7

The Spring Web MVC Framework

In the previous chapter, I gave you an overview of the Spring Framework. We looked at what Spring is, how it is packaged, and the various modules it contains. I also mentioned that with Spring, you do not have to take an all-or-nothing approach when trying to decide whether you should use Spring. In other words, based on your needs, you can phase in the Spring Framework one module at a time (along with any dependencies). In this chapter, I will demonstrate how to use Spring Web MVC Framework (module), to build Time Expression, our sample web application.

Note that from this point on, I will refer to the Spring Web MVC Framework as simply Spring MVC, in most places.

What's Covered in This Chapter

In this chapter, we will

- Look at the various benefits of using Spring MVC
- Take an in-depth look at the Spring Web MVC Framework
- Build three of the screens in Time Expression using Spring MVC: a no-form controller, two form controllers, and a Spring HTTP interceptor.

> **Note**
> The complete code for the examples used in this chapter can be found within this book's code zip file (available on the book's website).

This is an exciting chapter, so I won't waste any more time boring you with introductory material. Let's spring into action!

Benefits of the Spring Web MVC Framework

The Spring Web MVC Framework is a robust, flexible, and well-designed framework for rapidly developing web applications using the MVC design pattern. The benefits achieved from using this Spring module are similar to those you get from the rest of the Spring Framework. Let's review a few of these. I will demonstrate some of these benefits later in this chapter.

- Easier testing—This is a common theme you will find across all the Spring classes. The fact that most of Spring's classes are designed as JavaBeans enables you to inject test data using the setter methods of these classes. Spring also provides mock classes to simulate Java HTTP objects (`HttpServletRequest`, for example), which makes unit testing of the web layer much simpler.
- Bind directly to business objects—Spring MVC does not require your business (model) classes to extend any special classes; this enables you to reuse your business objects by binding them directly to the HTML forms fields. In fact, your

controller classes are the only ones that are required to extend Spring classes (or implement a Spring controller interface).

- Clear separation of roles—Spring MVC nicely separates the roles played by the various components that make up this web framework. For example, when we discuss concepts such as controllers, command objects, and validators, you will begin to see how each component plays a distinct role.
- Adaptable controllers—If your application does not require an HTML form, you can write a simpler version of a Spring controller that does need all the extra components required for form controllers. In fact, Spring provides several types of controllers, each serving a different purpose. For example, there are no-form controllers, simple form controllers, wizardlike form controllers, views with no controllers, and even prepackaged controllers that enable you to write views without your own custom controller.
- Simple but powerful tag library—Spring's tag library is small, straightforward, but powerful. For example, Spring uses the JSP expression language (EL) for arguments to the `<spring:bind>` tag.
- Web Flow—This module is a subproject and is not bundled with the Spring core distribution. It is built on top of Spring MVC and adds the capability to easily write wizardlike web applications that span across several HTTP requests (an online shopping cart, for example).
- View technologies and web frameworks—Although we are using JSP as our view technology, Spring supports other view technologies as well, such as Apache Velocity (jakarta.apache.org/velocity/) and FreeMarker (freemarker.org). This is a powerful concept because switching from JSP to Velocity is a matter of configuration. Furthermore, Spring provides integration support for Apache Struts (struts.apache.org), Apache Tapestry (jakarta.apache.org/tapestry), and OpenSymphony's WebWork (opensymphony.com/webwork/).
- Lighter-weight environment—As I mentioned in the previous chapter, Spring enables you to build enterprise-ready applications using POJOs; the environment setup can be simpler and less expensive because you could develop and deploy your application using a lighter-weight servlet container.

Spring Web MVC Concepts

The world of Java has seen many MVC design pattern-based web frameworks crop up in the past few years (several are listed at the very end of this chapter). MVC was originally conceived at XEROX PARC around the 1978–79 time frame and was later implemented in the Smalltalk-80 class library (also at XEROX PARC). It is a relatively simple concept to grasp and provides for a clean separation of presentation and data, as I'll explain briefly here.

First, let's look at our architecture diagram established earlier in the book and shown here in Figure 7.1.

