4 Spring Web MVC Framework

In the last chapter, we introduced some of the Spring core concepts, such as a Spring IoC container, Spring beans, Spring dependency injection, and so on. Equipped with the knowledge of those core concepts, we are ready to explore one of the most widely used Spring Frameworks – the Spring Web MVC Framework. Since this is one of the Spring Frameworks that SOBA is built on, this chapter covers it detail.

Before we start, let me remind you of the architecture of a typical enterprise web application in the next section.

4.1 ENTERPRISE WEB APPLICATION ARCHITECTURE

A typical enterprise web application consists of multiple tiers, as shown in Figure 4.1. Note that this is only a logical division that you can deploy physically all tiers on one system or several separate systems.

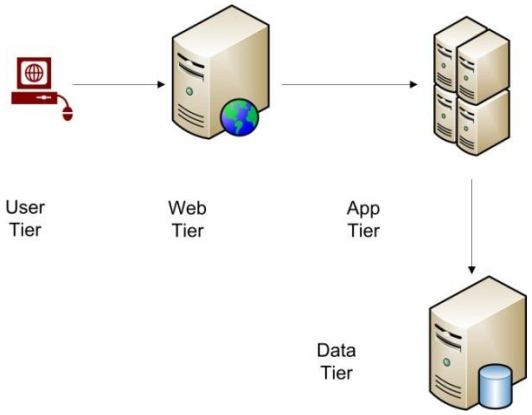

Figure 4.1 Architecture of a typical n-tier enterprise web application

Next, let us take a brief look at how all those tiers collaborate with each other to fulfill the tasks of an enterprise web application.

4.1.1 Data Tier

The data tier is also called backend tier, which stores and provides services related to your enterprise data. It has to be hosted on a particular database platform, such as MySQL, SQL Server, Oracle, or IBM's DB2. DB2 is not covered in this book, though. The common concerns about which database platform to choose are multi-faceted, such as whether your application needs to support multiple database platforms, preferences of your potential customers, strengths of a particular database platform, and the associated development and maintenance costs, etc. My preference is MySQL, because, first, it is a proven platform with successful deployments by many large companies; and secondly, it is one of the most platform-neutral database platforms that it can be deployed on UNIX, Linux, and Windows. In addition, it is quite development-friendly in the sense that all open source development technologies are well supported.

From a functional point of view, the data tier is where your domain objects reside eventually. If you have gone through the SOBA schema covered in Section 2.3 or you are an experienced enterprise developer, you already understand well what this tier does. We will get more concrete with the data tier later when I walk you through how SOBA is built with JDBC and Hibernate that provide connectivity between the app tier and the data tier.

4.1.2 Application Tier

The application tier implements your business logic. It requires a runtime environment to support executing your business logic, which can be fulfilled with one of the application server products built on Java or other technologies such as the Spring platform combined with a web server such as Tomcat, Oracles' WebLogic, IBM's WebSphere, RedHat's JBoss, and so on.

4.1.3 Web Tier

The basic function of a Web tier is to receive service requests from users, delegate such requests to the application tier, receives and send processed results back to users. In SOBA's context, the application tier and Web tier are inseparable that they run in the same JVM of the Tomcat web server. For large enterprise web applications, the web tier and application tier can be separated and deployed on two or more separate physical systems or clusters, in which case, some sort of remoting mechanisms such as JAVA RMI (Remote Method Invocation), web services, etc., are introduced to enable the communications between the two tiers.

4.1.4 User Tier

The user tier is where a user interacts with an enterprise Web application, mostly through the HTTP protocol. This tier is also called client tier, represented by a variety of client devices, such as PCs, Macs, tablets, mobile devices, etc. At this tier, responses from a frontend Web server are rendered to users through applications such as various types of browsers, etc.

In the next section, we introduce a generic Model-View-Controller (MVC) architecture, which serves as the backbone for building enterprise web applications. We'll see how it maps to the n-tier architecture that we have just described.

4.2 MVC ARCHITECTURE IN GENERAL

In a Model-View-Controller architecture, *model* handles application domain data and business logic, *view* represents what a user would see based on the responses a user receives from the system, whereas *controller* acts more like an orchestrator that coordinates the interactions between a user and the application system. Loosely speaking in the context of the n-tier enterprise architecture, *model* corresponds to the data tier and application tier, *controller* corresponds to the web tier, while *view* corresponds to the user or client tier.

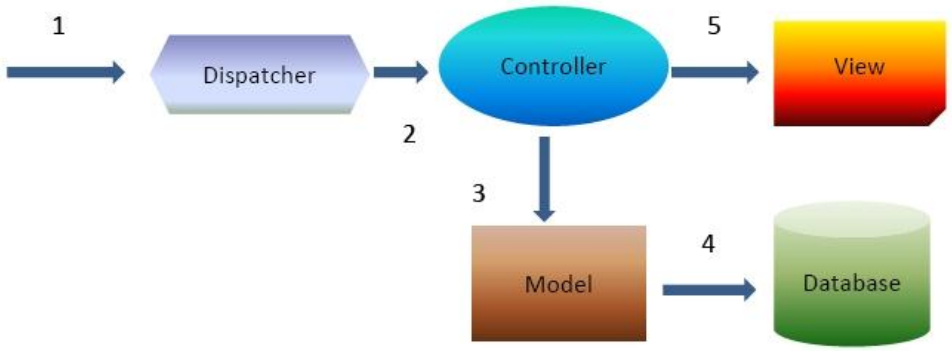

Figure 4.2 A generic MVC architecture

The interactions among the model, view and controller of an MVC architecture are explained in Figure 4.2 further:

1 A user request is sent to the system for viewing, creating, modifying, or deleting data.
2 The dispatcher dispatches the user request to the controller.
3 The controller parses the user requests and calls the model one or multiple times for retrieving data or carrying out the user requested changes to the model.
4 The model queries the database or initiates and commits changes to the database based on a user's request.
5 The controller returns data or outcome of the user request to the user.

The Spring MVC framework is a concrete implementation of a generic MVC architecture. Next, let's see how such a generic MVC architecture is implemented with the Spring Web MVC Framework.

4.3 SPRING WEB MVC FRAMEWORK

The Spring Web MVC Framework is closely patterned on the generic MVC architecture that we introduced in the previous section. See Figure 4.3 for how Spring MVC Framework works and compare it with the generic MVC architecture shown in Figure 4.2. As is seen, the workflow with the Spring Web MVC Framework gets more specific.

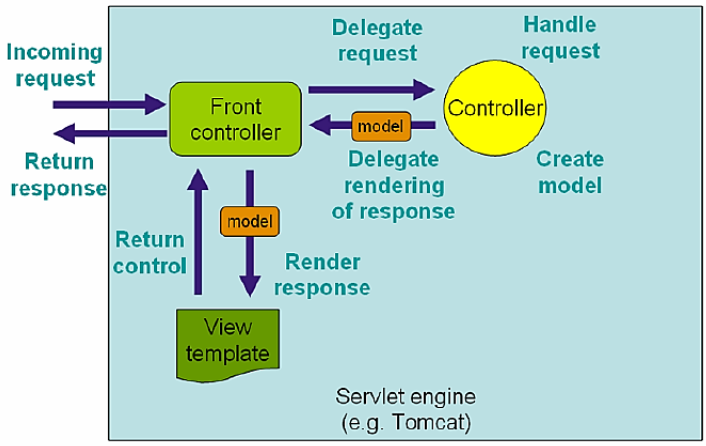

Figure 4.3 Spring Web MVC Framework workflow

Next, let us see how the Spring Web MVC Framework is implemented programmatically.

