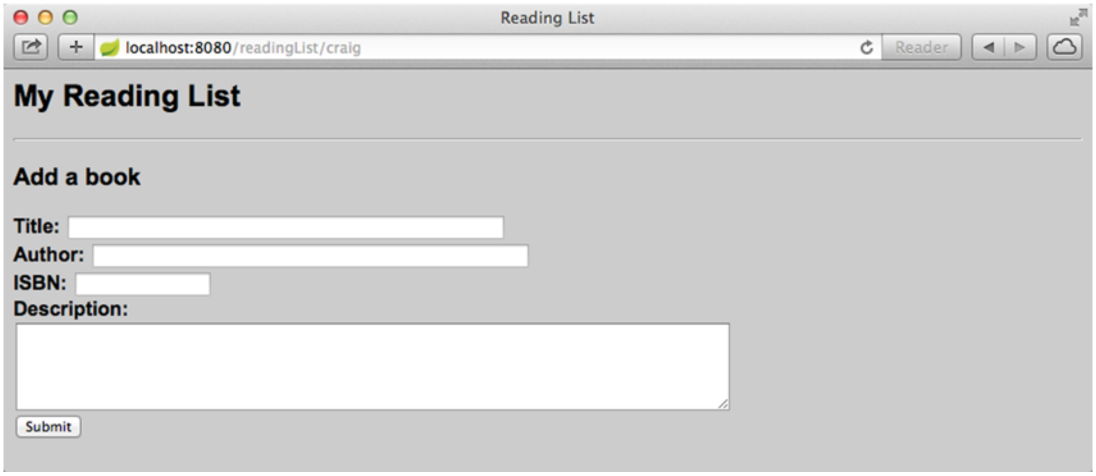
If you're developing your application with Spring Tool Suite, you also have the option of running the application within your IDE by selecting the project and choosing Run As > Spring Boot App from the Run menu, as shown in figure 2.3.


Spring – ch02/src/main/java/demo/ReadingListController.java – Spring Tool Suite – /Users/habuma/Projects/Workspaces/SpringBootInAction
Quick Access
Spring
Package Explorer
1 ch02 – Application
Run As
Run Configurations...
Organize Favorites...
1 Java Applet ⌥⌘X A
2 Java Application ⌥⌘X J
3 JUnit Test ⌥⌘X T
4 Spring Boot App ⇧⌥X B
ReadingListCont
ReadingListRepo
Book.java
ReadingList.jav
ch02
src/main/java
demo
Application.java
Book.java
ReadingList.java
ReadingListController.java
ReadingListRepository.java
src/main/resources
templates
readingList.html
static
style.css
application.properties
src/test/java
demo
ApplicationTests.java
JRE System Library [JavaSE-1.7]
Gradle Dependencies
aopalliance-1.0.jar – /Users/habuma
logback-classic-1.1.2.jar – /Users/h
logback-core-1.1.2.jar – /Users/hab
jackson-annotations-2.3.0.jar – /Us
jackson-core-2.3.4.jar – /Users/hab
jackson-databind-2.3.4.jar – /Users,
Servers
Pivotal tc Server Developer Edition v3.0 [St
Console
Markers
Progress
Problems
No related tests found for last change.
ch02

```
epository readingListRepository) {
istRepository;

    @RequestMapping(value="/{reader}", method=RequestMethod.GET)
    public String findByReader(
            @PathVariable("reader") String reader,
            Model model) {

        ReadingList readingList = readingListRepository.findOne(reader);
        if (readingList != null) {
            model.addAttribute("readingList", readingList);
        }
        return "readingList";
    }

    @RequestMapping(value="/{reader}", method=RequestMethod.POST)
    public String addToReadingList(@PathVariable("reader") String reader, Book book) {

        ReadingList readingList = readingListRepository.findOne(reader);
        if (readingList == null) {
            readingList = new ReadingList(reader);
        }
```

```
<terminated> ch02 – Application [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_09.jdk/Contents/Home/bin/java (Nov 26, 2014, 12:40

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.1.9.RELEASE)

2014-11-26 12:40:29.181  INFO 5646 --- [           main] demo.Application                         : Starting Applic
```

Figure 2.3 Running a Spring Boot application from Spring Tool Suite



My Reading List

Add a book

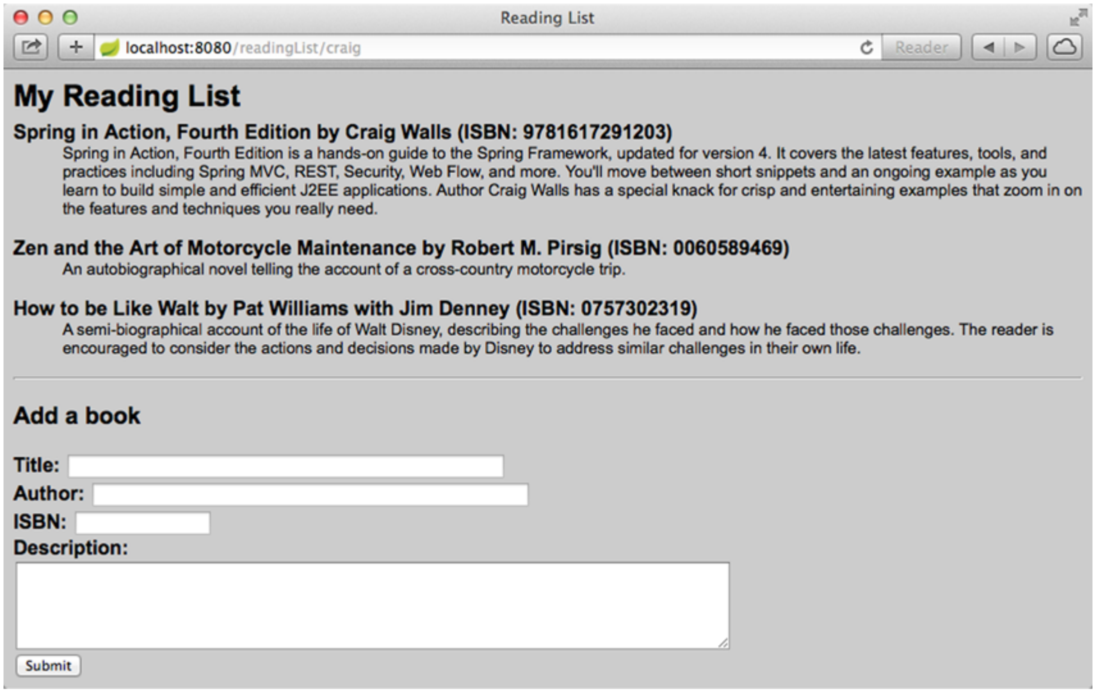
Title:
Author:
ISBN:
Description:

Submit

Figure 2.4 An initially empty reading list

Assuming everything works, your browser should show you an empty reading list along with a form for adding a new book to the list. Figure 2.4 shows what it might look like.

Now go ahead and use the form to add a few books to your reading list. After you do, your list might look something like figure 2.5.



My Reading List

Spring in Action, Fourth Edition by Craig Walls (ISBN: 9781617291203)
Spring in Action, Fourth Edition is a hands-on guide to the Spring Framework, updated for version 4. It covers the latest features, tools, and practices including Spring MVC, REST, Security, Web Flow, and more. You'll move between short snippets and an ongoing example as you learn to build simple and efficient J2EE applications. Author Craig Walls has a special knack for crisp and entertaining examples that zoom in on the features and techniques you really need.

Zen and the Art of Motorcycle Maintenance by Robert M. Pirsig (ISBN: 0060589469)
An autobiographical novel telling the account of a cross-country motorcycle trip.

How to be Like Walt by Pat Williams with Jim Denney (ISBN: 0757302319)
A semi-biographical account of the life of Walt Disney, describing the challenges he faced and how he faced those challenges. The reader is encouraged to consider the actions and decisions made by Disney to address similar challenges in their own life.

Add a book

Title:
Author:
ISBN:
Description:

Submit

Figure 2.5 The reading list after a few books have been added

Feel free to take a moment to play around with the application. When you're ready, move on and we'll see how Spring Boot made it possible to write an entire Spring application with no Spring configuration code.

2.3.3 What just happened?

As I said, it's hard to describe auto-configuration when there's no configuration to point at. So instead of spending time discussing what you don't have to do, this section has focused on what you do need to do—namely, write the application code.

But certainly there is some configuration somewhere, right? Configuration is a central element of the Spring Framework, and there must be something that tells Spring how to run your application.

When you add Spring Boot to your application, there's a JAR file named spring-boot-autoconfigure that contains several configuration classes. Every one of these configuration classes is available on the application's classpath and has the opportunity to contribute to the configuration of your application. There's configuration for Thymeleaf, configuration for Spring Data JPA, configuration for Spring MVC, and configuration for dozens of other things you might or might not want to take advantage of in your Spring application.

What makes all of this configuration special, however, is that it leverages Spring's support for conditional configuration, which was introduced in Spring 4.0. Conditional configuration allows for configuration to be available in an application, but to be ignored unless certain conditions are met.

It's easy enough to write your own conditions in Spring. All you have to do is implement the `Condition` interface and override its `matches()` method. For example, the following simple condition class will only pass if `JdbcTemplate` is available on the classpath:

```
package readinglist;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class JdbcTemplateCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context,
                         AnnotatedTypeMetadata metadata) {
    try {
      context.getClassLoader().loadClass(
             "org.springframework.jdbc.core.JdbcTemplate");
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

You can use this custom condition class when you declare beans in Java:

```
@Conditional(JdbcTemplateCondition.class)
public MyService myService() {
    ...
}
```

In this case, the `MyService` bean will only be created if the `JdbcTemplateCondition` passes. That is to say that the `MyService` bean will only be created if `JdbcTemplate` is available on the classpath. Otherwise, the bean declaration will be ignored.

Although the condition shown here is rather simple, Spring Boot defines several more interesting conditions and applies them to the configuration classes that make up Spring Boot auto-configuration. Spring Boot applies conditional configuration by defining several special conditional annotations and using them in its configuration classes. Table 2.1 lists the conditional annotations that Spring Boot provides.

Table 2.1 Conditional annotations used in auto-configuration

Conditional annotation	Configuration applied if…?
`@ConditionalOnBean`	…the specified bean has been configured
`@ConditionalOnMissingBean`	…the specified bean has not already been configured
`@ConditionalOnClass`	…the specified class is available on the classpath
`@ConditionalOnMissingClass`	…the specified class is not available on the classpath
`@ConditionalOnExpression`	…the given Spring Expression Language (SpEL) expression evaluates to `true`
`@ConditionalOnJava`	…the version of Java matches a specific value or range of versions
`@ConditionalOnJndi`	…there is a JNDI `InitialContext` available and optionally given JNDI locations exist
`@ConditionalOnProperty`	…the specified configuration property has a specific value
`@ConditionalOnResource`	…the specified resource is available on the classpath
`@ConditionalOnWebApplication`	…the application is a web application
`@ConditionalOnNotWebApplication`	…the application is not a web application

Generally, you shouldn't ever need to look at the source code for Spring Boot's auto-configuration classes. But as an illustration of how the annotations in table 2.1 are used, consider this excerpt from `DataSourceAutoConfiguration` (provided as part of Spring Boot's auto-configuration library):

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
```

```
@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class
      })
public class DataSourceAutoConfiguration {

...

}
```

As you can see, `DataSourceAutoConfiguration` is a `@Configuration`-annotated class that (among other things) imports some additional configuration from other configuration classes and defines a few beans of its own. What's most important to notice here is that `DataSourceAutoConfiguration` is annotated with `@ConditionalOnClass` to require that both `DataSource` and `EmbeddedDatabaseType` be available on the classpath. If they aren't available, then the condition fails and any configuration provided by `DataSourceAutoConfiguration` will be ignored.