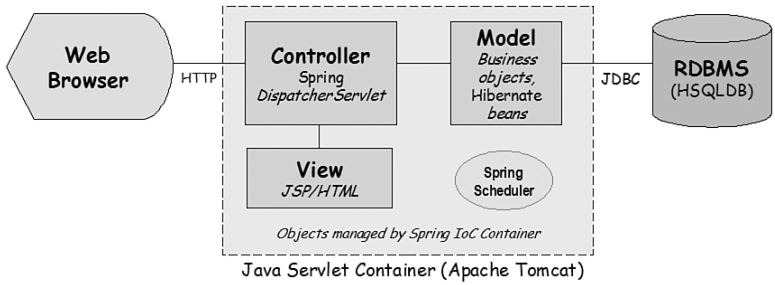

Figure 7.1 High-level architecture diagram for Time Expression.

As you can see, all incoming HTTP requests from a web browser are handled by Controllers. A *controller*, as the name indicates, controls the view and model by facilitating data exchange between them. The key benefit of this approach is that the model can worry only about the data and has no knowledge of the view. The view, on the other hand, has no knowledge of the model and business logic and simply renders the data passed to it (as a web page, in our case). The MVC pattern also allows us to change the view without having to change the model.

Let's review some basic Spring MVC concepts. First, we will look at the concepts related to Java coding, and then we will look at the configuration required to make all this work.

Spring MVC Java Concepts

Figure 7.1 provided us a high-level view of the architecture for Time Expression. Now let's take a slightly more detailed and focused look at the Spring MVC components. Figure 7.2 shows an end-to-end flow for a typical screen in Time Expression. This diagram shows many of the concepts we will discuss next.

Controller

Spring provides many types of controllers. This can be both good and bad. The good thing is that you have a variety of controllers to choose from, but that also happens to be the bad part because it can be a bit confusing at first about which one to use.

The best way to decide which controller type to use probably is by knowing what type of functionality you need. For example, do your screens contain a form? Do you need wizardlike functionality? Do you just want to redirect to a JSP page and have no controller at all? These are the types of questions you will need to ask yourself to help you narrow down the choices.

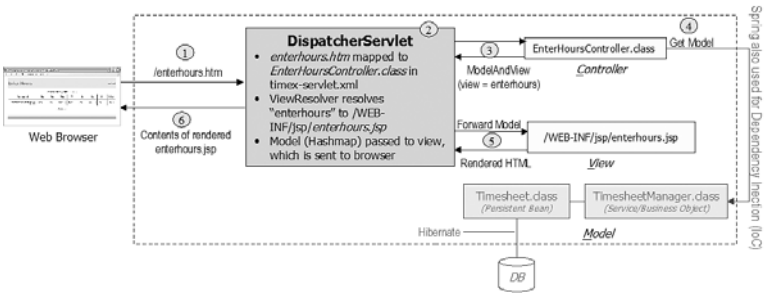

Figure 7.2 End-to-end flow for Enter Hours screen using Spring and Hibernate.

Figure 7.3 shows a class diagram of some of the more interesting controllers that are part of Spring MVC. Table 7.1 provides brief descriptions on the interface and classes shown in Figure 7.3. (Note: The descriptions provided in this table are taken directly out of the Spring Framework Javadocs.) I tend to use `SimpleFormController`, `UrlFilenameViewController`, and `AbstractController` most often. We will see examples of these later in this chapter.

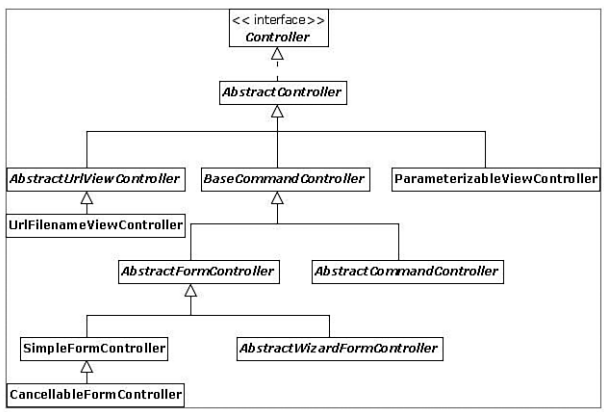

Figure 7.3 Class diagram showing a partial list of Spring controllers.

Table 7.1 **Description of Various Spring Controllers**

Controller	Description (Taken Directly from the Spring Javadocs)
AbstractCommandController	Abstract base class for custom command controllers.
AbstractController	Convenient superclass for controller implementations, using the Template Method design pattern.