4.3.1 Spring DispatcherServlet and WebApplicationContext

In Chapter 3, we introduced the `ApplicationContext` interface, which is defined in the `org.springframework.context` package. The `ApplicationContext` interface extends the `BeanFactory` interface, which is defined in the `org.springframework.beans.factory` package. However, in contrast to the `BeanFactory` interface, the `ApplicationContext` interface can be used in a completely declarative manner so that no programming in Java is needed on the developer's side. This is made possible with a support class named `ContextLoader` in conjunction with an XML configuration file that defines all beans. The context loader class automatically instantiates an `ApplicationContext` at the startup time of an application.

Since an `ApplicationContext` is an interface, it must be implemented in order to be usable. The `WebApplicationContext` is one of the implementations of the `ApplicationContext` interface. The `WebApplicationContext` and the `DispatcherServlet` work jointly and play the critical role of *controller* in the Spring MVC Framework, as discussed next.

The `DispatcherServlet` is an expression of the "Front Controller" design pattern. It is defined in the package of `org.springframework.web.servlet`. Its main function is to dispatch user requests to handlers that are managed in a `WebApplicationContext`. Figure 4.4 illustrates the interaction between a `DispatcherServlet` and a `WebApplicationContext` as part of the entire MVC workflow (we'll elaborate on the

concepts of controllers, handler mapping, view resolvers, etc., in the Spring's MVC context soon).

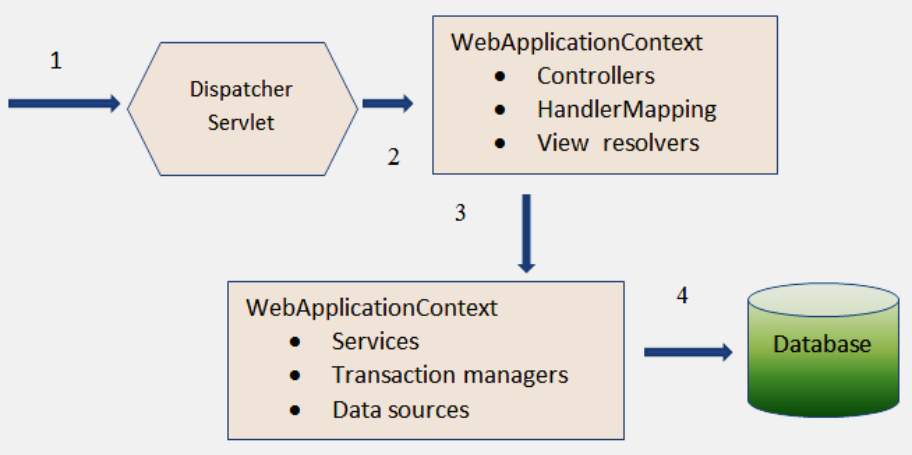

Figure 4.4 Two major elements of the Spring MVC Framework: DispatcherServlet and WebApplicationContext

Since the `DispatcherServlet` inherits from the `HttpServlet` base class, it is an actual servlet, which means that it can be defined in a `web.xml` file like in other non-Spring Web frameworks. To illustrate this, it is now time to show the `web.xml` file for SOBA next.

4.3.2 Bootstrapping and DispatcherServlet Defined in web.xml File

Since the `DispatcherServlet` inherits from the HttpServlet base class, a Spring-based web application is runnable on a general-purpose Web Server or a servlet engine like Tomcat, which depends on a web.xml file to bootstrap the web application. However, what is unique with a Spring MVC-based web application is that it has its own context loader, which provides a flexible mechanism for loading Spring beans defined in various Spring application context configurations.

In this section, we explain how a web application based on the Spring MVC Framework is bootstrapped and how the DispatcherServlet is defined in the `web.xml` file, which is exhibited in Listing 4.1. Now take a closer look at this web.xml file, especially those parts that are highlighted, and we will explain them following this listing.

Listing 4.1 web.xml for SOBA

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <!-- Log4j configuration loading -->
```

```
        <listener>
            <listener-class>org.springframework.web.util.Log4jConfigListener</listener-
class>
        </listener>
        <context-param>
            <param-name>log4jConfigLocation</param-name>
            <param-value>/WEB-INF/classes/log4j.xml</param-value>
        </context-param>
        <!-- Bootstrapping context loading -->
        <listener>
            <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
        </listener>
        <context-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
             /WEB-INF/soba-servlet.xml
             /WEB-INF/soba-services.xml
              /WEB-INF/soba-security.xml
           </param-value>
        </context-param>
        <context-param>
            <param-name>webAppRootKey</param-name>
            <param-value>soba.root</param-value>
        </context-param>
        <!-- session management listener -->
        <listener>
            <listener-
class>org.springframework.security.web.session.HttpSessionEventPublisher
    </listener-class>
        </listener>
        <session-config>
            <!-- session times out if no activities for 30 minutes -->
            <session-timeout>30</session-timeout>
        </session-config>
        <!-- Security entry point -->
        <filter>
            <filter-name>springSecurityFilterChain</filter-name>
            <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
        </filter>
        <filter-mapping>
            <filter-name>springSecurityFilterChain</filter-name>
            <url-pattern>/*</url-pattern>
        </filter-mapping>
        <!-- defining the DispatcherServlet -->
```