Within `DataSourceAutoConfiguration` there's a nested `JdbcTemplateConfiguration` class that provides auto-configuration of a `JdbcTemplate` bean:

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

  @Autowired(required = false)
  private DataSource dataSource;

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

...

}
```

`JdbcTemplateConfiguration` is an annotation with the low-level `@Conditional` to require that the `DataSourceAvailableCondition` pass—essentially requiring that a `DataSource` bean be available or that one will be created by auto-configuration. Assuming that a `DataSource` bean will be available, the `@Bean`-annotated `jdbcTemplate()` method configures a `JdbcTemplate` bean. But `jdbcTemplate()` is annotated with `@ConditionalOnMissingBean` so that the bean will be configured only if there is not already a bean of type `JdbcOperations` (the interface that `JdbcTemplate` implements).

There's a lot more to `DataSourceAutoConfiguration` and to the other auto-configuration classes provided by Spring Boot than is shown here. But this should give you a taste of how Spring Boot leverages conditional configuration to implement auto-configuration.

As it directly pertains to our example, the following configuration decisions are made by the conditionals in auto-configuration:

- Because H2 is on the classpath, an embedded H2 database bean will be created. This bean is of type `javax.sql.DataSource`, which the JPA implementation (Hibernate) will need to access the database.
- Because Hibernate Entity Manager is on the classpath (transitively via Spring Data JPA), auto-configuration will configure beans needed to support working with Hibernate, including Spring's `LocalContainerEntityManagerFactoryBean` and `JpaVendorAdapter`.
- Because Spring Data JPA is on the classpath, Spring Data JPA will be configured to automatically create repository implementations from repository interfaces.
- Because Thymeleaf is on the classpath, Thymeleaf will be configured as a view option for Spring MVC, including a Thymeleaf template resolver, template engine, and view resolver. The template resolver is configured to resolve templates from /templates relative to the root of the classpath.
- Because Spring MVC is on the classpath (thanks to the web starter dependency), Spring's `DispatcherServlet` will be configured and Spring MVC will be enabled.
- Because this is a Spring MVC web application, a resource handler will be registered to serve static content from /static relative to the root of the classpath. (The resource handler will also serve static content from /public, /resources, and /META-INF/resources).
- Because Tomcat is on the classpath (transitively referred to by the web starter dependency), an embedded Tomcat container will be started to listen on port 8080.

The main takeaway here, though, is that Spring Boot auto-configuration takes on the burden of configuring Spring so that you can focus on writing your application.

2.4 Summary

By taking advantage of Spring Boot starter dependencies and auto-configuration, you can more quickly and easily develop Spring applications. Starter dependencies help you focus on the type of functionality your application needs rather than on the specific libraries and versions that provide that functionality. Meanwhile, auto-configuration frees you from the boilerplate configuration that is common among Spring applications without Spring Boot.

Although auto-configuration is a convenient way to work with Spring, it also represents an opinionated approach to Spring development. What if you want or need to configure Spring differently? In the next chapter, we'll look at how you can override Spring Boot auto-configuration as needed to achieve the goals of your application. You'll also see how to apply some of the same techniques to configure your own application components.

3 Customizing configuration

This chapter covers

- Overriding auto-configured beans
- Configuring with external properties
- Customizing error pages

Freedom of choice is an awesome thing. If you've ever ordered a pizza (who hasn't?) then you know that you have full control over what toppings are placed on the pie. If you ask for sausage, pepperoni, green peppers, and extra cheese, then you're essentially configuring the pizza to your precise specifications.

On the other hand, most pizza places also offer a form of auto-configuration. You can ask for the meat-lover's pizza, the vegetarian pizza, the spicy Italian pizza, or the ultimate example of pizza auto-configuration, the supreme pizza. When ordering one of these pizzas, you don't have to explicitly specify the toppings. The type of pizza ordered implies what toppings are used.

But what if you like all of the toppings of the supreme pizza, but also want jalapenos and would rather not have mushrooms? Does your taste for spicy food and aversion to fungus mean that auto-configuration isn't applicable and that you must explicitly configure your pizza? Absolutely not. Most pizzerias will let you customize your pizza, even if you started with a preconfigured option from the menu.

Working with traditional Spring configuration is much like ordering a pizza and explicitly specifying all of the toppings. You have full control over what goes into your Spring configuration, but explicitly declaring all of the beans in the application is non-optimal. On the other hand, Spring Boot auto-configuration is like ordering a specialty pizza from the menu. It's easier to let Spring Boot handle the details than to declare each and every bean in the application context.

Fortunately, Spring Boot auto-configuration is flexible. Like the pizzeria that will leave off the mushrooms and add jalapenos to your pizza, Spring Boot will let you step in and influence how it applies auto-configuration.

In this chapter, we're going to look at two ways to influence auto-configuration: explicit configuration overrides and fine-grained configuration with properties. We'll also look at how Spring Boot has provided hooks for you to plug in a custom error page.

3.1 Overriding Spring Boot auto-configuration

Generally speaking, if you can get the same results with no configuration as you would with explicit configuration, no configuration is the no-brainer choice. Why would you do extra work, writing and maintaining extra configuration code, if you can get what you need without it?

Most of the time, the auto-configured beans are exactly what you want and there's no need to override them. But there are some cases where the best guess that Spring Boot can make during auto-configuration probably isn't going to be good enough.

A prime example of a case where auto-configuration isn't good enough is when you're applying security to your application. Security is not one-size-fits-all, and there are decisions around application security that Spring Boot has no business making for you. Although Spring Boot provides some basic auto-configuration for security, you'll certainly want to override it to meet your specific security requirements.

To see how to override auto-configuration with explicit configuration, we'll start by adding Spring Security to the reading-list example. After seeing what you get for free with auto-configuration, we'll then override the basic security configuration to fit a particular situation.

3.1.1 Securing the application

Spring Boot auto-configuration makes securing an application a piece of cake. All you need to do is add the security starter to the build. For Gradle, the following dependency will do:

```
compile("org.springframework.boot:spring-boot-starter-security")
```

Or, if you're using Maven, add this `<dependency>` to your build's `<dependencies>` block:

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

That's it! Rebuild your application and run it. It's now a secure web application! The security starter adds Spring Security (among other things) to the application's classpath. With Spring Security on the classpath, auto-configuration kicks in and a very basic Spring Security setup is created.

If you try to open the application in your browser, you'll be immediately met with an HTTP Basic authentication dialog box. The username you'll need to enter is "user". As for the password, it's a bit trickier. The password is randomly generated and written to the logs each time the application is run. You'll need to look through the logging messages (written to `stdout` by default) and look for a line that looks something like this:

```
Using default security password: d9d8abe5-42b5-4f20-a32a-76ee3df658d9
```

I can't say for certain, but I'm guessing that this particular security setup probably isn't ideal for you. First, HTTP Basic dialog boxes are clunky and not very user-friendly. And I'll bet that you don't develop too many applications that have only one user who doesn't mind looking up their password from a log file. Therefore, you'll probably want to make a few changes to how Spring Security is configured. At very least, you'll want to provide a nice-looking login page and specify an authentication service that operates against a database or LDAP-based user store.

Let's see how to do that by writing some explicit Spring Security configuration to override the auto-configured security scheme.

3.1.2 Creating a custom security configuration

Overriding auto-configuration is a simple matter of explicitly writing the configuration as if auto-configuration didn't exist. This explicit configuration can take any form that Spring supports, including XML configuration and Groovy-based configuration.

For our purposes, we're going to focus on Java configuration when writing explicit configuration. In the case of Spring Security, this means writing a configuration class that extends `WebSecurityConfigurerAdapter`. `SecurityConfig` in listing 3.1 is the configuration class we'll use.

Listing 3.1 Explicit configuration to override auto-configured security

```
package readinglist;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
                                builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
                                                         HttpSecurity;
```

```
import org.springframework.security.config.annotation.web.configuration.
                                        EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.
                                        WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.
                                           UsernameNotFoundException;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Autowired
  private ReaderRepository readerRepository;

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
        .antMatchers("/").access("hasRole('READER')")       <- Require READER access
        .antMatchers("/**").permitAll()

      .and()

      .formLogin()
        .loginPage("/login")                   <- Set login form path
        .failureUrl("/login?error=true");
  }

  @Override
  protected void configure(
              AuthenticationManagerBuilder auth) throws Exception {
    auth
      .userDetailsService(new UserDetailsService() {     <- Define custom UserDetailsService
        @Override
        public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
          return readerRepository.findOne(username);
        }
      });
  }

}
```

SecurityConfig is a very basic Spring Security configuration. Even so, it does a lot of what we need to customize security of the reading-list application. By providing this custom security configuration class, we're asking Spring Boot to skip security auto-configuration and to use our security configuration instead.

Configuration classes that extend WebSecurityConfigurerAdapter can override two different configure() methods. In SecurityConfig, the first configure() method specifies that requests for "/" (which ReadingListController's methods are mapped to) require an authenticated user with the READER role. All other request paths are configured for open access to all users. It also designates /login as the path for the login page as well as the login failure page (along with an `error` attribute).

Spring Security offers several options for authentication, including authentication against JDBC-backed user stores, LDAP-backed user stores, and in-memory user stores. For our application, we're going to authenticate users against the database via JPA. The second `configure()` method sets this up by setting a custom user details service. This service can be any class that implements `UsersDetailsService` and is used to look up user details given a username. The following listing has given it an anonymous inner-class implementation that simply calls the `findOne()` method on an injected `ReaderRepository` (which is a Spring Data JPA repository interface).

Listing 3.2 A repository interface for persisting readers

```
package readinglist;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ReaderRepository
         extends JpaRepository<Reader, String> {
}
```

Persist readers via JPA

As with `BookRepository`, there's no need to write an implementation of `Reader-Repository`. Because it extends `JpaRepository`, Spring Data JPA will automatically create an implementation of it at runtime. This affords you 18 methods for working with `Reader` entities.

Speaking of `Reader` entities, the `Reader` class (shown in listing 3.3) is the final piece of the puzzle. It's a simple JPA entity type with a few fields to capture the username, password, and full name of the user.

Listing 3.3 A JPA entity that defines a `Reader`

```
package readinglist;
import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

@Entity
public class Reader implements UserDetails {

  private static final long serialVersionUID = 1L;

  @Id
  private String username;
  private String fullname;
  private String password;
```

Reader fields

```
  public String getUsername() {
    return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getFullname() {
    return fullname;
  }

  public void setFullname(String fullname) {
    this.fullname = fullname;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(String password) {
    this.password = password;
  }