Table 7.1 **Continued**

Controller	Description (Taken Directly from the Spring Javadocs)
AbstractFormController	Form controller that autopopulates a form bean from the request.
AbstractUrlViewController	Abstract base class for Controllers that return a view name based on the URL.
AbstractWizardFormController	Form controller for typical wizard-style workflows.
BaseCommandController	Controller implementation that creates an object (the command object) on receipt of a request and attempts to populate this object with request parameters.
CancellableFormController	Extension of SimpleFormController that supports "cancellation" of form processing.
Controller	Base Controller interface, representing a component that receives HttpServletRequest and HttpServletResponse like a HttpServlet but is able to participate in an MVC workflow.
ParameterizableViewController	Trivial controller that always returns a named view.
SimpleFormController	Concrete FormController implementation that provides configurable form and success views, and an onSubmit chain for convenient overriding.
UrlFilenameViewController	Controller that transforms the virtual filename at the end of a URL into a view name and returns that view.

Model and View

Many of the methods in the Controller related subclasses return a `org.springframework.web.servlet.ModelAndView` object. This object holds the model (as a `java.util.Map` object) and view name and makes it possible to return both in one return value from a method. We will see examples of this later in this chapter when we build two of the screens for Time Expression.

Command (Form Backing) Object

Spring uses the notion of a command object, which essentially is a JavaBean style class that gets populated with the data from an HTML form's fields. This same object is also passed to our validators (discussed next) for data validation, and if the validations pass, it is passed to the `onSubmit` method (in controller related classes) for processing of valid data. Given that this command object is a simple JavaBean-style class, we can use our business objects directly for data binding instead of writing special classes just for data binding. I will demonstrate this benefit later in this chapter.

Validator

A Spring validator is an optional class that can be invoked for validating form data for a given command (form) controller. This validator class is a concrete class that implements the `org.springframework.validation.Validator` interface. One of the two methods required by this interface is the `validate` method, which is passed a `command` object, as mentioned previously, and an `Errors` object, which can be used to return errors. I will demonstrate an example of a Validator class later in this chapter. Another notable validation class is `org.springframework.validation.ValidationUtils`, which provides convenient methods for rejecting empty fields.

Spring Tag Library (`spring:bind`)

The spring bind tag library is simple yet powerful. It is typically used in JSP files via the `<spring:bind>` tag, which essentially binds HTML form fields to the `command` object. Furthermore, it provides access to special variables within JSP, such as `${status.value}`, `${status.expression}`, and `${status.errorMessages}`, which we will look at later in the chapter.

Spring MVC Configuration Concepts

In this section, we will review some core concepts related to configuring the Spring Web MVC Framework.

`DispatcherServlet`

`DispatcherServlet` (part of the `org.springframework.web.servlet` package) is the entry point to the world of Spring Web MVC, as depicted in Figure 7.2. It essentially dispatches requests to the controllers. If you have worked with Java web applications before, you will not be surprised to find out that this class is configured in the `web.xml` file, as shown in the following excerpt from the complete `web.xml` for Time Expression:

```
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
```

We will discuss `DispatcherServlet` in detail later in this chapter.

Handler Mappings

You can map handlers for incoming HTTP requests in the Spring application context file. These handlers are typically controllers that are mapped to partial or complete URLs of incoming requests. The handler mappings can also contain optional interceptors, which are invoked before and after the handler. This is a powerful concept. I will demonstrate an example of this later in this chapter when we use such a web interceptor for authentication and close our Hibernate session for the given HTTP request.

The following code excerpt taken from our complete `timex-servlet.xml` file shows how a handler can be mapped to a partial URL:

```
<bean id="urlMap"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
        <props>
            <prop key="/signin.htm">signInController</prop>
            <prop key="/signout.htm">signOutController</prop>
        </props>
    </property>
</bean>
```

View Resolvers

Spring uses the notion of view resolvers, which resolve view names to the actual views (`enterhours` to `enterhours.jsp`, for example). We will use Spring's `InternalResourceViewResolver` class to resolve our view names. (This is covered in the next section.)

Spring Setup for Time Expression

Now that I have provided you some fundamental concepts for Spring MVC, let's begin setting it up for development of Time Expression screens.

We need a couple of components to get Spring up and running for us. Figure 7.1 showed the Time Expression high-level architecture we established early in this book. As you can see, we need a servlet container that Spring can run within for our web application. So let's start with the installation of a Servlet container first, and then we will download and install the Spring Framework.