```
        <servlet>
            <servlet-name>soba</servlet-name>
            <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
            <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>soba</servlet-name>
            <url-pattern>/</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
            <servlet-name>soba</servlet-name>
            <url-pattern>*.htm</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
            <servlet-name>default</servlet-name>
            <url-pattern>*.jpg</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
            <servlet-name>default</servlet-name>
            <url-pattern>*.gif</url-pattern>
        </servlet-mapping>
        <error-page>
            <error-code>404</error-code>
            <location>/WEB-INF/jsp/notfound.jsp</location>
        </error-page>
        <welcome-file-list>
            <welcome-file>
                 login.jsp
            </welcome-file>
        </welcome-file-list>
    <!-- Spring jsp tag lib -->
        <jsp-config>
            <taglib>
                <taglib-uri>/spring</taglib-uri>
                <taglib-location>/WEB-INF/tld/spring-form.tld</taglib-location>
            </taglib>
        </jsp-config>
    </web-app>
```

This web.xml file defines a `web-app` as its top structural node indicates. It consists of the following entries:

- **Log4j configuration**. This entry defines configuring the log4j logging framework based on `the log4j.xml` file located in the `/WEB-INF/classes` directory.
- **Bootstrapping context loading**. This entry defines a context loader listener for loading contexts specified in the servlet, services, and security XML files. Those files contain all Spring beans in their respective contexts. Note that each of these

files is prefixed with `soba-` in its name, which by convention originates from the dispatcher servlet instance name for this application as defined in the servlet entry down this file. We will talk more about this later when we discuss the dispatcher servlet entry.

- **Session management listener**. This entry defines that each session times out in 30 minutes. You can change this value to suit your needs.
- **Security entry point**. This entry defines that the entire app of SOBA is secured with a url-pattern of `/*`.
- **Defining the DispacherServlet**. This is the entry we have discussed in detail. First, it defines the `DispacherServlet` named soba. Then it defines a few servlet mappings, such as the root path of "/" and all requests ending with `.htm`. Together with the `welcome-file` of `login.jsp` defined later, the URL of https://localhost:8443/soba takes a user to the login page of SOBA as shown in Figure 2.8. We will see later how those requests ending with `*.htm` are mapped to their corresponding Spring MVC controllers defined in the `soab-servlet.xml` file.
- **Spring jsp tag lib**. This last entry defines the Spring jsp tag lib that can be used in jsp files. We will see such examples later.

The key point to remember is that, upon initialization of a DispatcherServlet, the Spring MVC framework looks for a file named `<servlet-name>-servlet.xml` in the `WEB-INF` directory and creates beans defined there. Then the DispatcherServlet starts to dispatch user requests as users interact with a web application built on the Spring Web MVC Framework. As a Spring developer, it's necessary to fully understand Spring Web MVC programming logic flow from end to end. We will take a full examination at a typical Spring Web MVC programming logic flow using a typical workflow of creating a customer with SOBA as an example, after we get a chance next to see how Spring integrates standard view technologies with the Spring Web MVC Framework.

4.3.3 Spring Integration with View Technologies

Standard view technologies such as JSP, JSTL, Tiles (an Apache web template system), and so on, are separate from the Spring Web MVC framework. However, they are fully integrated into the Spring Web MVC Framework to enable a Spring-based enterprise web application to work seamlessly. In this section, we take a look at how Spring resolves views with those standard view technologies.

As its name implies, Spring resolves views with *View Resolvers*. However, a developer needs to know a lot more about the view technologies than about the Spring View Resolvers, which are easy to configure and just work behind the scene, as is discussed next.

The two most important Spring view interfaces are ViewResolver and View. The ViewResolver interface is a mapping mechanism for resolving a view name to an actual view, while the `View` interface prepares and hands requests over to a matched view technology. The two most commonly used Spring view resolvers are the `InternalResourceViewResolver` and the `ResourceBundleViewResolver`.

A view resolving process is typically defined in the `<app-name>-servlet.xml` file. To facilitate the discussion, let us look at how view resolving is defined for SOBA in the soba-`servlet.xml` file, shown in Listing 4.2 below.

To support multiple representations of a resource, Spring provides the ContentNegotiatingViewResolver interface to help resolve a view based on the file extension or HTTP request `Accept` header that signifies the type of the resource. This interface does not resolve views itself, but instead delegates views to a list of view resolvers that are set using the bean property `ViewResolvers`. This is exactly how it works as is shown in Listing 4.2 for SOBA as follows:

- At the top level, the `ContentNegotiatingViewResolver` is defined for the media types of `html`, `xml`, and `json`.
- Then, the following *View Resolvers* are defined:
  - **UrlBasedViewResolver**. The `ContentNegotiatingViewResolver` sends requests to this `ViewResolver` or `JstlView` if the content type is `text/html` or the file extension is `'.jsp'`.
  - **BeanNameViewResolver**. This ViewResolver interprets a view name in the current application context defined in the XML configuration file of the executing `DispatcherServlet`. It resolves views based on the entries defined in the form of `<bean name="/xyz.htm" class="xyzController/>` as can be found in the `soba-servlet.xml` file shown later.
  - **MappingJacksonJsonView**. The `ContentNegotiatingViewResolver` sends requests to this ViewResolver if the content type is `application/json` or `application/javascript`. This ViewResolver parses the object tree into JSON as the output.

We will see many such examples with SOBA later.

Listing 4.2 ViewResolvers defined in soba-servlet.xml for SOBA

```
    <bean class =
"org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
            <property name="mediaTypes">
                <map>
                    <entry key="html" value="text/html" />
                    <entry key="xml" value="application/xml" />
                    <entry key="json" value="application/json" />
                </map>
            </property>
            <property name="viewResolvers">
                <list>
                    <bean id="viewResolver"
                class="org.springframework.web.servlet.view.UrlBasedViewResolver">
                            <property name="viewClass"
                                value="org.springframework.web.servlet.view.JstlView" />
                            <property name="prefix" value="/WEB-INF/jsp/" />
                            <property name="suffix" value=".jsp" />
                        </bean>
    <bean class =   "org.springframework.web.servlet.view.BeanNameViewResolver" />
                </list>
            </property>
```