  // UserDetails methods

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {     Grant READER privilege
    return Arrays.asList(new SimpleGrantedAuthority("READER"));
  }

  @Override
  public boolean isAccountNonExpired() {
    return true;
  }

  @Override
  public boolean isAccountNonLocked() {
    return true;
  }
                                                    Do not expire, lock, or disable
  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }

  @Override
  public boolean isEnabled() {
    return true;
  }

}
```

As you can see, `Reader` is annotated with `@Entity` to make it a JPA entity. In addition, its `username` field is annotated with `@Id` to designate it as the entity's ID. This seemed like a natural choice, as the `username` should uniquely identify the `Reader`.

You'll also notice that `Reader` implements the `UserDetails` interface and several of its methods. This makes it possible to use a `Reader` object to represent a user in Spring Security. The `getAuthorities()` method is overridden to always grant users READER authority. The `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentials-NonExpired()`, and `isEnabled()` methods are all implemented to return `true` so that the reader account is never expired, locked, or revoked.

Rebuild and restart the application and you should be able to log in to the application as one of the readers.

> **KEEPING IT SIMPLE** In a larger application, the authorities granted to a user might themselves be entities and be maintained in a separate database table. Likewise, the `boolean` values indicating whether an account is non-expired, non-locked, and enabled might be fields drawn from the database. For our purposes, however, I've decided to keep these details simple so as not to distract from what it is we're really discussing … namely, overriding Spring Boot auto-configuration.

There's a lot more we could do with regard to security configuration,[1] but this is all we need here, and it does demonstrate how to override the security auto-configuration provided by Spring Boot.

Again, all you need to do to override Spring Boot auto-configuration is to write explicit configuration. Spring Boot will see your configuration, step back, and let your configuration take precedence. To understand how this works, let's take a look under the covers of Spring Boot auto-configuration to see how it works and how it allows itself to be overridden.

3.1.3 Taking another peek under the covers of auto-configuration

As we discussed in section 2.3.3, Spring Boot auto-configuration comes with several configuration classes, any of which can be applied in your application. All of this configuration uses Spring 4.0's conditional configuration support to make runtime decisions as to whether or not Spring Boot's configuration should be used or ignored.

For the most part, the `@ConditionalOnMissingBean` annotation described in table 2.1 is what makes it possible to override auto-configuration. The `JdbcTemplate` bean defined in Spring Boot's `DataSourceAutoConfiguration` is a very simple example of how `@ConditionalOnMissingBean` works:

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

[1] For a deeper dive into Spring Security, have a look at chapters 9 and 14 of my *Spring in Action, Fourth Edition* (Manning, 2014).

The `jdbcTemplate()` method is annotated with `@Bean` and is ready to configure a `JdbcTemplate` bean if needed. But it's also annotated with `@ConditionalOnMissingBean`, which requires that there not already be a bean of type `JdbcOperations` (the interface that `JdbcTemplate` implements). If there's already a `JdbcOperations` bean, then the condition will fail and the `jdbcTemplate()` bean method will not be used.

What circumstances would result in there already being a `JdbcOperation` bean? Spring Boot is designed to load application-level configuration before considering its auto-configuration classes. Therefore, if you've already configured a `JdbcTemplate` bean, then there will be a bean of type `JdbcOperations` by the time that auto-configuration takes place, and the auto-configured `JdbcTemplate` bean will be ignored.

As it pertains to Spring Security, there are several configuration classes considered during auto-configuration. It would be impractical to go over each of them in detail here, but the one that's most significant in allowing us to override Spring Boot's auto-configured security configuration is `SpringBootWebSecurityConfiguration`. Here's an excerpt from that configuration class:

```
@Configuration
@EnableConfigurationProperties
@ConditionalOnClass({ EnableWebSecurity.class })
@ConditionalOnMissingBean(WebSecurityConfiguration.class)
@ConditionalOnWebApplication
public class SpringBootWebSecurityConfiguration {

...

}
```

As you can see, `SpringBootWebSecurityConfiguration` is annotated with a few conditional annotations. Per the `@ConditionalOnClass` annotation, the `@EnableWebSecurity` annotation must be available on the classpath. And per `@ConditionalOnWebApplication`, the application must be a web application. But it's the `@ConditionalOnMissingBean` annotation that makes it possible for our security configuration class to be used instead of `SpringBootWebSecurityConfiguration`.

The `@ConditionalOnMissingBean` requires that there not already be a bean of type `WebSecurityConfiguration`. Although it may not be apparent on the surface, by annotating our `SecurityConfig` class with `@EnableWebSecurity`, we're indirectly creating a bean of type `WebSecurityConfiguration`. Therefore, by the time auto-configuration takes place, there will already be a bean of type `WebSecurityConfiguration`, the `@ConditionalOnMissingBean` condition will fail, and any configuration offered by `SpringBootWebSecurityConfiguration` will be skipped over.

Although Spring Boot's auto-configuration and `@ConditionalOnMissingBean` make it possible for you to explicitly override any of the beans that would otherwise be auto-configured, it's not always necessary to go to that extreme. Let's see how you can set a few simple configuration properties to tweak the auto-configured components.

3.2 Externalizing configuration with properties

When dealing with application security, you'll almost certainly want to take full charge of the configuration. But it would be a shame to give up on auto-configuration just to tweak a small detail such as a server port number or a logging level. If you need to set a database URL, wouldn't it be easier to set a property somewhere than to completely declare a data source bean?

As it turns out, the beans that are automatically configured by Spring Boot offer well over 300 properties for fine-tuning. When you need to adjust the settings, you can specify these properties via environment variables, Java system properties, JNDI, command-line arguments, or property files.

To get started with these properties, let's look at a very simple example. You may have noticed that Spring Boot emits an ascii-art banner when you run the reading-list application from the command line. If you'd like to disable the banner, you can do so by setting a property named `spring.main.show-banner` to `false`. One way of doing that is to specify the property as a command-line parameter when you run the app:

```
$ java -jar readinglist-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

Another way is to create a file named application.properties that includes the following line:

```
spring.main.show-banner=false
```

Or, if you'd prefer, create a YAML file named application.yml that looks like this:

```
spring:
  main:
    show-banner: false
```

You could also set the property as an environment variable. For example, if you're using the bash or zsh shell, you can set it with the `export` command:

```
$ export spring_main_show_banner=false
```

Note the use of underscores instead of periods and dashes, as required for environment variable names.

There are, in fact, several ways to set properties for a Spring Boot application. Spring Boot will draw properties from several property sources, including the following:

1. Command-line arguments
2. JNDI attributes from java:comp/env
3. JVM system properties
4. Operating system environment variables
5. Randomly generated values for properties prefixed with `random.*` (referenced when setting other properties, such as `${random.long})
6. An application.properties or application.yml file outside of the application

7. An application.properties or application.yml file packaged inside of the application
8. Property sources specified by `@PropertySource`
9. Default properties

This list is in order of precedence. That is, any property set from a source higher in the list will override the same property set on a source lower in the list. Command-line arguments, for instance, override properties from any other property source.

As for the application.properties and application.yml files, they can reside in any of four locations:

1. Externally, in a /config subdirectory of the directory from which the application is run
2. Externally, in the directory from which the application is run
3. Internally, in a package named "config"
4. Internally, at the root of the classpath

Again, this list is in order of precedence. That is, an application.properties file in a /config subdirectory will override the same properties set in an application.properties file in the application's classpath.

Also, I've found that if you have both application.properties and application.yml side by side at the same level of precedence, properties in application.yml will override those in application.properties.

Disabling an ascii-art banner is just a small example of how to use properties. Let's look at a few more common ways to tweak the auto-configured beans.

3.2.1 Fine-tuning auto-configuration

As I said, there are well over 300 properties that you can set to tweak and adjust the beans in a Spring Boot application. Appendix C gives an exhaustive list of these properties, but it'd be impossible to go over each and every one of them here. Instead, let's examine a few of the more commonly useful properties exposed by Spring Boot.

DISABLING TEMPLATE CACHING

If you've been tinkering around much with the reading-list application, you may have noticed that changes to any of the Thymeleaf templates aren't applied unless you restart the application. That's because Thymeleaf templates are cached by default. This improves application performance because you only compile the templates once, but it's difficult to make changes on the fly during development.

You can disable Thymeleaf template caching by setting `spring.thymeleaf.cache` to `false`. You can do this when you run the application from the command line by setting it as a command-line argument:

```
$ java -jar readinglist-0.0.1-SNAPSHOT.jar --spring.thymeleaf.cache=false
```

Or, if you'd rather have caching turned off every time you run the application, you might create an application.yml file with the following lines:

```
spring:
  thymeleaf:
    cache: false
```

You'll want to make sure that this application.yml file doesn't follow the application into production, or else your production application won't realize the performance benefits of template caching.

As a developer, you may find it convenient to have template caching turned off all of the time while you make changes to the templates. In that case, you can turn off Thymeleaf caching via an environment variable:

```
$ export spring_thymeleaf_cache=false
```

Even though we're using Thymeleaf for our application's views, template caching can be turned off for Spring Boot's other supported template options by setting these properties:

- `spring.freemarker.cache` (Freemarker)
- `spring.groovy.template.cache` (Groovy templates)
- `spring.velocity.cache` (Velocity)

By default, all of these properties are `true`, meaning that the templates are cached. Setting them to `false` disables caching.

CONFIGURING THE EMBEDDED SERVER

When you run a Spring Boot application from the command line (or via Spring Tool Suite), the application starts an embedded server (Tomcat, by default) listening on port 8080. This is fine for most cases, but it can become problematic if you find yourself needing to run multiple applications simultaneously. If all of the applications try to start a Tomcat server on the same port, there'll be port collisions starting with the second application.

If, for any reason, you'd rather the server listen on a different port, then all you need to do is set the `server.port` property. If this is a one-time change, it's easy enough to do this as a command-line argument:

```
$ java -jar readinglist-0.0.1-SNAPSHOT.jar --server.port=8000
```

But if you want the port change to be more permanent, you could set `server.port` in one of the other supported locations. For instance, you might set it in an application.yml file at the root of the application's classpath:

```
server:
  port: 8000
```

Aside from adjusting the server's port, you might also need to enable the server to serve securely over HTTPS. The first thing you'll need to do is create a keystore using the JDK's `keytool` utility:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

You'll be asked several questions about your name and organization, most of which are irrelevant. But when asked for a password, be sure to remember what you choose. For the sake of this example, I chose "letmein" as the password.

Now you just need to set a few properties to enable HTTPS in the embedded server. You could specify them all at the command line, but that would be terribly inconvenient. Instead, you'll probably set them in application.properties or application.yml. In application.yml, they might look like this:

```
server:
  port: 8443
  ssl:
    key-store: file:///path/to/mykeys.jks
    key-store-password: letmein
    key-password: letmein
```

Here the `server.port` property is being set to 8443, a common choice for development HTTPS servers. The `server.ssl.key-store` property should be set to the path where the keystore file was created. Here it's shown with a file:// URL to load it from the filesystem, but if you package it within the application JAR file, you should use a classpath: URL to reference it. And both the `server.ssl.key-store-password` and `server.ssl.key-password` properties are set to the password that was given when creating the keystore.