Installing a Servlet Container (Apache Tomcat)

I have chosen to use Apache Tomcat (http://tomcat.apache.org/) as the Servlet container for the Time Expression application. However, you can use any other product you want; this can be a servlet-container-only product, such as Tomcat, or a full-blown application server, such as JBoss Application Server, BEA WebLogic, or IBM Websphere.

> **Note**
>
> If you have been following along the examples in this book, you will recall the `timex/local.properties` file used by our Ant `build.xml` file (both files are provided in this book's code zip file). Note the `deploy.dir` property in the `timex/local.properties` file; this can be adjusted to point to your servlet container's deployment directory. For example, in my case, the `deploy.dir` property is set up as shown here:
>
> ```
> deploy.dir=/apache-tomcat-5.5.15/webapps
> ```

Now we can run the `ant deploy` from a command line using our `build.xml` file.

By running this `ant` command, a fresh new `timex.war` web archive file will be built and deployed to the specified directory (in `deploy.dir`).

Hot Deploying WAR Files and HTTP Mock Style Testing

In 2001, I wrote an article titled "How Many Times Do You Restart Your Server During Development?" (http://www.javaworld.com/javaworld/jw-04-2001/jw-0406-soapbox.html). Although various servlet containers or application servers handle reloading of applications differently, restarting the server every time you make a change to your application can become a waste of time. Much of this has to do with the way Java's class loading works, but it still doesn't make it any less frustrating.

If your server doesn't (hot) redeploy your war files successfully, you could consider tweaking your style of coding and testing. One good alternative (discussed in this chapter) is to use Spring's mock classes to simulate a HTTP request and use JUnit to unit test the code instead of relying completely on the web application server for your testing.

Incidentally, I recently came across an option for Apache Tomcat that will enable to us to avoid restarts when deploying our application. This can be activated by setting the following attributes in the `conf/context.xml` file found under the Tomcat install directory, `<Context antiJARLocking="true" antiResourceLocking="true">`.

Documentation on these attributes can be found at http://tomcat.apache.org/tomcat-5.5-doc/config/context.html#Standard%20Implementation.

Alternatively, we could use the Tomcat Ant deploy tasks; however, I wanted to keep our `build.xml` generic for most web servers. Nevertheless, documentation on these tasks can be found at the tomcat.apache.org website.

Installing the Spring Framework

By now, you should have a thorough understanding of what Spring can do for you. Next, it is time to download Spring, install it, and begin using it!

The Spring Framework can be downloaded from http://springframework.org. We will now follow the instructions provided on the website to download and install it.

The following are one-time setup steps we will need to follow to get Spring set up for our environment. From here, you might add external jars for added Spring functionality as needed to the `timex/lib/` directory. (In Chapter 10, "Beyond the Basics," we will add OpenSymphony's `quartz.jar` file to our directory.)

- Spring—Copy `spring.jar` to the `timex/lib/` directory of Time Expression, based on the directory structure we established in Chapter 3, "XP and AMDD-Based Architecture and Design Modeling," and shown here in Figure 7.4.
- JSTL—We also need to obtain JavaServer Pages Standard Tag Library (JSTL), which is part of the Jakarta taglibs project and can be downloaded from http://jakarta.apache.org/taglibs/. After downloading this package, copy the

`jstl.jar` and `standard.jar` files to the `timex/lib/` directory. JSTL helps eliminate (or at least significantly reduces) the amount of embedded scriptlet code in our JSP files. For example, JSTL provides tags for iterations/loops (`<forEach>`, for example), conditional tags (`<if>`, for example), formatting tags (`fmt:formatDate`, for example), and several other tags. You will see examples of many of these tags in this chapter.

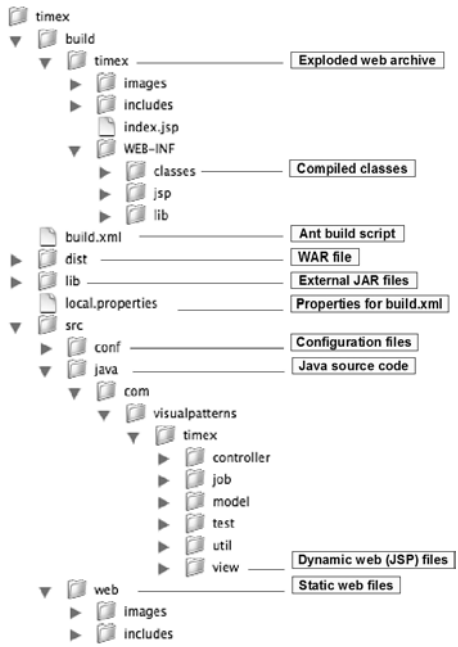

Figure 7.4 Development directory structure for Time Expression.