```
        <property name="defaultViews">
            <list>
                <bean
class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
                    <property name="prefixJson" value="false" />
                </bean>
            </list>
        </property>
    </bean>
```

4.4 PROGRAMMING LOGIC FLOW WITH SPRING WEB MVC FRAMEWORK

First, let us demonstrate what a typical application workflow looks like when using Spring Web MVC to develop an enterprise Web application. The workflow we described for testing SOBA setup in Section 2.7 is re-usable for our purposes here, as is discussed next.

4.4.1 A Typical Workflow of SOBA Defined in Spring MVC Framework

To make it easier, Figure 4.5 is presented to demonstrate the workflow with the process of *creating a new customer* as described in Chapter 2 with a series of screenshots from Figure 2.8 through Figure 2.14. The steps involved include:

1 **User Gestures**. In this step, a user opens a Web page to initiate an action to interact with the application. In this use scenario, a user opens up the home page of the application.
2 **Requests Sent to the Spring Dispatcher Servlet**. The gesture of a user is formatted into an HTTP request and submitted to the system. The request is then routed to the Spring dispatching servlet.
3 **Requests Dispatched to the Application Controller.** The Spring dispatching servlet automatically sends the requests to the corresponding application controller, based on the handler mappings set up in an external XML configuration file that will be discussed shortly.
4 **User Data Validated by Calling a Validator**. It's very rare that a request would be sent directly to the corresponding service without being validated. This step must be followed rigorously in a real enterprise application, because the application must distinguish between valid and invalid user input data. This is what a validator would be responsible for.
5 **The Corresponding Service Gets Invoked**. After data validation, the corresponding service is called to execute the tasks required for fulfilling user's requests. The service at this step typically makes DAO calls to initiate data related operations, as is discussed next.
6 **SQL Execution via a DAO**. The methods of a DAO class mainly consist of a standard set of SQL executions such as `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and so on.

7 **The Data Model Gets Updated**. As the result of the DAO operations as stated in the preceding step, the data model gets updated, which represents the intended operations by the user.
8 **The Result Set is Forwarded to Another Application Controller**. Typically, another application controller is invoked to handle the results returned from the service calls at step 5.
9 **Responses Rendered to a User View**. At this step, the results are reformatted and rendered to a user view to be consumed by the user.
10 **The User Gets the Updated View**. At this step, the user finally sees the responses routed back from the application.

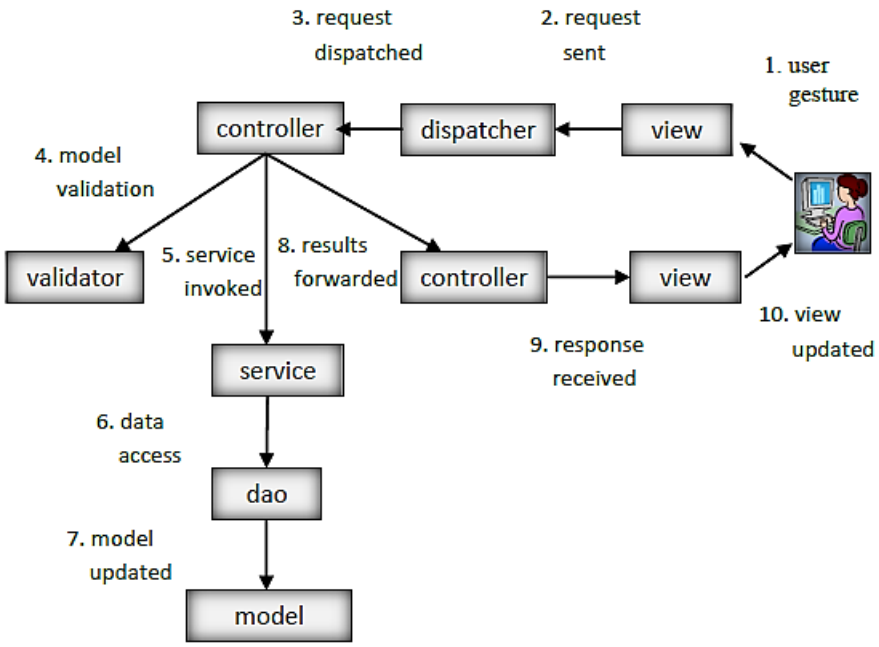

Figure 4.5 SOBA: Workflow for creating a new customer

The programming logic flow for the above workflow using the Spring Web MVC Framework begins with as simple as a login.jsp file, as discussed next.

4.4.2 A JSP Login Page Defining a Web Entry Point

To continue our discussion, note the `<welcome-file-list>` XML element in the previous `web.xml` file shown in Listing 4.1. This element specifies the home page of the application. In this case, the home page of SOBA can be invoked with either https://localhost:8443/soba implicitly or https://localhost:8443/soba/login.jsp explicitly. Either way, the login.jsp file is called. This file provides an entry point for SOBA.

Listing 4.3 shows the contents of the login.jsp file. Note the segment of "`<a href="<c:url value="createCustomerForm.htm"/>">` Open Now.," highlighted in

bold face. This is what would work behind the scene when you click the *Open Now* link as shown in Figure 2.8. This link would start the process of creating a new customer. The exact semantics of this segment is defined in the `jsp/jstl/core` tag library, as is indicated by the first line of this `index.jsp` file. We are less concerned with it now, but we would like to know the exact implication of the part of `createCustomerForm.htm`. We know that since it ends with .htm, it would be routed by the `DispatcherServlet`, according to what we have learnt from the `web.xml` file previously. But what destination will it be directed to by the `DispatcherServlet`? That is the subject of Handler Mapping and the answer lies in the `soba-servlet.xml` file to be discussed next.

Listing 4.3 login.jsp

```
    <%@ include file = "WEB-INF/jsp/include.jsp" %>
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <html>
    <head>
    <title>Login</title>
    </head>
    <%@ include file = "WEB-INF/jsp/banner.jsp" %>
    <script language="javascript">
    function focusOnUsername () {
        document.loginForm.j_username.focus();
    }
    </script>
    <body onLoad="focusOnUsername ()"> <center>
    <table> <tr>
    <td> Prospective Customers: <i> Don't have an account? </i></td>
    <td> <a href="<c:url value="createCustomerForm.htm"/>"> Open Now </a> </td>
<tr>
    <td> Established Customers: <i>Don't have a user ID or password? </i></td>
     <td> <a href="<c:url value="createLoginUserForm.htm"/>"> Register </a></td></tr>
    </tr></table>
    <hr>
       <br> <br>
    <form name="loginForm" method="POST" action="<c:url
value="/j_spring_security_check" />">
    <div align = "center">
    <table align="center" width="300" border="7" CELLPADDING="7"
CELLSPACING="10" BGCOLOR="#C6EFF7">
    <th colspan="2" bgcolor="#00184A"><FONT COLOR="#FFFFFF">Existing User Login
</FONT></th>
      <tr>
        <td >Username: </td>
        <td><input type="text" name="j_username" /></td>
      </tr>
      <tr>
        <td align="center">Password: </td>
```

```
    <td><input type="password" name="j_password" /></td>
   </tr>
   <tr>
    <td> <B><img src="/soba/images/arrow.jpg"/> </B></td><td>
     <select name="signInRole">
     <option value="customer" selected> An Established Customer</option>
     <option value="rep"> A Rep</option>
     <option value="admin"> A System Admin</option>
     </select>
     </td>
     </tr>
   <tr>
    <td colspan="2" align="center">
     <input type="submit" value="Login" />
     <input type="reset" value="Reset" />
    </td>
   </tr>
  </table>
  </form>
  </div>
</center>
  <%@ include file = "WEB-INF/jsp/showLoadTime.jsp" %>
  </body>
  </html>
```

4.4.3 Handler Mapping

Handler mappings are defined in the soba-servlet.xml file, which is shown in Listing 4.4 below. First, we see Spring beans defined in each `<bean ..> </bean>` XML element. Some beans are defined by id, and some beans are defined by name. By default, the `DispatcherServlet` uses `BeanNameUrlHandlerMapping` for its handler mapping task. Thus, a URL pattern like `manageTx.htm` will be mapped to the corresponding class `ManageTxController`; similarly, the URL of `createCustomerForm.htm` will be mapped to `CreateCustomerFormController` with `'Controller'` attached to its name before .htm. However, this default mapping rule can be simplified since Spring 2.5 by using annotation based configuration and auto-detection. Note the lines of `context:component-scan base-package=<..>` in the soba-servlet.xml file. These lines enable all annotated Spring beans to be auto-detected without having to specify the URL mapping in an XML configuration file like `soba-servlet.xml`.

Listing 4.4 soba-servlet.xml