With these properties in place, your application should be listening for HTTPS requests on port 8443. (Depending on which browser you're using, you may encounter a warning about the server not being able to verify its identity. This is nothing to worry about when serving from localhost during development.)

CONFIGURING LOGGING

Most applications provide some form of logging. And even if your application doesn't log anything directly, the libraries that your application uses will certainly log their activity.

By default, Spring Boot configures logging via Logback (http://logback.qos.ch) to log to the console at INFO level. You've probably already seen plenty of INFO-level logging as you've run the application and other examples.

Swapping out Logback for another logging implementation

Generally speaking, you should never need to switch logging implementations; Logback should suit you fine. However, if you decide that you'd rather use Log4j or Log4j2, you'll need to change your dependencies to include the appropriate starter for the logging implementation you want to use and to exclude Logback.

(continued)
For Maven builds, you can exclude Logback by excluding the default logging starter transitively resolved by the root starter dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

In Gradle, it's easiest to place the exclusion under the `configurations` section:

```
configurations {
  all*.exclude group:'org.springframework.boot',
               module:'spring-boot-starter-logging'
}
```

With the default logging starter excluded, you can now include the starter for the logging implementation you'd rather use. With a Maven build you can add Log4j like this:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

In a Gradle build you can add Log4j like this:

```
compile("org.springframework.boot:spring-boot-starter-log4j")
```

If you'd rather use Log4j2, change the artifact from "spring-boot-starter-log4j" to "spring-boot-starter-log4j2".

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here's an example of a simple logback.xml file you might use:

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>

  <logger name="root" level="INFO"/>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Aside from the pattern used for logging, this Logback configuration is more or less equivalent to the default you'll get if you have no logback.xml file. But by editing logback.xml you can gain full control over your application's log files. The specifics of what can go into logback.xml are outside the scope of this book, so refer to Logback's documentation for more information.

Even so, the most common changes you'll make to a logging configuration are to change the logging levels and perhaps to specify a file where the logs should be written. With Spring Boot configuration properties, you can make those changes without having to create a logback.xml file.

To set the logging levels, you create properties that are prefixed with `logging.level`, followed by the name of the logger for which you want to set the logging level. For instance, suppose you'd like to set the root logging level to WARN, but log Spring Security logs at DEBUG level. The following entries in application.yml will take care of it for you:

```
logging:
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Optionally, you can collapse the Spring Security package name to a single line:

```
logging:
  level:
    root: WARN
    org.springframework.security: DEBUG
```

Now suppose that you want to write the log entries to a file named BookWorm.log at /var/logs/. The `logging.path` and `logging.file` properties can help with that:

```
logging:
  path: /var/logs/
  file: BookWorm.log
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Assuming that the application has write permissions to /var/logs/, the log entries will be written to /var/logs/BookWorm.log. By default, the log files will rotate once they hit 10 megabytes in size.

Similarly, all of these properties can be set in application.properties like this:

```
logging.path=/var/logs/
logging.file=BookWorm.log
logging.level.root=WARN
logging.level.root.org.springframework.security=DEBUG
```

If you still need full control of the logging configuration, but would rather name the Logback configuration file something other than logback.xml, you can specify a custom name by setting the `logging.config` property:

```
logging:
  config:
    classpath:logging-config.xml
```

Although you usually won't need to change the configuration file's name, it can come in handy if you want to use two different logging configurations for different runtime profiles (see section 3.2.3).

CONFIGURING A DATA SOURCE

At this point, we're still developing our reading-list application. As such, the embedded H2 database we're using is perfect for our needs. But once we take the application into production, we may want to consider a more permanent database solution.

Although you could explicitly configure your own `DataSource` bean, it's usually not necessary. Instead, simply configure the URL and credentials for your database via properties. For example, if you're using a MySQL database, your application.yml file might look like this:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/readinglist
    username: dbuser
    password: dbpass
```

You usually won't need to specify the JDBC driver; Spring Boot can figure it out from the database URL. But if there is a problem, you can try setting the `spring.datasource.driver-class-name` property:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/readinglist
    username: dbuser
    password: dbpass
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot will use this connection data when auto-configuring the `DataSource` bean. The `DataSource` bean will be pooled, using Tomcat's pooling `DataSource` if it's available on the classpath. If not, it will look for and use one of these other connection pool implementations on the classpath:

- HikariCP
- Commons DBCP
- Commons DBCP 2

Although these are the only connection pool options available through auto-configuration, you are always welcome to explicitly configure a `DataSource` bean to use whatever connection pool implementation you'd like.

You may also choose to look up the `DataSource` from JNDI by setting the `spring.datasource.jndi-name` property:

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/readingListDS
```

If you set the `spring.datasource.jndi-name` property, the other datasource connection properties (if set) will be ignored.

There are many ways to influence the components that Spring Boot auto-configures by just setting a property or two. But this style of externalized configuration is not limited to the beans configured by Spring Boot. Let's look at how you can use the very same property configuration mechanism to fine-tune your own application components.

3.2.2 Externally configuring application beans

Suppose that we wanted to show not just the title of a book on someone's reading list, but also provide a link to the book on Amazon.com. And, not only do we want to provide a link to the book, but we also want to tag the book to take advantage of Amazon's associate program so that if anyone purchases a book through one of the links in our application, we'd receive a small payment for the referral.

This is simple enough to do by changing the Thymeleaf template to render the title of each book as a link:

```
<a th:href="'http://www.amazon.com/gp/product/'
            + ${book.isbn}
            + '/tag=habuma-20'"
   th:text="${book.title}">Title</a>
```

This will work perfectly. Now if anyone clicks on the link and buys the book, *I* will get credit for the referral. That's because "habuma-20" is *my* Amazon Associate ID. If you'd rather receive credit, you can easily change the value of the `tag` attribute to your Amazon Associate ID in the Thymeleaf template.

Even though it's easy enough to change the Amazon Associate ID in the template, it's still hard-coded. We're only linking to Amazon from this one template, but we may later add features to the application where we link to Amazon from several pages. In that case, changes to the Amazon Associate ID would require changes to several places in the application code. That's why details like this are often better kept out of the code so that they can be managed in a single place.

Rather than hard-code the Amazon Associate ID in the template, we can refer to it as a value in the model:

```
<a th:href="'http://www.amazon.com/gp/product/'
            + ${book.isbn}
            + '/tag=' + ${amazonID}"
   th:text="${book.title}">Title</a>
```

In addition, `ReadingListController` will need to populate the model at the key "amazonID" to contain the Amazon Associate ID. Again, we shouldn't hard-code it, but instead refer to an instance variable. And that instance variable should be populated from the property configuration. Listing 3.4 shows the new `ReadingListController`, which populates the model from an injected Amazon Associate ID.

Listing 3.4 `ReadingListController` modified to accept an Amazon ID

```
package readinglist;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")
@ConfigurationProperties(prefix="amazon")                    <-- Inject with properties
public class ReadingListController {

  private String associateId;

  private ReadingListRepository readingListRepository;

  @Autowired
  public ReadingListController(
        ReadingListRepository readingListRepository) {
    this.readingListRepository = readingListRepository;
  }

  public void setAssociateId(String associateId) {           <-- Setter method for associateId
    this.associateId = associateId;
  }

  @RequestMapping(method=RequestMethod.GET)
  public String readersBooks(Reader reader, Model model) {
    List<Book> readingList =
              readingListRepository.findByReader(reader);
    if (readingList != null) {
      model.addAttribute("books", readingList);
      model.addAttribute("reader", reader);
      model.addAttribute("amazonID", associateId);     <-- Put associateId into model
    }
    return "readingList";
  }

  @RequestMapping(method=RequestMethod.POST)
  public String addToReadingList(Reader reader, Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/";
  }

}
```

As you can see, the `ReadingListController` now has an `associateId` property and a corresponding `setAssociateId()` method through which the property can be set. And `readersBooks()` now adds the value of `associateId` to the model under the key "amazonID".

Perfect! Now the only question is where `associateId` gets its value.

Notice that `ReadingListController` is now annotated with `@ConfigurationProperties`. This specifies that this bean should have its properties injected (via setter methods) with values from configuration properties. More specifically, the `prefix` attribute specifies that the `ReadingListController` bean will be injected with properties with an "amazon" prefix.

Putting this all together, we've specified that `ReadingListController` should have its properties injected from "amazon"-prefixed configuration properties. `ReadingListController` has only one property with a setter method—the `associateId` property. Therefore, all we need to do to specify the Amazon Associate ID is to add an `amazon.associateId` property in one of the supported property source locations.

For example, we could set that property in application.properties:

```
amazon.associateId=habuma-20
```

Or in application.yml:

```
amazon:
  associateId: habuma-20
```

Or we could set it as an environment variable, specify it as a command-line argument, or add it in any of the other places where configuration properties can be set.

ENABLING CONFIGURATION PROPERTIES Technically, the `@ConfigurationProperties` annotation won't work unless you've enabled it by adding `@EnableConfigurationProperties` in one of your Spring configuration classes. This is often unnecessary, however, because all of the configuration classes behind Spring Boot auto-configuration are already annotated with `@EnableConfigurationProperties`. Therefore, unless you aren't taking advantage of auto-configuration at all (and why would that ever happen?), you shouldn't need to explicitly use `@EnableConfigurationProperties`.

It's also worth noting that Spring Boot's property resolver is clever enough to treat camel-cased properties as interchangeable with similarly named properties with hyphens or underscores. In other words, a property named `amazon.associateId` is equivalent to both `amazon.associate_id` and `amazon.associate-id`. Feel free to use the naming convention that suits you best.

COLLECTING PROPERTIES IN ONE CLASS

Although annotating `ReadingListController` with `@ConfigurationProperties` works fine, it may not be ideal. Doesn't it seem a little odd that the property prefix is "amazon" when, in fact, `ReadingListController` has little to do with Amazon? Moreover, future enhancements might present the need to configure properties unrelated to Amazon in `ReadingListController`.

Instead of capturing the configuration properties in `ReadingListController`, it may be better to annotate a separate bean with `@ConfigurationProperties` and let that bean collect all of the configuration properties. `AmazonProperties` in listing 3.5, for example, captures the Amazon-specific configuration properties.

Listing 3.5 Capturing configuration properties in a bean

```
package readinglist;

import org.springframework.boot.context.properties.
                                   ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties("amazon")            <-- Inject with "amazon"-
public class AmazonProperties {                   prefixed properties

  private String associateId;

  public void setAssociateId(String associateId) {   <-- associateId
    this.associateId = associateId;                      setter method
  }

  public String getAssociateId() {
    return associateId;
  }

}
```

With `AmazonProperties` capturing the `amazon.associateId` configuration property, we can change `ReadingListController` (as shown in listing 3.6) to pull the Amazon Associate ID from an injected `AmazonProperties`.

