What this book will do for you...

- Install Maven and Tomcat and create a Spring MVC web application
- Display, validate, and process forms
- Query a database with minimal code
- Authenticate users and manage their roles
- Build REST, Java RMI, and Hessian web services
- Create and schedule batch jobs
- Offer customized page layouts to mobiles and tablets
- Exchange data with Facebook and Twitter

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

Spring Cookbook

Over 100 hands-on recipes to build Spring web applications easily and efficiently

Jérôme Jaglale

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Using Controllers and Views'
- A synopsis of the book’s content
- More information on Spring Cookbook

About the Author

Jérôme Jaglale is a web developer based in Vancouver, Canada. He has spent the last 10 years working with several web frameworks, including Struts, CodeIgniter, and Laravel, and doing frontend work with CSS and JavaScript that is focused on user experience.

He's currently working at Simon Fraser University on a prototype of iReceptor, a scientific c gateway federating immune genetics databases in order to help researchers answer complex questions about immune response.
Spring Cookbook

Web development with Java has a high learning curve compared to Ruby on Rails, Django, and modern PHP frameworks. Spring, the most used Java framework for web development, makes it as easy as it can get, especially with its recent move of using plain Java classes instead of XML files for configuration classes for configuration. This book focuses on getting you up and running with Spring 4 in the most efficient manner.

What This Book Covers

Chapter 1, Creating a Spring Application, covers the installation of Java and other software on Mac OS, Ubuntu, and Windows. It also explains how to build a Spring application and a Spring web application.

Chapter 2, Defining Beans and Using Dependency Injection, introduces Spring beans and demonstrates how to define and use them.

Chapter 3, Using Controllers and Views, describes how to create controller methods, use JSP views, and build page templates and multilingual pages.

Chapter 4, Querying a Database, explains how to interact with a database using JDBC and how to integrate Hibernate into a Spring application.

Chapter 5, Using Forms, describes how to initialize a form, retrieve the form data when the form is submitted, and elaborates on the use of forms widgets (text fields, select fields, and so on).

Chapter 6, Managing Security, introduces Spring Security and demonstrates how to perform user authentication and user authorization, and how to configure HTTPS.

Chapter 7, Unit Testing, introduces unit testing with JUnit and TestNG and explains how to test Spring applications.

Chapter 8, Running Batch Jobs, details how batch jobs work with Spring and explains how to build batch jobs and execute them from a web application or from the command line.
Chapter 9, *Handling Mobiles and Tablets*, explains how to make a Spring web application display different content based on the type of device accessing it.

Chapter 10, *Connecting to Facebook and Twitter*, explains how to access a Facebook or Twitter account in order to fetch some existing data or to create new data (tweets, posts, and so on).

Chapter 11, *Using the Java RMI, HTTP Invoker, Hessian, and REST*, covers how a Spring application can interact with other pieces of software over the network using various technologies.

Chapter 12, *Using Aspect-oriented Programming*, explains what AOP (aspect-oriented programming) is, how it can be used with Spring, and covers several of its common uses.
In this chapter, we will cover the following recipes:

- Associating a route to a controller method
- Using a JSP view
- Passing attributes from a controller to a JSP view
- Using dynamic route parameters in a controller method
- Using a common prefix for the routes of a controller
- Using a page template with Tiles
- Executing some code before and after controllers using interceptors
- Building multilingual pages
Introduction

A Spring web application uses a MVC (Model-View-Controller) architecture to process HTTP requests, as shown in the following image:

An HTTP request, identified by a route (for example, /user/list), executes a controller method. A view, usually a JSP file, is rendered afterwards and the resulting HTML is sent back as a response.

In this chapter, we will start by creating a controller and view. Then, you'll learn how to retrieve URL parameters from a controller method. We'll cover two standard ways to reduce code repetition with page templates and URL prefixes. We will finish with more advanced topics related to controllers and views: interceptors and internationalization.