Running Our SpringTest

Incidentally, the three files we discussed in the previous chapter can now be created in the following paths, and we could run `ant springtest` (from our `timex/` top-level directory) to test that we can use Spring in our code. The complete code for these files can be found in this book's code zip file:

- `src/conf/springtest-applicationcontext.xml`
- `src/java/com/visualpatterns/timex/test/SpringTest.java`
- `src/java/com/visualpatterns/timex/test/SpringTestMessage.java`

Configuring Spring MVC

Now that we have the servlet container and Spring software installed, we need to configure Spring MVC so that we can begin developing and deploying the Time Expression sample application.

Configure `DispatcherServlet` in `web.xml`

The very first thing we need to do is to have all incoming HTTP requests (that match a certain pattern) forwarded to Spring MVC, by Tomcat.

The following excerpt from our `web.xml` file demonstrates how we can configure all requests ending with an `.htm` extension to be processed by the Spring's `org.springframework.web.servlet.DispatcherServlet` class:

```
<servlet>
    <servlet-name>timex</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>timex</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Later on we will see how requests with a `.jsp` extension are handled by Spring's `DispatcherServlet`.

> **Note**
>
> Our Spring application context file, `timex-servlet.xml`, will automatically be searched for and loaded by Spring for us.
>
> This file is stored under `timex/src/conf` but automatically copied to the `timex/build/timex/WEB-INF/` directory by our Ant `build.xml` file when the build, dist, or deploy targets are used.

Create Spring's Application Context XML File (`timex-servlet.xml`)

Now we need to create our application context XML file, `timex-servlet.xml`. We will review various parts of this file throughout the remainder of this chapter. You will see how this file quickly becomes an essential part of working with Spring MVC.

The following excerpt from `timex-servlet.xml` shows how we configure a Spring view resolver to resolve logical view names to the physical view (JSP) file:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
```

```
    </property>
    <property name="prefix">
        <value>/WEB-INF/jsp/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
```

Note

By storing our JSP files in the `build/timex/WEB-INF/jsp/` directory, we are essentially *hiding* these files so they cannot be accessed directly from a web browser using their actual filenames (that is, only views ending with `.htm` are mapped to these files). To access `.jsp` files directly, they must be placed a couple of levels up, under `build/timex/`, the same location where our welcome file, `index.jsp`, will reside.

Hiding files is a security precautionary measure. Appendix D, "Securing Web Applications," provides additional security guidelines.

Developing Time Expression User Interfaces with Spring

Now that we have Tomcat and Spring installed and set up, we can go through the steps required to develop our sample screens. Let's look at two Time Expression screens we will develop in this chapter—one a nonform screen and the other an HTML form screen.

Timesheet List Screen

Figure 7.5 shows the Timesheet List screen, which is a nonform screen (that is, it contains no input fields a user can fill in because it is a display-only screen). From the perspective of coding a controller, this is the most basic screen that you can develop using Spring MVC; we will review the code behind this shortly.

Enter Hours Screen

Figure 7.6 shows the Enter Hours screen, a form screen (that is, it contains input fields a user can fill in). This is a little more complicated than the Timesheet List screen because we will have to bind the HTML form fields to our Java code, perform validations on the data entered, display errors, and so on.

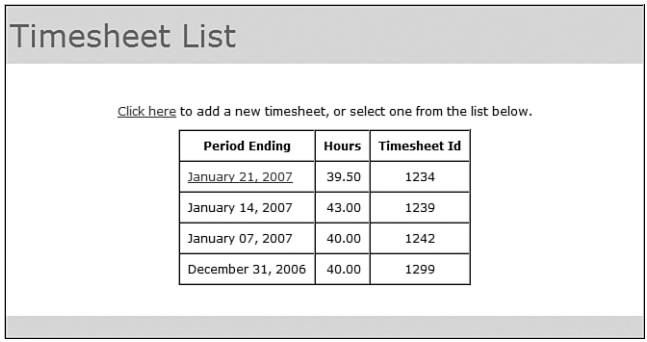

Figure 7.5 Time Expression's Timesheet List web page (view name: timesheetlist).