Listing 3.6 `ReadingListController` injected with `AmazonProperties`

```
package readinglist;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")
public class ReadingListController {

  private ReadingListRepository readingListRepository;
  private AmazonProperties amazonProperties;

  @Autowired
  public ReadingListController(
      ReadingListRepository readingListRepository,
      AmazonProperties amazonProperties) {          <-- Inject AmazonProperties
    this.readingListRepository = readingListRepository;
    this.amazonProperties = amazonProperties;
  }

  @RequestMapping(method=RequestMethod.GET)
  public String readersBooks(Reader reader, Model model) {
    List<Book> readingList =
        readingListRepository.findByReader(reader);
    if (readingList != null) {
      model.addAttribute("books", readingList);
      model.addAttribute("reader", reader);
      model.addAttribute("amazonID", amazonProperties.getAssociateId());   <-- Add Associate ID to model
    }
    return "readingList";
  }

  @RequestMapping(method=RequestMethod.POST)
  public String addToReadingList(Reader reader, Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/";
  }

}
```

`ReadingListController` is no longer the direct recipient of configuration properties. Instead, it obtains the information it needs from the injected `AmazonProperties` bean.

As we've seen, configuration properties are useful for tweaking both auto-configured components as well as the details injected into our own application beans. But what if we need to configure different properties for different deployment environments? Let's take a look at how to use Spring profiles to set up environment-specific configuration.

3.2.3 Configuring with profiles

When applications are deployed to different runtime environments, there are usually some configuration details that will differ. The details of a database connection, for instance, are likely different in a development environment than in a quality assurance environment, and different still in a production environment. The Spring Framework introduced support for profile-based configuration in Spring 3.1. Profiles are a type of conditional configuration where different beans or configuration classes are used or ignored based on what profiles are active at runtime.

For instance, suppose that the security configuration we created in listing 3.1 is for production purposes, but the auto-configured security configuration is fine for development. In that case, we can annotate `SecurityConfig` with `@Profile` like this:

```
@Profile("production")
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

...

}
```

The `@Profile` annotation used here requires that the "production" profile be active at runtime for this configuration to be applied. If the "production" profile isn't active, this configuration will be ignored and, for lack of another overriding security configuration, the auto-configured security configuration will be applied.

Profiles can be activated by setting the `spring.profiles.active` property using any of the means available for setting any other configuration property. For example, you could activate the "production" profile by running the application at the command line like this:

```
$ java -jar readinglist-0.0.1-SNAPSHOT.jar --
     spring.profiles.active=production
```

Or you can add the `spring.profiles.active` property to application.yml:

```
spring:
  profiles:
    active: production
```

Or you could set an environment variable and put it in application.properties or use any of the other options mentioned at the beginning of section 3.2.

But because Spring Boot auto-configures so much for you, it would be very inconvenient to write explicit configuration just so that you can have a place to put `@Profile`.

Fortunately, Spring Boot supports profiles for properties set in application.properties and application.yml.

To demonstrate profiled properties, suppose that you want a different logging configuration in production than in development. In production, you're only interested in log entries at WARN level or higher, and you want to write the log entries to a log file. In development, however, you only want things logged to the console and at DEBUG level or higher.

All you need to do is create separate configurations for each environment. How you do that, however, depends on whether you're using a properties file configuration or YAML configuration.

WORKING WITH PROFILE-SPECIFIC PROPERTIES FILES

If you're using application.properties to express configuration properties, you can provide profile-specific properties by creating additional properties files named with the pattern "application-{profile}.properties".

For the logging scenario, the development configuration would be in a file named application-development.properties and contain properties for verbose, console-written logging:

```
logging.level.root=DEBUG
```

But for production, application-production.properties would configure logging to be at WARN level and higher and to write to a log file:

```
logging.path=/var/logs/
logging.file=BookWorm.log
logging.level.root=WARN
```

Meanwhile, any properties that aren't specific to any profile or that serve as defaults (in case a profile-specific configuration doesn't specify otherwise) can continue to be expressed in application.properties:

```
amazon.associateId=habuma-20
logging.level.root=INFO
```

CONFIGURING WITH MULTI-PROFILE YAML FILES

If you're using YAML for configuration properties, you can follow a similar naming convention as for properties files. That is, you can create YAML files whose names follow a pattern of "application-{profile}.yml" and continue to put non-profiled properties in application.yml.

But with YAML, you also have the option of expressing configuration properties for all profiles in a single application.yml file. For example, the logging configuration we want can be declared in application.yml like this:

```
logging:
  level:
    root: INFO
```

```
---

spring:
  profiles: development

logging:
  level:
    root: DEBUG

---

spring:
  profiles: production

logging:
  path: /tmp/
  file: BookWorm.log
  level:
    root: WARN
```

As you can see, this application.yml file is divided into three sections by a set of triple hyphens (`---`). The second and third sections each specify a value for `spring.profiles`. This property indicates which profile each section's properties apply to. The properties defined in the middle section apply to development because it sets `spring.profiles` to "development". Similarly, the last section has `spring.profiles` set to "production", making it applicable when the "production" profile is active.

The first section, on the other hand, doesn't specify a value for `spring.profiles`. Therefore, its properties are common to all profiles or are defaults if the active profile doesn't otherwise have the properties set.

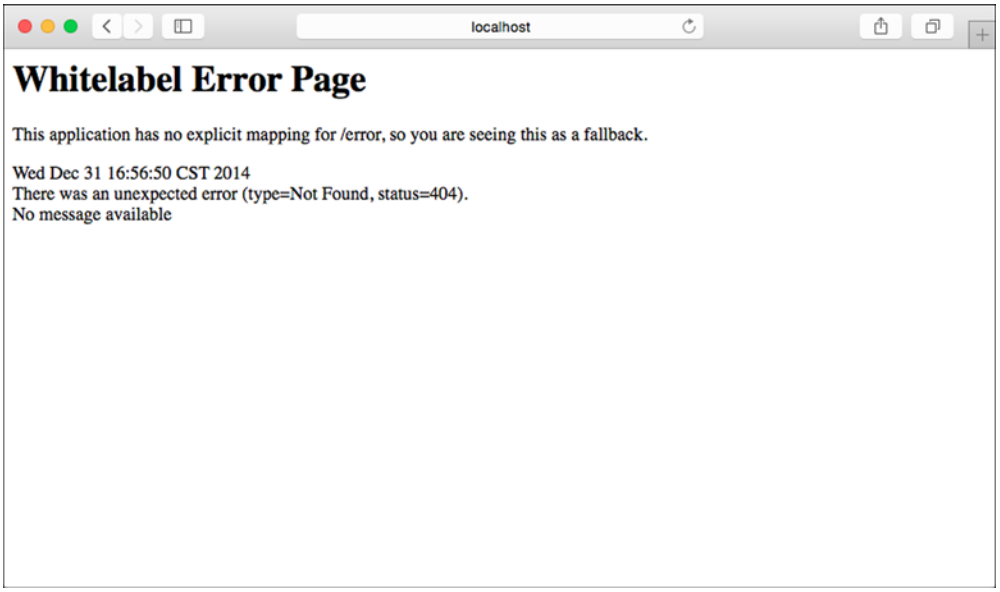
Aside from auto-configuration and external configuration properties, Spring Boot has one other trick up its sleeve to simplify a common development task: it automatically configures a page to be displayed when an application encounters any errors. To wrap up this chapter, we'll take a look at Spring Boot's error page and see how to customize it to fit our application.

3.3 Customizing application error pages

Errors happen. Even some of the most robust applications running in production occasionally run into trouble. Although it's important to reduce the chance that a user will encounter an error, it's also important that your application still present itself well when displaying an error page.

In recent years, creative error pages have become an art form. If you've ever seen the Star Wars–inspired error page at GitHub.com or DropBox.com's Escher-like error page, you have an idea of what I'm talking about.

I don't know if you've encountered any errors while trying out the reading-list application, but if so you've probably seen an error page much like the one in figure 3.1.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Dec 31 16:56:50 CST 2014
There was an unexpected error (type=Not Found, status=404).
No message available

Figure 3.1 Spring Boot's default whitelabel error page.

Spring Boot offers this "whitelabel" error page by default as part of auto-configuration. Even though it's slightly more attractive than a stack trace, it doesn't compare with some of the great works of error art available on the internet. In the interest of presenting your application failures as masterpieces, you'll probably want to create a custom error page for your applications.

The default error handler that's auto-configured by Spring Boot looks for a view whose name is "error". If it can't find one, it uses its default whitelabel error view shown in figure 3.1. Therefore, the easiest way to customize the error page is to create a custom view that will resolve for a view named "error".

Ultimately this depends on the view resolvers in place when the error view is being resolved. This includes

- Any bean that implements Spring's `View` interface and has a bean ID of "error" (resolved by Spring's `BeanNameViewResolver`)
- A Thymeleaf template named "error.html" if Thymeleaf is configured
- A FreeMarker template named "error.ftl" if FreeMarker is configured
- A Velocity template named "error.vm" if Velocity is configured
- A JSP template named "error.jsp" if using JSP views

Because we're using Thymeleaf for the reading-list application, all we must do to customize the error page is create a file named "error.html" and place it in the templates folder along with our other application templates. Listing 3.7 shows a simple, yet effective replacement for the default whitelabel error page.

Listing 3.7 Custom error page for the reading-list application

```
<html>
  <head>
    <title>Oops!</title>
    <link rel="stylesheet" th:href="@{/style.css}"></link>
  </head>

  <html>
    <div class="errorPage">
      <span class="oops">Oops!</span><br/>
      <img th:src="@{/MissingPage.png}"></img>
      <p>There seems to be a problem with the page you requested
         (<span th:text="${path}"></span>).</p>                    <-- Show requested path

      <p th:text="${'Details: ' + message}"></p>       <-- Show error details
    </div>
  </html>

</html>
```

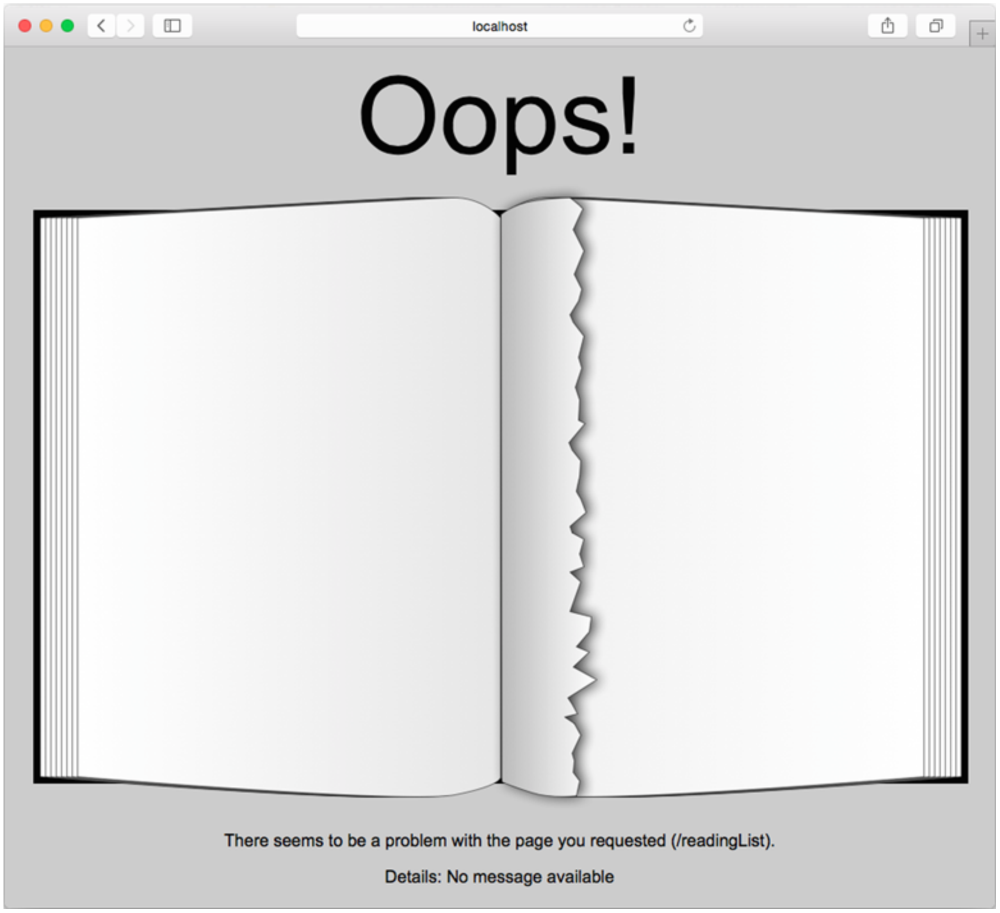
This custom error template should be named "error.html" and placed in the templates directory for the Thymeleaf template resolver to find. For a typical Maven or Gradle build, that means putting it in src/main/resources/templates so that it's at the root of the classpath during runtime.