```
  <?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.1.xsd">
        <!-- the application context definition for the soba DispatcherServlet -->
        <context:component-scan base-package="com.perfmath.spring.soba.web" />
        <context:component-scan base-package="com.perfmath.spring.soba.model" />
        <context:component-scan base-package="com.perfmath.spring.soba.service" />
        <context:component-scan base-package="com.perfmath.spring.soba.restservice"
/>
        <context:annotation-config />
        <bean id="myAuthenticationManager"
class="com.perfmath.spring.soba.util.MyAuthenticationManager">
        </bean>
        <bean id="messageSource"

    class="org.springframework.context.support.ResourceBundleMessageSource">
            <property name="basename" value="messages" />
        </bean>
        <bean name="/customerList.htm"
class="com.perfmath.spring.soba.web.CustomerController">
            <property name="customerManager" ref="customerManager" />
        </bean>
        <bean name="/accountList.htm"
class="com.perfmath.spring.soba.web.AccountController">
            <property name="accountManager" ref="accountManager" />
        </bean>
        <bean name="/transactionList.htm"
class="com.perfmath.spring.soba.web.TxController">
            <property name="txManager" ref="txManager" />
        </bean>
        <bean name="/manageTx.htm"
class="com.perfmath.spring.soba.web.ManageTxController">
            <property name="aclTxManager" ref="aclTxManager" />
        </bean>
        <bean name="/reverseTx.htm"
class="com.perfmath.spring.soba.web.ReverseTxController">
            <property name="aclTxManager" ref="aclTxManager" />
        </bean>
        <bean name="/disputeTx.htm"
class="com.perfmath.spring.soba.web.DisputeTxController">
            <property name="aclTxManager" ref="aclTxManager" />
        </bean>
        <!-- spring 3 restful begin -->
        <bean id="jaxbMarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
```

```
            <property name="classesToBeBound">
                <list>
                    <value>com.perfmath.spring.soba.model.domain.BankingTx</value>
                </list>
            </property>
        </bean>
        <bean id="restTxList"
            class="org.springframework.web.servlet.view.xml.MarshallingView">
            <constructor-arg ref="jaxbMarshaller" />
        </bean>
        <bean
        class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver"
>
            <property name="mediaTypes">
                <map>
                    <entry key="html" value="text/html" />
                    <entry key="xml" value="application/xml" />
                    <entry key="json" value="application/json" />
                </map>
            </property>
            <property name="viewResolvers">
                <list>
                    <bean id="viewResolver"

    class="org.springframework.web.servlet.view.UrlBasedViewResolver">
                        <property name="viewClass"
                            value="org.springframework.web.servlet.view.JstlView" />
                        <property name="prefix" value="/WEB-INF/jsp/" />
                        <property name="suffix" value=".jsp" />
                    </bean>
                    <bean
class="org.springframework.web.servlet.view.BeanNameViewResolver" />
                </list>
            </property>
            <property name="defaultViews">
                <list>
                    <bean

    class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
                        <property name="prefixJson" value="false" />
                    </bean>
                </list>
            </property>
        </bean>
    </beans>
```

Now let's come back to the login.jsp file discussed in the preceding section. Note again the segment highlighted in bold face of "<a href = "<c:url value="createCustomerForm.htm"/>"> Open Now.". As we clarified in the beginning of this section, the URL pattern createCustomerForm.htm will be mapped to CreateCustomerFormController by default if it's not specified explicitly in the configuration file. This is possible only if the Spring bean coded in CreateCustomerFormController.java is annotated properly, as is discussed next.

4.4.4 Implementing Spring Controllers with Annotations

Handler Controllers are configured externally and so are view resolutions, providing developers with options for building flexible, modularized applications. In particular, since Spring 2.5, the default handler is based on the @Controller and @RequestMapping annotations specified in the Java code implementing the handler. Annotation has become more and more popular since Java 5, providing an extra dimension for flexibility. Another benefit of the annotation feature is to allow building applications based on RESTful Web Services painlessly, thanks to the method argument level annotation through the @PathVaribale annotation and other features. We'll see such examples with SOBA later.

To see how Spring Controller is implemented with the help of Spring's annotation feature, Listing 4.5 shows the actual code of the controller class of CreateCustomerFormController.java defined in the package of com.perfmath.spring.soba.web.

Listing 4.5 CreateCustomerFormController.java

```
package com.perfmath.spring.soba.web;

import java.sql.Timestamp;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import com.perfmath.spring.soba.model.domain.Customer;
import com.perfmath.spring.soba.service.CreateCustomerValidator;
import com.perfmath.spring.soba.service.CustomerManager;
import com.perfmath.spring.soba.util.RandomID;
```

```
@Controller
@RequestMapping("/createCustomerForm")
@SessionAttributes("customer")
public class CreateCustomerFormController {
    private CreateCustomerValidator validator;
    private CustomerManager customerManager;
    @Autowired
    public CreateCustomerFormController (CustomerManager customerManager,
            CreateCustomerValidator validator) {
        this.customerManager = customerManager;
        this.validator = validator;
    }
    @RequestMapping(method = RequestMethod.GET)
    public String setupForm (
            @RequestParam(required = false, value = "username") String username,
            Model model) {
        Customer customer = new Customer();
        model.addAttribute("customer", customer);
        return "createCustomerForm";
    }
    @RequestMapping(method = RequestMethod.POST)
    public String submitForm (
            @ModelAttribute("customer") Customer customer,
            BindingResult result, SessionStatus status) {
        validator.validate(customer, result);

     if (result.hasErrors()) {
            return "createCustomerForm";
        } else {
            customerManager.createCustomer(customer);
            status.setComplete();
            return "redirect:createCustomerSuccess/customerId/" +
                customer.getCustomerId();
        }
    }
}
```

By examining the above CreateCustomerFormController.java file, we notice the following annotations:

- **@Controller**. This annotation indicates that the annotated class plays the role of a controller. In this case, the controller class does not have to extend any controller base class or reference the Servlet API. We can also say that the @Controller annotation acts as a stereotype for the annotated class, indicating its role (in UML vocabulary, a *stereotype* is an extension mechanism for defining a new kind of model element based on an existing model element. It is expressed by placing its name as a string around a pair of angle brackets or *guillemets* in French, for example,

<<*StereoType*>> . So a class with the stereotype <<*Controller*>> is read as "a class of the Controller stereotype." The particular characteristics a Controller class must have are defined when the stereotype is defined. Also, note in Java we use @ instead of *guillemets* to define stereotypes). The annotated beans can be defined explicitly in a configuration file using the URL mapping mechanism. However, they can be more conveniently auto-detected or scanned if it belongs to one of those packages specified in the `<context:component-scan base-package=<..>` XML element. In particular, the controller `CreateCustomerFormController` in the package of `com.perfmath.odps.soba.web` is auto-scanned when the application starts up .

- **@RequestMapping**. This mapping is used to map URLs onto an entire class or a particular handler method. Typically, the class-level annotation maps a specific request path or path pattern onto a form controller, for example, the URL `/createCustomerForm` is mapped to the form controller of `CreateCustomerFormController`. Also, note those `RequestMappings` associated with `HTTP GET` and `POST` methods in Listing 4.5.
- **@SessionAttributes.** This annotation declares session attributes used by a specific handler. It typically lists the names of model attributes that should be maintained in the session, serving as form-backing beans between subsequent requests. In this case, the session attribute defined is customer.
- **@Autowired.** This annotation auto-wires the class with its dependent classes. For example, the class `CreateCustomerFormController` depends on two other classes: `CustomerManager` and `CreateCustomerValidator`. In this case, it's equivalent to the property element of a bean definition explicitly specified in its associated configuration file.
- **@RequestParam.** This annotation binds the annotated parameter to the corresponding HTTP request parameter if it exists.
- **@ModelAttribute.** This annotation provides a link to data in the model. When used with the submitForm method of a controller, this annotation binds the specified model attribute to the parameter following it. This is how the controller gets a reference to the data entered in the form.
- **@PathVariable.** This annotation binds a method parameter with the value of a URI template variable. We don't see this annotation here, but we'll see such examples with the SOBA classes that implement RESTful Web Services later.

Note that the form controller `CreateCustomerFormController` has two methods: setupForm and submitForm. When a URL that contains the destination to this form controller as embedded in the login.jsp file is clicked, control is routed to the `DispatcherServlet` that routes control to this form controller based on the URL mapping it knows about. Then the setupForm method of this form controller is invoked first. This is where you can pre-populate some of the entries of the form before control is turned over to the form of `createCustomerForm.jsp` as specified in the return statement of the `setupForm` method.

After a user enters all required entries on the form and clicks the *Submit* button, control is returned to the form controller, and the validator is invoked to validate the data entered onto the form. This is another point of time that you can decide how you want to set some of the entries on the form and how you want to validate the data entered on the form (in

this sample implementation, validation logic is only for illustrative purposes. Validation should be beefed up significantly in a real application). Listing 4.6 shows the implementation of the `CreateCustomerFormValidator` class. Note a few things:

- This validator class is annotated with `@Component`, which is a generic stereotype for any Spring-managed component. Other annotations such as `@Controller`, `@Service`, and `@Repository` are specializations of `@Component` for more specific uses, for example, in the presentation, service, and persistence layers.
- In addition to using `ValidationUtils`, you can implement your own validation classes or methods to be used here. Section 5.6 introduces more about Spring data validation using the SOBA `BillPayment` service as an example.

Listing 4.6 CreateCustomerFormValidator.java