The recipes in this chapter will work with a project similar to the one in the Creating a Spring web application recipe in Chapter 1, Creating a Spring Application, with a Spring configuration class annotated with @EnableWebMvc and scanning a Java package dedicated to controller classes:

```java
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.springcookbook.controller"})
public class AppConfig {
}
```
This is the project structure:

```
+ com.springcookbook.config
  └── AppConfig.java
    │ ServletInitializer.java
+ com.springcookbook.controller
  └── HelloController.java
```

## Associating a route to a controller method

In this recipe, you will learn how to define a controller method to be executed for a given route.

### How to do it...

Here are the steps for creating a controller method for a given route:

1. Create a controller class in your controller package (for example, `com.springcookbook.controller`). This is a normal Java class annotated with `@Controller`:
   ```java
   @Controller
   public class UserController {
     ...
   }
   ```
2. Add a controller method. This is a standard Java method annotated with `@RequestMapping`, which takes the route as a parameter:
   ```java
   @RequestMapping("/user/list")
   public void userList() {
     ...
   }
   ```

### How it works...

A request with the `/user/list` route will execute the `userList()` method.

## Using a JSP view

In this recipe, you'll learn how to render and return a JSP view after the execution of a controller method.
How to do it...

Here are the steps to create a JSP view:

1. Add the Maven dependency for JSTL in `pom.xml`:
   ```xml
   <dependency>
     <groupId>javax.servlet</groupId>
     <artifactId>jstl</artifactId>
     <version>1.2</version>
   </dependency>
   
   2. Add a JSP view resolver to the Spring configuration class:
      ```java
      @Bean
      public ViewResolver jspViewResolver(){
        InternalResourceViewResolver resolver = new
        InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/jsp/");
        resolver.setSuffix(".jsp");
        return resolver;
      }
      
   3. Create a controller method:
      ```java
      @RequestMapping("/user/list")
      public void userList() {
        ...
      }
      
   4. Create the `/WEB-INF/jsp/user/list.jsp` JSP:
      ```html
      <html>
      <body>
        There are many users.
      </body>
      </html>
      
How it works...

The controller method path is `/user/list`. Using the JSP view resolver, Spring will find and render the corresponding `/WEB-INF/jsp/user/list.jsp` JSP.

If the path had been `/user_list`, the corresponding JSP would have been `/WEB-INF/jsp/user_list.jsp`. 
This is the current project structure:

There's more...

It's possible to explicitly return a String object from the controller method, which Spring will use to find the JSP. In this example, `/WEB-INF/jsp/my_friends.jsp` will be used:

```java
@RequestMapping("/user/list")
public String userList() {
    return "my_friends";
}
```

For more information about what can be done in a JSP file, refer to [http://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm](http://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm).

Thymeleaf, FreeMarker, and Velocity are popular view frameworks that provide an alternative to JSPs. FreeMarker and Velocity are supported by Spring by default. Thymeleaf provides its own view resolver.
Passing attributes from a controller to a JSP view

In this recipe, you'll learn how to set attributes in a controller method and use them in a JSP view.

How to do it...

Here are the steps to pass data from a controller to a view:

1. Add a Model argument to the controller method:
   ```java
   @RequestMapping("/user/list")
   public void userList(Model model) {
   ...
   ```

2. In the controller method, add attributes to the Model object:
   ```java
   model.addAttribute("nbUsers", 13);
   ```

3. Use the attributes in the JSP file:
   ```html
   <p>There are ${nbUsers} users</p>
   ```

How it works...

The nbUsers variable is set to 13 in the controller. In the JSP file, the `${nbUsers}` EL (Expression Language) element will be rendered to 13, so that the following HTML will be returned:

```html
<p>There are 13 users</p>
```

Using dynamic route parameters in a controller method

Now we will define dynamic segments for a route and use them in the associated controller method. For example, we want the /user/5/name and /user/6/email routes to execute the same controller method with different arguments: `showUserField(5, "name")` and `showUserField(6, "email")`, respectively.
How to do it...

Use {} to enclose the dynamic route segments and @PathVariable to annotate the corresponding controller method arguments:

```java
@PathVariable("id") Long userId,
@PathVariable("field") String field) {
...
}
```

How it works...

A request for the /user/5/email route will execute the showUserField(5,"email") method. @PathVariable("id") Long userId casts the id route parameter to the userId method argument. Similarly, the field route parameter is passed as String to showUserField().

An incorrect route such as /user/test/email (it's incorrect because the test substring cannot be converted to a Long object) will trigger a 400 server error with the message **The request sent by the client was syntactically incorrect.**

Using a common prefix for the routes of a controller

In this recipe, we will define in one place a route prefix shared by all the routes of a controller. We will start the routes of the UserController controller with /user.

How to do it...

Here are the steps to set a route prefix:

1. Add @RequestMapping with the common route prefix to the controller class:

   ```java
   @Controller
   @RequestMapping("/user")
   public class UserController {
     ...
   }
   ```
Using Controllers and Views

2. Add `@RequestMapping` with the remainder of the route to the controller methods:

```java
@RequestMapping("/list")
public void userList() {
    ...
}

@RequestMapping("/add")
public void addUser() {
    ...
}
```

**How it works...**

A request for the `/user/add` route will execute the `addUser()` method. A request for the `/user/list` route will execute the `userList()` method.

Using a page template with Tiles

With a page template, avoid repeating the common elements of the pages (HTML head, header, footer, navigation, and so on) in every JSP.

**How to do it...**

Here are the steps to use Tiles:

1. Add the Tiles Maven dependencies in `pom.xml`:

   ```xml
   <dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-servlet</artifactId>
   <version>3.0.5</version>
   </dependency>

   <dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-jsp</artifactId>
   <version>3.0.5</version>
   </dependency>
   ```

2. In the Spring configuration class, remove the JSP view resolver (if it's there).
3. Declare Tiles in the Spring configuration class:

```java
// declare Tiles configuration file
@Bean
public TilesConfigurer tilesConfigurer() {
    TilesConfigurer tilesConfigurer = new TilesConfigurer();
    final String[] definitions = { "/WEB-INF/tiles.xml" };
    tilesConfigurer.setDefinitions(definitions);
    return tilesConfigurer;
}

// declare Tiles as a view resolver
@Bean
public ViewResolver tilesViewResolver() {
    TilesViewResolver resolver = new TilesViewResolver();
    return resolver;
}
```

4. Create the /WEB-INF/tiles.xml Tiles configuration file:

```xml
<tiles-definitions>
    <definition name="template" template="/WEB-INF/jsp/templates/template.jsp" />
    <definition name="*" extends="template">
        <put-attribute name="body" value="/WEB-INF/jsp/{1}.jsp" />
    </definition>
</tiles-definitions>
```

5. Create the /WEB-INF/jsp/templates/template.jsp page template:

```html
<!DOCTYPE HTML>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>

<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Spring Cookbook</h1>

        <tiles:insertAttribute name="body" />
    </body>
</html>
```
6. In the controller methods, return the base name of a standard JSP file. For example, for /jsp/home.jsp:

... 
return "home";

How it works...

When the Spring configuration is loaded, Tiles is initialized using the declared tiles.xml configuration file.

When a request arrives, the controller method is executed and returns the "home" String, which matches the definition named "*" in tiles.xml. This definition will use the template definition and pass the body variable to it with the /WEB-INF/jsp/home.jsp value. In template.jsp, the tiles:insertAttribute tag will be replaced by the contents of home.jsp.

To summarize, the home.jsp file is integrated with template.jsp and the resulting HTML is sent as a response.

In template.php, make sure to include the following to be able to use Tiles tags such as:

<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
There's more...

Tiles can be used when JSP files are in subfolders and support multiple page templates. It's also possible to define repeated text in one place.

Organizing the JSP with subfolders

As the number of JSP files grows, you can maintain them by grouping them according to sections, using subfolders:

```
/jsp
  |- /user
  |   |- list.jsp
  |   |- add.jsp
  |- /article
  |   |- list.jsp
  |   |- add.jsp
  |- home.jsp
```

In the controller method, return the folder with the `jsp` base name, for example, `user/list`.

In `tiles.xml`, add the definition:

```
<definition name="**" extends="template">
  <put-attribute name="body" value="/WEB-INF/jsp/{1}/{2}.jsp" />
</definition>
```

Using multiple page templates

To handle multiple templates, define a prefix for each template in `tiles.xml`. For example, we defined below a main template with the `main_` prefix using the `template1.jsp` JSP and a secondary template with the `secondary_` prefix using the `template2.jsp` JSP:

```
<definition name="template1">
  template="/WEB-INF/templates/template1.jsp" />
<definition name="template2">
  template="/WEB-INF/templates/template2.jsp" />

<definition name="main_*" extends="template1">
  <put-attribute name="body" value="/WEB-INF/jsp/{1}.jsp" />
</definition>

<definition name="secondary_*" extends="template2">
  <put-attribute name="body" value="/WEB-INF/jsp/{1}.jsp" />
</definition>
```
In the controller method, for home.jsp, return "main_home" to use template1 or "secondary_home" to use template2.

Defining page titles only once using a text attribute

A title usually needs to appear twice in an HTML page: once in the <title> tag of the page <head> section and once in a <h1> tag of the page <body> section. Using Tiles, you can define it only once in an external .properties file:

1. In tiles.xml, add a title attribute to the template definition:
   ```xml
   <definition name="*" extends="template">
     <put-attribute name="title" value="{1}.title" />
   ...
   ```

2. In template.jsp, get the title and use it:
   ```jsp
   <!DOCTYPE HTML>
   <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
   <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
   
   <c:set var="titleKey">
     <tiles:getAsString name="title" />
   </c:set>
   
   <html>
   <head>
     <title><spring:message code="${titleKey}" /></title>
   </head>
   <body>
     <h1><spring:message code="${titleKey}" /></h1>
   ...
   ```

3. Create the src/main/resources/messages.properties file:
   ```properties
   home.title=Home
   ```

To learn more about Tiles, go to https://tiles.apache.org/framework/tutorial/.
Executing some code before and after controllers using interceptors

In this recipe, you'll learn how, with interceptors, we can execute some code across all controllers at different moments of a request workflow with the `preHandle()`, `postHandle()`, and `afterCompletion()` hooks:

```
request
  ↓
preHandle()
  ↓
controller method is executed
↓
postHandle()
↓
View is rendered
↓
afterCompletion()
response
```

Interceptors are used for authentication, logging, and profiling (among others).

**How to do it...**

Here are the steps to create and register an interceptor:

1. Create a class extending `HandlerInterceptorAdapter`:
   ```java
   public class PerformanceInterceptor extends HandlerInterceptorAdapter {
   
   2. Override the methods you want to use:
      ```java
      @Override
      public boolean preHandle(HttpServletRequest request,
                                HttpServletResponse response, Object handler) throws Exception {
      ```
Using Controllers and Views

... return true;
}

@override
public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler,
    ModelAndView modelAndView) throws Exception {
    ...
}

@override
public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
    ...
}

Note that if preHandle() returns false, the request workflow will be stopped and the controller method won’t be called.

3. Make the Spring configuration class extend WebMvcConfigurerAdapter and annotate it with @EnableWebMvc:

@Configuration
@EnableWebMvc
public class AppConfig extends WebMvcConfigurerAdapter{
...

4. In the Spring configuration class, declare the interceptor as a bean and register it with the addInterceptors() method:

@Bean
public HandlerInterceptor performanceInterceptor() {
    PerformanceInterceptor interceptor;
    interceptor = new PerformanceInterceptor();
    return interceptor;
}

@override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(performanceInterceptor());
}
How it works...

The interceptor methods are executed at the corresponding moments of the request workflow.

There's more...

To restrict the interceptor to specific URLs, add path patterns to the interceptor registration:

```java
@override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(performanceInterceptor())
        .addPathPatterns("/home",="/user/*");
}
```

In this example, the interceptor methods will be executed for /home, /user/list, and /user/add but not for /contact.