For the most part, this is a simple Thymeleaf template that displays an image and some error text. There are two specific pieces of information that it also renders: the request path of the error and the exception message. These aren't the only details available to an error page, however. By default, Spring Boot makes the following error attributes available to the error view:

- `timestamp`—The time that the error occurred
- `status`—The HTTP status code
- `error`—The error reason
- `exception`—The class name of the exception
- `message`—The exception message (if the error was caused by an exception)
- `errors`—Any errors from a `BindingResult` exception (if the error was caused by an exception)
- `trace`—The exception stack trace (if the error was caused by an exception)
- `path`—The URL path requested when the error occurred

Some of these attributes, such as `path`, are useful when communicating the problem to the user. Others, such as `trace`, should be used sparingly, be hidden, or be used cleverly on the error page to keep the error page as user-friendly as possible.

You'll also notice that the template references an image named MissingPage.png. The actual content of the image is unimportant, so feel free to flex your graphic design muscles and come up with an image that suits you. But be sure to put it in src/main/resources/static or src/main/resources/public so that it can be served when the application is running.



There seems to be a problem with the page you requested (/readingList).

Details: No message available

Figure 3.2 A custom error page exhibits style in the face of failure

Figure 3.2 shows what the user will see when an error occurs. It may not quite be a work of art, but I think it raises the aesthetics of the application's error page a notch or two.

3.4 Summary

Spring Boot eliminates much of the boilerplate configuration that's often required in Spring applications. But by letting Spring Boot do all of the configuration, you're relying on it to configure components in ways that suit your application. When auto-configuration doesn't fit your needs, Spring Boot allows you to override and fine-tune the configuration it provides.

Overriding auto-configuration is a simple matter of writing explicit Spring configuration as you would in the absence of Spring Boot. Spring Boot's auto-configuration is designed to favor application-provided configuration over its own auto-configuration.

Even when auto-configuration is suitable, you may need to adjust a few details. Spring Boot enables several property resolvers that let you tweak configuration by setting properties as environment variables, in properties files, in YAML files, and in several other ways. This same property-based configuration model can even be applied to application-defined components, enabling value-injection into bean properties from external configuration sources.

Spring Boot also auto-configures a simple whitelabel error page. Although it's more user-friendly than an exception and stack trace, the whitelabel error page still leaves a lot to be desired aesthetically. Fortunately, Spring Boot offers several options for customizing or completely replacing the whitelabel error page to suit an application's specific style.

Now that we've written a complete application with Spring Boot, we should verify that it actually does what we expect it to do. That is, instead of poking at it in the web browser manually, we should write some automated and repeatable tests that exercise the application and prove that it's working correctly. That's exactly what we'll do in the next chapter.

4 Testing with Spring Boot

This chapter covers

- Integration testing
- Testing apps in a server
- Spring Boot's test utilities

It's been said that if you don't know where you're going, any road will get you there. But with software development, if you don't know where you're going, you'll likely end up with a buggy application that nobody can use.

The best way to know for sure where you're going when writing applications is to write tests that assert the desired behavior of an application. If those tests fail, you know you have some work to do. If they pass, then you've arrived (at least until you think of some more tests that you can write).

Whether you write tests first or after the code has already been written, it's important that you write tests to not only verify the accuracy of your code, but to also to make sure it does everything you expect it to. Tests are also a great safeguard to make sure that things don't break as your application continues to evolve.

When it comes to writing unit tests, Spring is generally out of the picture. Loose coupling and interface-driven design, which Spring encourages, makes it really easy to write unit tests. But Spring isn't necessarily involved in those unit tests.

Integration tests, on the other hand, require some help from Spring. If Spring is responsible for configuring and wiring up the components in your production application, then Spring should also be responsible for configuring and wiring up those components in your tests.

Spring's `SpringJUnit4ClassRunner` helps load a Spring application context in JUnit-based application tests. Spring Boot builds on Spring's integration testing support by enabling auto-configuration and web server startup when testing Spring Boot applications. It also offers a handful of useful testing utilities.

In this chapter, we'll look at all of the ways that Spring Boot supports integration testing. We'll start by looking at how to test with a fully Spring Boot-enabled application context.

4.1 Integration testing auto-configuration

At the core of everything that the Spring Framework does, its most essential task is to wire together all of the components that make up an application. It does this by reading a wiring specification (whether it be XML, Java-based, Groovy-based, or otherwise), instantiating beans in an application context, and injecting beans into other beans that depend on them.

When integration testing a Spring application, it's important to let Spring wire up the beans that are the target of the test the same way it wires up those beans when the application is running in production. Sure, you might be able to manually instantiate the components and inject them into each other, but for any substantially big application, that can be an arduous task. Moreover, Spring offers additional facilities such as component-scanning, autowiring, and declarative aspects such as caching, transactions, and security. Given all that would be required to recreate what Spring does, it's generally best to let Spring do the heavy lifting, even in an integration test.

Spring has offered excellent support for integration testing since version 1.1.1. Since Spring 2.5, integration testing support has been offered in the form of `SpringJUnit4ClassRunner`, a JUnit class runner that loads a Spring application context for use in a JUnit test and enables autowiring of beans into the test class.

For example, consider the following listing, which shows a very basic Spring integration test.

Listing 4.1 Integration testing Spring with `SpringJUnit4ClassRunner`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
      classes=AddressBookConfiguration.class)     <- Loads application context
public class AddressServiceTests {

  @Autowired
  private AddressService addressService;          <- Injects address service

  @Test
  public void testService() {                                         Tests address
    Address address = addressService.findByLastName("Sheman");        service
    assertEquals("P", address.getFirstName());
    assertEquals("Sherman", address.getLastName());
    assertEquals("42 Wallaby Way", address.getAddressLine1());
    assertEquals("Sydney", address.getCity());
    assertEquals("New South Wales", address.getState());
    assertEquals("2000", address.getPostCode());
  }

}
```

As you can see, `AddressServiceTests` is annotated with both `@RunWith` and `@ContextConfiguration`. `@RunWith` is given `SpringJUnit4ClassRunner.class` to enable Spring integration testing.[1] Meanwhile, `@ContextConfiguration` specifies how to load the application context. Here we're asking it to load the Spring application context given the specification defined in `AddressBookConfiguration`.

In addition to loading the application context, `SpringJUnit4ClassRunner` also makes it possible to inject beans from the application context into the test itself via autowiring. Because this test is targeting an `AddressService` bean, it is autowired into the test. Finally, the `testService()` method makes calls to the address service and verifies the results.

Although `@ContextConfiguration` does a great job of loading the Spring application context, it doesn't load it with the full Spring Boot treatment. Spring Boot applications are ultimately loaded by `SpringApplication`, either explicitly (as in listing 2.1) or using `SpringBootServletInitializer` (which we'll look at in chapter 8). `SpringApplication` not only loads the application context, but also enables logging, the loading of external properties (application.properties or application.yml), and other features of Spring Boot. If you're using `@ContextConfiguration`, you won't get those features.

To get those features back in your integration tests, you can swap out `@ContextConfiguration` for Spring Boot's `@SpringApplicationConfiguration`:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
      classes=AddressBookConfiguration.class)
public class AddressServiceTests {
  ...
}
```

The use of `@SpringApplicationConfiguration` is largely identical to `@ContextConfiguration`. But unlike `@ContextConfiguration`, `@SpringApplicationConfiguration` loads the Spring application context using `SpringApplication` the same way and with the same treatment it would get if it was being loaded in a production application. This includes the loading of external properties and Spring Boot logging.

[1] As of Spring 4.2, you can optionally use `SpringClassRule` and `SpringMethodRule` as JUnit rule-based alternatives to `SpringJUnit4ClassRunner`.

Suffice it to say that, for the most part, `@SpringApplicationConfiguration` replaces `@ContextConfiguration` when writing tests for Spring Boot applications. We'll certainly use `@SpringApplicationConfiguration` throughout this chapter as we write tests for our Spring Boot application, including tests that target the web front end of the application.

Speaking of web testing, that's what we're going to do next.

4.2 Testing web applications

One of the nice things about Spring MVC is that it promotes a programming model around plain old Java objects (POJOs) that are annotated to declare how they should process web requests. This programming model is not only simple, it enables you to treat controllers just as you would any other component in your application. You might even be tempted to write tests against your controller that test them as POJOs.

For instance, consider the `addToReadingList()` method from `ReadingListController`:

```
@RequestMapping(method=RequestMethod.POST)
public String addToReadingList(Book book) {
  book.setReader(reader);
  readingListRepository.save(book);
  return "redirect:/readingList";
}
```

If you were to disregard the `@RequestMapping` method, you'd be left with a rather basic Java method. It wouldn't take much to imagine a test that provides a mock implementation of `ReadingListRepository`, calls `addToReadingList()` directly, and asserts the return value and verifies the call to the repository's `save()` method.

The problem with such a test is that it only tests the method itself. While that's better than no test at all, it fails to test that the method handles a `POST` request to /readingList. It also fails to test that form fields are properly bound to the `Book` parameter. And although you could assert that the returned `String` contains a certain value, it would be impossible to test definitively that the request is, in fact, redirected to /readingList after the method is finished.

To properly test a web application, you need a way to throw actual HTTP requests at it and assert that it processes those requests correctly. Fortunately, there are two options available to Spring Boot application developers that make those kinds of tests possible:

- *Spring Mock MVC*—Enables controllers to be tested in a mocked approximation of a servlet container without actually starting an application server
- *Web integration tests*—Actually starts the application in an embedded servlet container (such as Tomcat or Jetty), enabling tests that exercise the application in a real application server

Each of these kinds of tests has its share of pros and cons. Obviously, starting a server will result in a slower test than mocking a servlet container. But there's no doubt that server-based tests are closer to the real-world environment that they'll be running in when deployed to production.

We're going to start by looking at how you can test a web application using Spring's Mock MVC test framework. Then, in section 4.3, you'll see how to write tests against an application that's actually running in an application server.