```
package com.perfmath.spring.soba.service;

import java.sql.Timestamp;
import java.util.Calendar;
import java.util.Date;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import org.springframework.stereotype.Component;

import com.perfmath.spring.soba.model.domain.Customer;
import com.perfmath.spring.soba.util.RandomID;

import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.*;
@Component
public class CreateCustomerValidator implements Validator {

   public boolean supports(Class clazz) {
      return Customer.class.isAssignableFrom(clazz);
   }

   public void validate(Object target, Errors errors) {
     Object principal =
     SecurityContextHolder.getContext().getAuthentication().getPrincipal();
     if (principal instanceof UserDetails) {
          UserDetails ud = (UserDetails)principal;
          String username = ud.getUsername();
          System.out.println ("current user name:" + username);
          if (ud.isEnabled()){
          System.out.println (" {current user is enabled:");
```

```
            } else {
                System.out.println (" {current user is not enabled:");
            }
        } else {
            String username = principal.toString();
            System.out.println ("current user details:" + username);
        }

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
              "required.firstName", "firstName is required.");
        ValidationUtils.rejectIfEmpty(errors, "lastName",
              "required.lastName", "lastName is required.");
        ValidationUtils.rejectIfEmpty(errors, "phone",
              "required.phone", "phone is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "address",
              "required.address", "address is required.");
        ValidationUtils.rejectIfEmpty(errors, "city",
              "required.city", "city is required.");
        ValidationUtils.rejectIfEmpty(errors, "state",
              "required.state", "state is required.");

        Customer customer = (Customer) target;
            customer.setCustomerId((new RandomID(9)).getId());
            customer.setStatus(0);
            customer.setCreateDate(new Timestamp(System.currentTimeMillis()));
            String state = customer.getState();
        if (state.length() != 2) {
            errors.reject("invalid.stateNameLength", "State name must be two letters.");
        }
    }
}
```

If the form data validation is passed, the createCustomer service is called, a new customer would be created if everything goes well, and control is turned over to the createCustomerSuccess.jsp file that we'll take a look after we look at the createCustomerForm.jsp file next. However, before we move to the next section, you might want to take a look at the return statement of the submitForm method of the CreateCustomerFormController class, which is called out below:

```
return "redirect:createCustomerSuccess/customerId/" + customer.getCustomerId();
```

This is in the form of "…/customerId/{customerId}" where the part {customerId} represents the actual value of a customerId. This is called a *URI Template Pattern* in Spring and we'll talk more about it later.

4.4.5 A Typical View Defined in a Spring MVC Web Form

The createCustomerForm.jsp file shown in Listing 4.7 below illustrates a typical view defined in a Spring MVC Web form (we consider a jsp file a Spring MVC Web form if it defines a form that uses the Spring jsp tag library). Consult *Appendix G spring-form.tld* of the *Reference Documentation for Spring 3.1* for valid Spring form entries. Next, let's go over those lines highlighted in bold face.

Listing 4.7 createCustomerForm.jsp

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
</head>
<%@ include file = "banner.jsp" %>
 <style>
.error { color: red; }
</style> </head>
<body> <center>
<h1><fmt:message key="createcustomer.heading"/></h1>
<div align = "center">
<form:form method="post" commandName="customer">
 <table align ="center" width="600" bgcolor="#94D6E7" border="3"
cellspacing="10" cellpadding="2">
      <tr>
       <td align="right" width="100">First Name:</td>
        <td width="100">
         <form:input path="firstName"/>
        </td>
        <td width="400">
         <form:errors path="firstName" cssClass="error"/>
        </td>
      </tr>
       <tr>
       <td align="right" width="100">Last Name:</td>
        <td width="100">
         <form:input path="lastName"/>
        </td>
        <td width="400">
         <form:errors path="lastName" cssClass="error"/>
        </td>
      </tr>
       <tr>
       <td align="right" width="100">Phone:</td>
```

```
    <td width="100">
     <form:input path="phone"/>
    </td>
    <td width="400">
     <form:errors path="phone" cssClass="error"/>
    </td>
  </tr>
  <tr>
   <td align="right" width="100">Address:</td>
    <td width="100">
     <form:input path="address"/>
    </td>
    <td width="400">
     <form:errors path="address" cssClass="error"/>
    </td>
  </tr>
  <tr>
   <td align="right" width="100">City:</td>
    <td width="100">
     <form:input path="city"/>
    </td>
    <td width="400">
     <form:errors path="city" cssClass="error"/>
    </td>
  </tr>
  <tr>
   <td align="right" width="100">State:</td>
    <td width="100">
     <form:input path="state"/>
    </td>
    <td width="400">
     <form:errors path="state" cssClass="error"/>
    </td>
  </tr>
  <tr>
   <td align="right" width="100">Zipcode:</td>
    <td width="100">
     <form:input path="zipcode"/>
    </td>
    <td width="400">
     <form:errors path="zipcode" cssClass="error"/>
    </td>
  </tr>
  <tr align="center">
   <td align="right" width="100">Email:</td>
    <td width="100">
```

```
        <form:input path="email"/>
       </td>
       <td width="400">
        <form:errors path="email" cssClass="error"/>
       </td>
     </tr>
    </table>
    <br>
   <input type="submit" value="Submit">
   </form:form>
    </div>
    <%@ include file = "showLoadTime.jsp" %>
   </body>
   </center>
   </html>
```