Building multilingual pages

Next we will learn how to create a multilingual page (in English and French) using only one JSP, and display it in English by default, with a link to switch to French. We will then store the text outside the JSP, in both languages, in separate .properties files.

How to do it...

Here are the steps to build a bilingual JSP view:

1. Create the JSP:

   ```html
   <html>
   <body>
   <h1><spring:message code="home.title" /></h1>
   <p><spring:message code="home.intro" /></p>
   
   <p>
   <a href="?lang=en">English</a> | 
   <a href="?lang=fr">French</a>
   </p>
   </body>
   </html>
   ```
2. Create the English `.properties` file `src/main/resources/messages.properties`:
   home.title=Home
   home.intro=This is a magnificent home page, isn't it?

3. Create the French `.properties` file `src/main/resources/messages_fr.properties`:
   home.title=Accueil
   home.intro=Splendide page d'accueil, non ?

4. In Spring configuration, declare the `.properties` files:
   ```java
   @Bean
   public MessageSource messageSource() {
     ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();
     messageSource.setBasename("classpath:/messages");
     messageSource.setUseCodeAsDefaultMessage(true);
     return messageSource;
   }
   ```

5. Make sure that the Spring configuration class extends `WebMvcConfigurerAdapter` and is annotated with `@EnableWebMvc`:
   ```java
   @Configuration
   @EnableWebMvc
   public class AppConfig extends WebMvcConfigurerAdapter{
     ...
   }
   ```

6. Define a `LocaleChangeInterceptor` interceptor to allow the current language to be changed with a `lang` URL parameter. Register the interceptor:
   ```java
   @Bean
   public HandlerInterceptor localeChangeInterceptor() {
     LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
     interceptor.setParamName("lang");
     return interceptor;
   }
   ```

   ```java
   @Override
   public void addInterceptors(InterceptorRegistry registry) {
     registry.addInterceptor(localeChangeInterceptor());
   }
   ```
7. Store the user language selection in a cookie and declare the default language:

```java
@Bean
public LocaleResolver localeResolver() {
    CookieLocaleResolver localeResolver = new CookieLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("en"));
    return localeResolver;
}
```

**How it works...**

The following steps describe how the preceding code works:

1. When a request comes in, Spring first checks whether it has a cookie value containing a language. If the answer is yes, it uses it as the current language; otherwise, it uses the default language. This behavior comes from the declaration of CookieLocaleResolver as the locale resolver.

2. The LocaleChangeInterceptor then checks whether there's a lang parameter in the URL. If the answer is yes, it uses it as the current language (instead of the default or cookie language).

3. When home.jsp is rendered, its text is fetched from the .properties file corresponding to the current language. If no text is found for a given message key, the key itself is displayed. This behavior comes from messageSource.setUseCodeAsDefaultMessage(true).

**There's more...**

You may need to retrieve the current language name from a controller method. You may also need to have the page language in the URL instead of in a cookie.

**Getting the current language**

To retrieve the current language from a controller or an interceptor, use the following code:

```java
Locale locale = LocaleContextHolder.getLocale();
String lang = locale.getLanguage(); // fr
String language = locale.getDisplayLanguage(); // French
String language2 = locale.getDisplayLanguage(locale); // français
```
Using the language in the URL

Spring does not provide a convenient way to handle the language in the URL (for example, /en/user/list). Instead, it has to be done manually:

1. Use an interceptor to retrieve the language from the URL and override the current language.

2. Prefix the controller method mappings with the supported languages (so that Spring can retrieve it from the route with the language):

   ```java
   @Controller
   @RequestMapping("{en|fr}/user/*")
   public class UserController {
     @RequestMapping("list")
     public String userList(Model model) {
       ...
     }
   }
   ```

3. When generating an internal link, prefix it with the current language, assuming that $lang contains the current language:

   ```html
   <spring:url value="/${lang}/home" var="home" />
   <a href="${home}">Home</a>
   ```
Where to buy this book

You can buy Spring Cookbook from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.