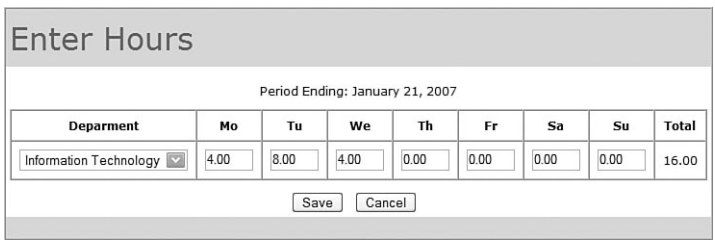

Figure 7.6 Time Expression's Enter Hours web page (view name: enterhours).

Java Files

By now we have enough information to come up with filenames for our Java classes and JSP (view) filenames. Table 7.2 shows a map of the view, controller, and collaborator (model) classes required to complete the two screens shown in Figures 7.5 and 7.6. You might recall that we designed this map in Chapter 3 (see Table 3.5).

Table 7.2 **Sample Application Flow Map (from Chapter 3)**

Story Tag	View	Controller Class	Collaborators	Tables Impacted
Timesheet List	timesheetlist	TimeSheetListController	TimesheetManager	Timesheet
Enter Hours	enterhours	EnterHoursController	TimesheetManager	Timesheet Department

Note that the collaborator classes mentioned here were already developed in Chapter 5, "Using Hibernate for Persistent Objects," so we need to develop the view and controller classes now.

Figure 7.7 shows a rudimentary class diagram on how the controller and model related classes fit together.

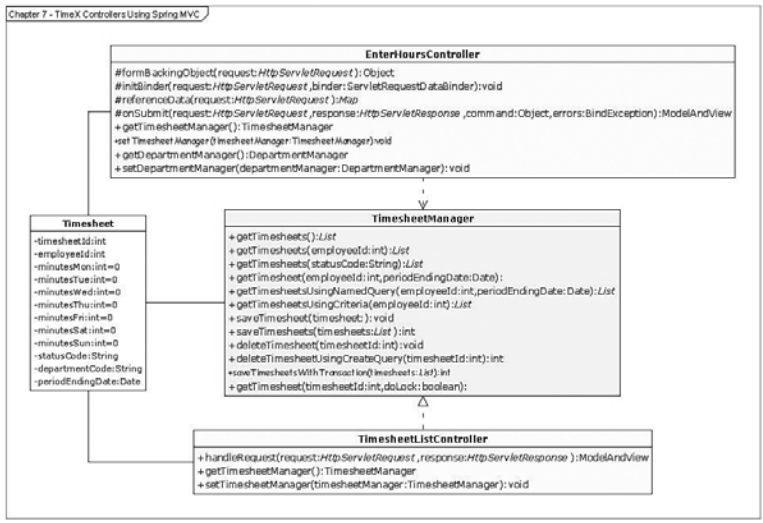

Figure 7.7 Class diagram showing relationship between Time Expression model and controller classes.

If you have developed web applications in Java before, you might question the placement of `.jsp` files under the same directory structure as my Java classes (that is, `java/com/visualpatterns/timex/`); this is purely a personal preference because I like to see my MVC files grouped together under the same parent directory.

Let's look at how to develop the Timesheet List and Enter Hours screens, step-by-step. We will later look at how to develop the Sign In screen because it is a special case because of the authentication (sign in) required.

Cascading Style Sheet (CSS)

Other than the Java and JSP files we discussed, we are also using a cascading style sheet (CSS) file named `timex.css` (placed in our `src/web/includes` directory). CSS provides a consistent look-and-feel across our user interfaces; furthermore, it helps reduce the size of our JSP/HTML code because we don't have as much formatting code in our view (JSP) files.

Timesheet List Screen: A No-Form Controller Example

Developing a *no-form* controller in Spring is a relatively straightforward process. Let's look at the steps involved to do this.

Step-by-Step Configuration

The following are Spring-related items we need to configure in `timex-servlet.xml`, our Spring application context file.

Map Handler

The first thing we need to do is to map the incoming request URL to an actual controller, which will handle the request. The following excerpt from the `timex-servlet.xml` file shows how we can map the `timesheetlist.htm` URL to an internal bean reference named `timesheetListController` (discussed next) with the help of Spring's `SimpleUrlHandlerMapping` class:

```
<bean id="urlMapAuthenticate"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="httpRequestInterceptor" />
        </list>
    </property>
    <property name="urlMap">
        <props>
            <prop key="/timesheetlist.htm">
                timesheetListController
            </prop>
```

Also, notice the `interceptors` property; by configuring this, we can intercept HTTP requests, for example, to implement authentication (interceptors are discussed in detail later in this chapter).