4.2.1 Mocking Spring MVC

Since Spring 3.2, the Spring Framework has had a very useful facility for testing web applications by mocking Spring MVC. This makes it possible to perform HTTP requests against a controller without running the controller within an actual servlet container. Instead, Spring's Mock MVC framework mocks enough of Spring MVC to make it almost as though the application is running within a servlet container … but it's not.

To set up a Mock MVC in your test, you can use `MockMvcBuilders`. This class offers two static methods:

- `standaloneSetup()`—Builds a Mock MVC to serve one or more manually created and configured controllers
- `webAppContextSetup()`—Builds a Mock MVC using a Spring application context, which presumably includes one or more configured controllers

The primary difference between these two options is that `standaloneSetup()` expects you to manually instantiate and inject the controllers you want to test, whereas `webAppContextSetup()` works from an instance of `WebApplicationContext`, which itself was probably loaded by Spring. The former is slightly more akin to a unit test in that you'll likely only use it for very focused tests around a single controller. The latter, however, lets Spring load your controllers as well as their dependencies for a full-blown integration test.

For our purposes, we're going to use `webAppContextSetup()` so that we can test the `ReadingListController` as it has been instantiated and injected from the application context that Spring Boot has auto-configured.

The `webAppContextSetup()` takes a `WebApplicationContext` as an argument. Therefore, we'll need to annotate the test class with `@WebAppConfiguration` and use `@Autowired` to inject the `WebApplicationContext` into the test as an instance variable. The following listing shows the starting point for our Mock MVC tests.

Listing 4.2 Creating a Mock MVC for integration testing controllers

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
      classes = ReadingListApplication.class)
@WebAppConfiguration                              <-- Enables web context testing
public class MockMvcWebTests {
  @Autowired
  private WebApplicationContext webContext;       <-- Injects
                                                      WebApplicationContext
  private MockMvc mockMvc;

  @Before
  public void setupMockMvc() {
    mockMvc = MockMvcBuilders                  <-- Sets up
        .webAppContextSetup(webContext)            MockMvc
        .build();
  }

}
```

The `@WebAppConfiguration` annotation declares that the application context created by `SpringJUnit4ClassRunner` should be a `WebApplicationContext` (as opposed to a basic non-web `ApplicationContext`).

The `setupMockMvc()` method is annotated with JUnit's `@Before`, indicating that it should be executed before any test methods. It passes the injected `WebApplicationContext` into the `webAppContextSetup()` method and then calls `build()` to produce a `MockMvc` instance, which is assigned to an instance variable for test methods to use.

Now that we have a `MockMvc`, we're ready to write some test methods. Let's start with a simple test method that performs an HTTP `GET` request against /readingList and asserts that the model and view meet our expectations. The following `homePage()` test method does what we need:

```
@Test
public void homePage() throws Exception {
  mockMvc.perform(MockMvcRequestBuilders.get("/readingList"))
      .andExpect(MockMvcResultMatchers.status().isOk())
      .andExpect(MockMvcResultMatchers.view().name("readingList"))
      .andExpect(MockMvcResultMatchers.model().attributeExists("books"))
      .andExpect(MockMvcResultMatchers.model().attribute("books",
          Matchers.is(Matchers.empty())));
}
```

As you can see, a lot of static methods are being used in this test method, including static methods from Spring's `MockMvcRequestBuilders` and `MockMvcResultMatchers`, as well as from the Hamcrest library's `Matchers`. Before we dive into the details of this test method, let's add a few static imports so that the code is easier to read:

```
import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
    ➥ MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
    ➥ MockMvcResultMatchers.*;
```

With those static imports in place, the test method can be rewritten like this:

```
@Test
public void homePage() throws Exception {
  mockMvc.perform(get("/readingList"))
      .andExpect(status().isOk())
      .andExpect(view().name("readingList"))
      .andExpect(model().attributeExists("books"))
      .andExpect(model().attribute("books", is(empty())));
}
```

Now the test method almost reads naturally. First it performs a `GET` request against /readingList. Then it expects that the request is successful (`isOk()` asserts an HTTP 200 response code) and that the view has a logical name of `readingList`. It also asserts that the model contains an attribute named `books`, but that attribute is an empty collection. It's all very straightforward.

The main thing to note here is that at no time is the application deployed to a web server. Instead it's run within a mocked out Spring MVC, just capable enough to handle the HTTP requests we throw at it via the `MockMvc` instance.

Pretty cool, huh?

Let's try one more test method. This time we'll make it a bit more interesting by actually sending an HTTP `POST` request to post a new book. We should expect that after the `POST` request is handled, the request will be redirected back to /readingList and that the `books` attribute in the model will contain the newly added book. The following listing shows how we can use Spring's Mock MVC to do this kind of test.

Listing 4.3 Testing the post of a new book

```
@Test
public void postBook() throws Exception {                          Performs
mockMvc.perform(post("/readingList")                               POST request
       .contentType(MediaType.APPLICATION_FORM_URLENCODED)
       .param("title", "BOOK TITLE")
       .param("author", "BOOK AUTHOR")
       .param("isbn", "1234567890")
       .param("description", "DESCRIPTION"))
       .andExpect(status().is3xxRedirection())
       .andExpect(header().string("Location", "/readingList"));

Book expectedBook = new Book();                  Sets up
expectedBook.setId(1L);                          expected book
expectedBook.setReader("craig");
expectedBook.setTitle("BOOK TITLE");
expectedBook.setAuthor("BOOK AUTHOR");
expectedBook.setIsbn("1234567890");
expectedBook.setDescription("DESCRIPTION");

mockMvc.perform(get("/readingList"))                   Performs GET
       .andExpect(status().isOk())                     request
       .andExpect(view().name("readingList"))
```

```
            .andExpect(model().attributeExists("books"))
            .andExpect(model().attribute("books", hasSize(1)))
            .andExpect(model().attribute("books",
                         contains(samePropertyValuesAs(expectedBook))));
}
```

Obviously, the test in listing 4.3 is a bit more involved. It's actually two tests in one method. The first part posts the book and asserts the results from that request. The second part performs a fresh `GET` request against the home page and asserts that the newly created book is in the model.

When posting the book, we must make sure we set the content type to "application/x-www-form-urlencoded" (with `MediaType.APPLICATION_FORM_URLENCODED`) as that will be the content type that a browser will send when the book is posted in the running application. We then use the `MockMvcRequestBuilders`'s `param()` method to set the fields that simulate the form being submitted. Once the request has been performed, we assert that the response is a redirect to /readingList.

Assuming that much of the test method passes, we move on to part two. First, we set up a `Book` object that contains the expected values. We'll use this to compare with the value that's in the model after fetching the home page.

Then we perform a `GET` request for /readingList. For the most part, this is no different than how we tested the home page before, except that instead of an empty collection in the model, we're checking that it has one item, and that the item is the same as the expected `Book` we created. If so, then our controller seems to be doing its job of saving a book when one is posted to it.

So far, these tests have assumed an unsecured application, much like the one we wrote in chapter 2. But what if we want to test a secured application, such as the one from chapter 3?

4.2.2 Testing web security

Spring Security offers support for testing secured web applications easily. In order to take advantage of it, you must add Spring Security's test module to your build. The following `testCompile` dependency in Gradle is all you need:

```
testCompile("org.springframework.security:spring-security-test")
```

Or if you're using Maven, add the following `<dependency>` to your build:

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

With Spring Security's test module in your application's classpath, you just need to apply the Spring Security configurer when creating the `MockMvc` instance:

```
@Before
public void setupMockMvc() {
mockMvc = MockMvcBuilders
    .webAppContextSetup(webContext)
    .apply(springSecurity())
    .build();
}
```

The `springSecurity()` method returns a Mock MVC configurer that enables Spring Security for Mock MVC. By simply applying it as shown here, Spring Security will be in play on all requests performed through `MockMvc`. The specific security configuration will depend on how you've configured Spring Security (or how Spring Boot has auto-configured Spring Security). In the case of the reading-list application, it's the same security configuration we created in `SecurityConfig.java` in chapter 3.

THE SPRINGSECURITY() METHOD `springSecurity()` is a static method of `SecurityMockMvcConfigurers`, which I've statically imported for readability's sake.

With Spring Security enabled, we can no longer simply request the home page and expect an HTTP 200 response. If the request isn't authenticated, we should expect a redirect to the login page:

```
@Test
public void homePage_unauthenticatedUser() throws Exception {
mockMvc.perform(get("/"))
    .andExpect(status().is3xxRedirection())
    .andExpect(header().string("Location",
                               "http://localhost/login"));
}
```

But how can we perform an authenticated request? Spring Security offers two annotations that can help:

- `@WithMockUser`—Loads the security context with a `UserDetails` using the given username, password, and authorization
- `@WithUserDetails`—Loads the security context by looking up a `UserDetails` object for the given username

In both cases, Spring Security's security context is loaded with a `UserDetails` object that is to be used for the duration of the annotated test method. The `@WithMockUser` annotation is the most basic of the two. It allows you to explicitly declare a `UserDetails` to be loaded into the security context:

```
@Test
@WithMockUser(username="craig",
              password="password",
              roles="READER")
public void homePage_authenticatedUser() throws Exception {
  ...
}
```

As you can see, @WithMockUser bypasses the normal lookup of a UserDetails object and instead creates one with the values specified. For simple tests, this may be fine. But for our test, we need a Reader (which implements UserDetails) instead of the generic UserDetails that @WithMockUser creates. For that, we'll need @WithUserDetails.

The @WithUserDetails annotation uses the configured UserDetailsService to load the UserDetails object. As you'll recall from chapter 3, we configured a UserDetailsService bean that looks up and returns a Reader object for a given username. That's perfect! So we'll annotate our test method with @WithUserDetails, as shown in the following listing.

Listing 4.4 Testing a secured method with user authentication

```
@Test
@WithUserDetails("craig")                                      <- Uses "craig" user
public void homePage_authenticatedUser() throws Exception {

  Reader expectedReader = new Reader();                        <- Sets up expected Reader
  expectedReader.setUsername("craig");
  expectedReader.setPassword("password");
  expectedReader.setFullname("Craig Walls");

  mockMvc.perform(get("/"))                                    <- Performs GET request
      .andExpect(status().isOk())
      .andExpect(view().name("readingList"))
      .andExpect(model().attribute("reader",
                          samePropertyValuesAs(expectedReader)))
      .andExpect(model().attribute("books", hasSize(0)))