The first line defines an `include.jsp` file, which contains all common needs shared among all jsp files. The contents of the `include.jsp` file are shown as follows. Note that the first line specifies that page session is not maintained, which is a common jsp performance and scalability practice. The next four lines specify the various jsp tag libraries to be used. The last line defines a JAVA statement that saves the begin time of the jsp file. When used with the line containing showLoadTime.jsp at the bottom of the `createCustomerForm.jsp`, the total elapsed time associated with this form can be timed and displayed to the user. Since both the begin and end timing calls are made on the server, the measured elapsed time is the time spent on the server only, namely, the network latency between the server and the client is not accounted for.

```
<%@ page session="false"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<%@ taglib prefix="security"
uri="http://www.springframework.org/security/tags" %>

<%@ page language="java" contentType="text/html;charset=UTF-8" %>

<% long beginPageLoadTime = System.currentTimeMillis();%>
```

Next, note the line containing `fmt:message` in Listing 4.7. This provides an option for specifying text output defined in an external file. For SOBA, this file is named messages.properties and stored at the root classpath of `/WEB-INF/classes`. The contents of this file are shown below.

```
title=SOBA (Safe Online Banking Application)
heading=SOBA :: Safe Online Banking Application
greeting=Greetings, it is now
createcustomer.heading=SOBA :: create a new customer
```

```
createloginuser.heading=SOBA :: create a new login user
createaccount.heading=SOBA :: create a new account
createtx.heading=SOBA :: Post a transaction to an account
required=Entry required.
typeMismatch=Invalid data.
```

Note the line `<form:form method="post" commandName = "customer">` shown in the above jsp file. That line defines that the HTTP method to be used to send the form to the `CreateCustomerFormController` would be `POST`, and that the command object to be invoked would be a customer object, which is defined in the `Customer.java` file in the `domain` package of SOBA. For the implementation of the Customer class, see Listing 4.8 below.

Listing 4.8 Customer.java

```
package com.perfmath.spring.soba.model.domain;

import java.io.Serializable;
import java.sql.Timestamp;

public class Customer implements Serializable {
    private String customerId;
    private String firstName;
    private String lastName;
    private String phone;
    private String address;
    private String city;
    private String state;
    private String zipcode;
    private String email;
    private int status;
    private Timestamp createDate;

// getters/setters are omitted to save space

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(" customerId: " + customerId + ";");
        buffer.append(" firstName: " + firstName);
        buffer.append(" lastName: " + lastName);
        buffer.append(" phone: " + phone);
        buffer.append(" address: " + address);
        buffer.append(" city: " + city);
        buffer.append(" state: " + state);
        buffer.append(" zipcode: " + zipcode);
        buffer.append(" email: " + email);
        buffer.append(" status: " + status);
```

```
            buffer.append(" createDate: " + createDate);
            return buffer.toString();
        }
    }
```

The above `createCustomerForm.jsp` form contains many lines like, for example, `<form:input path = "firstName">`. There is a one-to-one corresponding relationship between a path defined on the form and the property of the class to be targeted. This is how a form and a domain class gets associated with each other. Also note that for each line of `<form:input …>` there is a corresponding line of `<form:errors …>`, which associates the entry with the errors found during validation.

Finally, the line containing "`Submit`" in `createCustomerForm.jsp` defines the exit of this jsp file, which returns control to the form's form controller `CreateCustomerFormController`. After the transaction of creating a new customer is completed successfully, control is returned to the `CreateCustomerSuccessController`, which is discussed next.

4.4.6 A Typical Form Success Controller with Spring ModelAndView

Listing 4.9 shows a typical form success controller associated with the transaction of creating a new customer. Note the following specific details in this controller:

- It is annotated with the `@Controller` annotation.
- It is annotated with an `@RequestMapping` annotation with its value consistent with what is specified in the return statement of the submitForm method of the `CreateCustomerFormController` class.
- The object returned from this class and directed to `DispatcherServlet` is a Spring `ModelAndView` object. ModelAndView is a Spring Web MVC Framework class for holding both model and view in one object. It has six variants based on what arguments to pass in when it is created, but in our case here, it uses the constructor of `ModelAndView (String viewName, String modelName, Object modelObject)`, with `viewName = "createCustomerSuccess"`, `modelName="model"`, and `modelObject = myModel` of type `Map<String, Object>`. The view part specifies where control should be routed to, and the model part provides the convenience of accessing the model data in the `view`.

As stated above, the `createCustomerSuccessController` returns a `ModelAndView` object with the return URL of `createCustomerSuccess`, which is routed to `createCustomerSuccess.jsp` as shown in Listing 4.10. This jsp file displays a message showing that the transaction is completed successfully. It then waits for the user to initiate the next transaction with a link embedded that is mapped to another controller using the same mapping mechanism discussed previously. Note how the model object is used in `createCustomerSuccess.jsp` to access the data item customerId.

Listing 4.9 CreateCustomerSuccessController.java

```
package com.perfmath.spring.soba.web;
```

```
import java.util.HashMap;
import java.util.Map;

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

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.servlet.ModelAndView;
@Controller
public class CreateCustomerSuccessController {
   @RequestMapping(value="/createCustomerSuccess/customerId/{customerId}",
    method=RequestMethod.GET)
   public ModelAndView createCustomerSuccess(@PathVariable("customerId")
String
    customerId) {
    Map<String, Object> myModel = new HashMap<String, Object>();
    myModel.put("customerId", customerId);
    return new ModelAndView("createCustomerSuccess", "model", myModel);
   }
}
```

Listing 4.10 createCustomerSuccess.jsp

```
<%@ include file="/WEB-INF/jsp/include.jsp"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
<%@ include file="banner.jsp"%>
<title>Create Customer Success</title>
</head>
<body>
    <center>
        Your customer ID
        <c:out value="${model.customerId}" />
        has been created successfully. <br> <br> Use your customer
        ID to <a
            href="<c:url
value="/createLoginUserForm/customerId/${model.customerId}"/>">
            create</a> your login ID for banking online. <br> <br>
</center>
</body>
</html>
```

Figure 4.6 summarizes the programming logic flow for creating a customer using the Spring Web MVC framework. The transitional URI's are included at each step. How these transitions actually happen are specified in the `web.xml` and `soba-servlet.xml` file. It might be helpful that you trace this flow one more time by following the detailed descriptions we have provided so far in this section.