}
```

In listing 4.4, we use @WithUserDetails to declare that the "craig" user should be loaded into the security context for the duration of this test method. Knowing that the Reader will be placed into the model, the method starts by creating an expected Reader object that it can compare with the model later in the test. Then it performs the GET request and asserts the view name and model contents, including the model attribute with the name "reader".

Once again, no servlet container is started up to run these tests. Spring's Mock MVC takes the place of an actual servlet container. The benefit of this approach is that the test methods run faster because they don't have to wait for the server to start. Moreover, there's no need to fire up a web browser to post the form, so the test is simpler and faster.

On the other hand, it's not a complete test. It's better than simply calling the controller methods directly, but it doesn't truly exercise the application in a web browser and verify the rendered view. To do that, we'll need to start a real web server and hit it with a real web browser. Let's see how Spring Boot can help us start a real web server for our tests.

4.3 *Testing a running application*

When it comes to testing web applications, nothing beats the real thing. Firing up the application in a real server and hitting it with a real web browser is far more indicative of how it will behave in the hands of users than poking at it with a mock testing engine.

But real tests in real servers with real web browsers can be tricky. Although there are build-time plugins for deploying applications in Tomcat or Jetty, they are clunky to set up. Moreover, it's nearly impossible to run any one of a suite of many such tests in isolation or without starting up your build tool.

Spring Boot, however, has a solution. Because Spring Boot already supports running embedded servlet containers such as Tomcat or Jetty as part of the running application, it stands to reason that the same mechanism could be used to start up the application along with its embedded servlet container for the duration of a test.

That's exactly what Spring Boot's `@WebIntegrationTest` annotation does. By annotating a test class with `@WebIntegrationTest`, you declare that you want Spring Boot to not only create an application context for your test, but also to start an embedded servlet container. Once the application is running along with the embedded container, you can issue real HTTP requests against it and make assertions against the results.

For example, consider the simple web test in listing 4.5, which uses `@WebIntegrationTest` to start the application along with a server and uses Spring's `RestTemplate` to perform HTTP requests against the application.

Listing 4.5 Testing a web application in-server

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
      classes=ReadingListApplication.class)
@WebIntegrationTest
public class SimpleWebTest {

  @Test(expected=HttpClientErrorException.class)
  public void pageNotFound() {
    try {
      RestTemplate rest = new RestTemplate();
      rest.getForObject(
           "http://localhost:8080/bogusPage", String.class);
      fail("Should result in HTTP 404");
    } catch (HttpClientErrorException e) {
      assertEquals(HttpStatus.NOT_FOUND, e.getStatusCode());
      throw e;
    }
  }

}
```

Runs test in server

Performs GET request

Asserts HTTP 404 (not found) response

Although this is a very simple test, it sufficiently demonstrates how to use the `@WebIntegrationTest` to start the application with a server. The actual server that's started will be determined in the same way it would be if we were running the application at the command line. By default, it starts Tomcat listening on port 8080. Optionally, however, it could start Jetty or Undertow if either of those is in the classpath.

The body of the test method is written assuming that the application is running and listening on port 8080. It uses Spring's `RestTemplate` to make a request for a nonexistent page and asserts that the response from the server is an HTTP 404 (not found). The test will fail if any other response is returned.

4.3.1 Starting the server on a random port

As mentioned, the default behavior is to start the server listening on port 8080. That's fine for running a single test at a time on a machine where no other server is already listening on port 8080. But if you're like me, you've probably *always* got something listening on port 8080 on your local machine. In that case, the test would fail because the server wouldn't start due to the port collision. There must be a better way.

Fortunately, it's easy enough to ask Spring Boot to start up the server on a randomly selected port. One way is to set the `server.port` property to 0 to ask Spring Boot to select a random available port. `@WebIntegrationTest` accepts an array of `String` for its `value` attribute. Each entry in the array is expected to be a name/value pair, in the form `name=value`, to set properties for use in the test. To set `server.port` you can use `@WebIntegrationTest` like this:

```
@WebIntegrationTest(value={"server.port=0"})
```

Or, because there's only one property being set, it can take a simpler form:

```
@WebIntegrationTest("server.port=0")
```

Setting properties via the `value` attribute is handy in the general sense, but `@WebIntegrationTest` also offers a `randomPort` attribute for a more expressive way of asking the server to be started on a random port. You can ask for a random port by setting `randomPort` to `true`:

```
@WebIntegrationTest(randomPort=true)
```

Now that we have the server starting on a random port, we need to be sure we use the correct port when making web requests. At the moment, the `getForObject()` method is hard-coded with port 8080 in its URL. If the port is randomly chosen, how can we construct the request to use the right port?

First we'll need to inject the chosen port as an instance variable. To make this convenient, Spring Boot sets a property with the name `local.server.port` to the value of the chosen port. All we need to do is use Spring's `@Value` to inject that property:

```
@Value("${local.server.port}")
private int port;
```

Now that we have the port, we just need to make a slight change to the `getForObject()` call to use it:

```
rest.getForObject(
    "http://localhost:{port}/bogusPage", String.class, port);
```

Here we've traded the hardcoded 8080 for a `{port}` placeholder in the URL. By passing the `port` property as the last parameter in the `getForObject()` call, we can be assured that the placeholder will be replaced with whatever value was injected into `port`.

4.3.2 Testing HTML pages with Selenium

`RestTemplate` is fine for simple requests and it's perfect for testing REST endpoints. But even though it can be used to make requests against URLs that return HTML pages, it's not very convenient for asserting the contents of the page or performing operations on the page itself. At best, you'll be able to assert the precise content of the resulting HTML (which will result in fragile tests). But you won't easily be able to assert selected content on the page or perform operations such as clicking links or submitting forms.

A better choice for testing HTML applications is Selenium (www.seleniumhq.org). Selenium does more than just perform requests and fetch the results for you to verify. Selenium actually fires up a web browser and executes your test within the context of the browser. It's as close as you can possibly get to performing the tests manually with your own hands. But unlike manual testing, Selenium tests are automated and repeatable.

To test our reading list application using Selenium, let's write a test that fetches the home page, fills out the form for a new book, posts the form, and then finally asserts that the landing page includes the newly added book.

First we'll need to add Selenium to the build as a test dependency:

```
testCompile("org.seleniumhq.selenium:selenium-java:2.45.0")
```

Now we can write the test class. The following listing shows a basic template for a Selenium test that uses Spring Boot's `@WebIntegrationTest`.

Listing 4.6 A template for Selenium testing with Spring Boot

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
      classes=ReadingListApplication.class)
@WebIntegrationTest(randomPort=true)           <-- Starts on a random port
public class ServerWebTests {

  private static FirefoxDriver browser;

  @Value("${local.server.port}")         <-- Injects the port
  private int port;
  @BeforeClass
  public static void openBrowser() {
    browser = new FirefoxDriver();
    browser.manage().timeouts()
        .implicitlyWait(10, TimeUnit.SECONDS);   <-- Sets up Firefox driver
  }

  @AfterClass
  public static void closeBrowser() {
    browser.quit();                <-- Shuts down browser
  }

}
```

As with the simpler web test we wrote earlier, this class is annotated with `@WebIntegrationTest` and sets `randomPort` to `true` so that the application will be started and run with a server listening on a random port. And, as before, that port is injected into the `port` property so that we can use it to construct URLs to the running application.

The static `openBrowser()` method creates a new instance of `FirefoxDriver`, which will open a Firefox browser (it will need to be installed on the machine running the test). When we write our test method, we'll perform browser operations through the `FirefoxDriver` instance. The `FirefoxDriver` is also configured to wait up to 10 seconds when looking for any elements on the page (in case those elements are slow to load).

After the test has completed, we'll need to shut down the Firefox browser. Therefore, `closeBrowser()` calls `quit()` on the `FirefoxDriver` instance to bring it down.

> **PICK YOUR BROWSER** Although we're testing with Firefox, Selenium also provides drivers for several other browsers, including Internet Explorer, Google's Chrome, and Apple's Safari. Not only can you use other browsers, it's probably a good idea to write your tests to use any and all browsers you want to support.

Now we can write our test method. As a reminder, we want to load the home page, fill in and submit the form, and then assert that we land on a page that includes our newly added book in the list. The following listing shows how to do this with Selenium.

Listing 4.7 Testing the reading-list application with Selenium

```
@Test
public void addBookToEmptyList() {
  String baseUrl = "http://localhost:" + port;

  browser.get(baseUrl);                    <-- Fetches the home page

  assertEquals("You have no books in your book list",
               browser.findElementByTagName("div").getText());   <-- Asserts an empty book list

  browser.findElementByName("title")
```

```
         .sendKeys("BOOK TITLE");
  browser.findElementByName("author")
         .sendKeys("BOOK AUTHOR");
  browser.findElementByName("isbn")
         .sendKeys("1234567890");
  browser.findElementByName("description")
         .sendKeys("DESCRIPTION");
  browser.findElementByTagName("form")
         .submit();                          <-- Fills in and submits form

  WebElement dl =
      browser.findElementByCssSelector("dt.bookHeadline");
  assertEquals("BOOK TITLE by BOOK AUTHOR (ISBN: 1234567890)",
               dl.getText());
  WebElement dt =
      browser.findElementByCssSelector("dd.bookDescription");
  assertEquals("DESCRIPTION", dt.getText());        <-- Asserts new book in list
}
```

The very first thing that the test method does is use the `FirefoxDriver` to perform a `GET` request for the reading list's home page. It then looks for a `<div>` element on the page and asserts that its text indicates that no books are in the list.

The next several lines look for the fields in the form and use the driver's `sendKeys()` method to simulate keystroke events on those field elements (essentially filling in those fields with the given values). Finally, it looks for the `<form>` element and submits it.

After the form submission is processed, the browser should land on a page with the new book in the list. So the final few lines look for the `<dt>` and `<dd>` elements in that list and assert that they contain the data that the test submitted in the form.

When you run this test, you'll see the browser pop up and load the reading-list application. If you pay close attention, you'll see the form filled out, as if by a ghost. But it's no spectre using your application—it's the test.

The main thing to notice about this test is that `@WebIntegrationTest` was able to start up the application and server for us so that Selenium could start poking at it with a web browser. But what's especially interesting about how this works is that you can use the test facilities of your IDE to run as many or as few of these tests as you want, without having to rely on some plugin in your application's build to start a server for you.

If testing with Selenium is something that you think you'll find useful, you should check out *Selenium WebDriver in Practice* by Yujun Liang and Alex Collins (http://manning.com/liang/), which goes into far more details about testing with Selenium.

4.4 Summary

Testing is an important part of developing quality software. Without a good suite of tests, you'll never know for sure if your application is doing what it's expected to do.

For unit tests, which focus on a single component or a method of a component, Spring isn't really necessary. The benefits and techniques promoted by Spring—loose coupling, dependency injection, and interface-driven design—make writing unit tests easy. But Spring doesn't need to be directly involved in unit tests.

Integration-testing multiple components, however, begs for help from Spring. In fact, if Spring is responsible for wiring those components up at runtime, then Spring should also be responsible for wiring them up in integration tests.

The Spring Framework provides integration-testing support in the form of a JUnit class runner that loads a Spring application context and enables beans from the context to be injected into a test. Spring Boot builds upon Spring integration-testing support with a configuration loader that loads the application context in the same way as Spring Boot itself, including support for externalized properties and Spring Boot logging.

Spring Boot also enables in-container testing of web applications, making it possible to fire up your application to be served by the same container that it will be served by when running in production. This gives your tests the closest thing to a real-world environment for verifying the behavior of the application.

At this point we've built a rather complete (albeit simple) application that leverages Spring Boot starters and auto-configuration to handle the grunt work so that we can focus on writing our application. And we've also seen how to take advantage of Spring Boot's support for testing the application. Coming up in the next couple of chapters, we're going to take a slightly different tangent and explore the ways that Groovy can make developing Spring Boot applications even easier. We'll start in the next chapter by looking at a few features from the Grails framework that have made their way into Spring Boot.