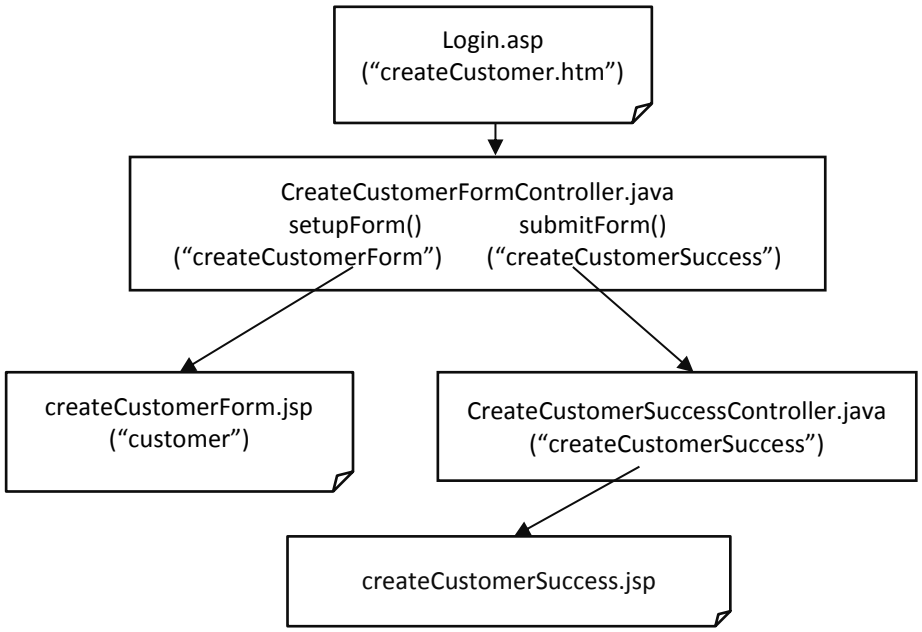

Figure 4.6 Programming logic flow for creating a SOBA customer

Next, we'll dig deeper into how controllers collaborate with POJOs to complete the task of creating a customer in SOBA.

4.4.7 POJOs Referenced in the CreateCustomerFormController

Let's take a look at Listing 4.5 for the `CreateCustomerFormController` again. After a customer domain object is successfully constructed, in the submit method of the `CreateCustomerFormController`, the `createCustomer` method of the `customerManager` class is called to create the customer passed in. Actually, the `customerManager` is only an interface, and its implementation is realized in `SimpleCustomerManager.java`. Listings 4.11 (a) and (b) show how the `CustomerManager` interface is defined in `CustomerManager.java` and how it is implemented in `SimpleCustomerManager.java`, respectively. Note that `customerManager` and `SimpleCustomerManager` are pure POJOs, as by definition of a POJO they don't use any vendor-specific packages. Since these POJOs are ordinary Java objects and what they do are fairly self-explanatory, a more detailed explanation is omitted here. However, I do need to point out that such POJOs need to be defined in an

external configuration file, and they are indeed defined in the `soba-services.xml` file. I will remind you of this when we discuss the subject of *Data Access* in the next chapter.

Listing 4.11 (a) CustomerManager.java

```
package com.perfmath.odps.soba.service;
import java.io.Serializable;
import java.util.List;
import com.perfmath.odps.soba.model.domain.Customer;

      public interface CustomerManager extends Serializable{
   public void createCustomer(Customer customer);
   public List<Customer> getCustomers();

}
```

Listing 4.11 (b) SimpleCustomerManager.java

```
      package com.perfmath.odps.soba.service;

import java.util.List;
import com.perfmath.odps.soba.model.dao.CustomerDao;
import com.perfmath.odps.soba.model.domain.Customer;

      public class SimpleCustomerManager implements CustomerManager {
   private CustomerDao customerDao;
   public List<Customer> getCustomers() {
      return customerDao.getCustomerList();
   }

   public void createCustomer(Customer customer) {
     customerDao.insert(customer);
   }
   public void setCustomerDao(CustomerDao customerDao) {
      this.customerDao = customerDao;
   }
}
```

You might have noticed that `CustomerManager` depends on `CustomerDao` to persist a customer object into the database. This kind of data access can be implemented either in JDBC or in some object-relational mapping (ORM) frameworks such as Hibernate. This is one of the most important layers in building enterprise applications, and we will cover it in detail in the next chapter.

4.5 SUMMARY

In this chapter, we introduced the Spring Web MVC Framework and demonstrated how it is used for building SOBA with a simple workflow of creating a customer. The most important thing is to understand the programming logic flow that needs to be followed when developing an enterprise Web application with the Spring Web MVC Framework. Here is a recap of this programming logic flow we demonstrated in this chapter:

- Spring Web MVC Framework's DispatcherServlet inherits from the standard HttpServlet base class. The implication with this is that you can run Spring-based web applications on popular web servers or servlet engines like Tomcat. In this case, your web app is configured with a web.xml file, in which you specify a start page such as `login.jsp` among other configurations.
- It's important to understand how URLs are mapped to handlers or controllers with the Spring Web MVC Framework. This is specified in the <app-name>-servlet.xml file, e.g., the soba-servlet.xml file as you can find in your SOBA setup on Eclipse. One important convention to keep in mind is that if you have an `xyzForm.jsp`, then it is implied that it will be mapped to the xyzFormController controller. In SOBA, the pair of createCustomerForm.jsp and CreateCustomerFormController.java is one of such examples.
- Since web applications heavily depend on forms to work, it's crucial to understand how Spring form controllers work. This includes not only how a controller should be properly annotated, but also how its setupForm and submitForm methods work. We certainly have seen this with the CreateCustomerFormController.java class.
- Paired with a form controller, typically a form success controller is used to communicate the final status to the user via a `view`, for example, a jsp page. We demonstrated this with the CreateCustomerSuccessController.java class and the createCustomerSuccess.jsp file. A very important Spring specific technique here is the Spring `ModelAndView` class, which wraps `view` and `model` in one object so that `view` can consume `model` information within itself.

We certainly have not covered all bits of the Spring Web MVC Framework in this chapter, especially on some more elaborate annotations. Refer to Spring's reference documentation if you need to learn more.

RECOMMENDED READING

Study Chapter 16 of the Spring Reference Documentation 3.1 to have a more thorough understanding on the Spring MVC framework. Here is a complete list of the sections about the Spring MVC framework introduced there (highlighted subjects are covered in this chapter).

Spring Reference Documentation 3.1/ Chapter16:

16.1 Introduction to Spring Web MVC framework

16.2 The DispatcherServlet

16.3 Implementing Controllers

16.4 Handler mappings

16.5 Resolving views

16.6 Using flash attributes

16.7 Building URIs

16.8 Using locales

16.9 Using themes

16.10 Spring's multipart (file upload) support

16.11 Handling exceptions

16.12 Convention over configuration support

16.13 ETag support

16.14 Configuring Spring MVC

It is particularly worthwhile to try to understand other return types in addition to *ModelAndView* and *String* that we covered in this chapter. Refer to page 456 of the Spring Reference Documentation 3.1 for a brief description of each return type that Spring 3.1 supports.

You may also want to review *Chapter 17 View Technologies*, especially the JSP and JSTL section that we use a lot with SOBA.

SELF-REVIEW EXERCISES

4.1 Is MVC a generic architecture or specific to Spring only?

4.2 What's the purpose of the ApplicationContext interface? Which configuration XML file is for loading application contexts?

4.3 How do you configure your Spring-based application so that it would support Jackson Json view?

4.4 Which configuration file defines Spring handler mappings?

4.5 Which one is more heavily annotated, a Controller or a Validator? Why?

4.6 Identify all annotations in Listing 4.5 and explain how they are used. Especially, what is the purpose of the *Autowired* annotation?

4.7 What's the purpose of Spring context component auto-scan? What are the prerequisites in order to have a Spring component auto-scanned?

4.8 What are the purposes of the *setupForm* and *submitForm* method with a Spring *Form Controller*? At what point are they initiated?

4.9 How many Controller return types are defined in Spring 3.1? Which ones are most commonly used?