Define Controller and Associated Class

The next step is to define the controller class referenced by the map handler. The following excerpt from the `timex-servlet.xml` file demonstrates how this is done:

```
<bean name="timesheetListController"
    class="com.visualpatterns.timex.controller.TimesheetListController">
    <property name="timesheetManager">
        <ref bean="timesheetManager" />
    </property>
    <property name="applicationSecurityManager">
        <ref bean="applicationSecurityManager" />
    </property>
```

```
    <property name="successView">
        <value>timesheetlist</value>
    </property>
</bean>
```

Notice the `ref` attributes. As you might guess, these are references to other beans defined in our application context, as shown in this XML excerpt:

```
<bean id="timesheetManager"
    class="com.visualpatterns.timex.model.TimesheetManager"/>
<bean id="applicationSecurityManager"
    class="com.visualpatterns.timex.util.ApplicationSecurityManager" />
```

We already developed the `TimesheetManager` class in Chapter 5; we will develop the `ApplicationSecurityManager` class later in this chapter.

This is all we need to configure for the Timesheet List screen. Now we need to write the controller and view code, referenced here. Let's look at that next.

Step-by-Step Coding

The Timesheet List screen is a relatively simple screen and will be developed using the most basic type of Spring controller because it contains no form fields; therefore, it will not require things such as Command and Validator classes. Basically, if we look at this from an MVC design pattern perspective, the files for this screen will include the following:

- Model—`TimesheetManager.java` and `Timesheet.java`
- View—`timesheetlist.jsp`
- Controller—`TimesheetController.java`

We already developed the model files in the previous chapter, so all we need to develop here are the controller and view files. Let's dissect and review parts of our complete code.

Let's begin by writing the unit test code for our controller class.

Writing Our Test First with Mock Objects

The next few code excerpts from our `TimesheetListControllerTest.java` file show how we can unit test controller classes. We will create this in the `timex/src/java/com/visualpatterns/timex/controller` directory.

We start by creating an instance of the `org.springframework.mock.web.MockHttpServletRequest` class to simulate a real HTTP request. This class not only provides the benefit of being able to unit test our code but also reduces the need to deploy the application and potentially restart the servlet container (Tomcat, for example) each time we want to test something.

```
mockHttpServletRequest = new MockHttpServletRequest("GET",
"/timesheetlist.htm");
```

Next, we will create some test dependency objects and *inject* them, as Spring will do for us at runtime:

```
Employee employee = new Employee();
employee.setEmployeeId(EMPLOYEE_ID);
applicationSecurityManager.setEmployee(mockHttpServletRequest,
        employee);

// inject objects that Spring normally would
timesheetListController = new TimesheetListController();
timesheetListController.setTimesheetManager(timesheetManager);
timesheetListController
        .setApplicationSecurityManager(applicationSecurityManager);
```

In our test code, we instantiated our own TimesheetManager class for the sake of simplicity. However in real-world applications, you might want to use Spring's FileSystemXmlApplicationContext or ClassPathXmlApplicationContext classes to instantiate your classes. This way, you not only get an instance of a class but also have its dependent objects loaded and injected by Spring.

Now we can complete our test by checking the `java.util.List` we just retrieved; our test ensures that list is not empty and also that it contains `Timesheet` objects for the employee we requested the records for:

```
ModelAndView modelAndView = timesheetListController.handleRequest(
        mockHttpServletRequest, null);

assertNotNull(modelAndView);
assertNotNull(modelAndView.getModel());

List timesheets = (List) modelAndView.getModel().get(
        TimesheetListController.MAP_KEY);
assertNotNull(timesheets);

Timesheet timesheet;
for (int i = 0; i < timesheets.size(); i++)
{
    timesheet = (Timesheet) timesheets.get(i);
    assertEquals(EMPLOYEE_ID, timesheet.getEmployeeId());
    System.out.println(timesheet.getTimesheetId() + " passed!");
}
```

That's about it for our unit test class; now let's review the actual TimesheetListController class.

Writing Unit Test and Actual Code in the Same Sitting

This book's code zip file shows the complete code for our `TimesheetListControllerTest.java` class, which is the JUnit test case for `TimesheetController.java`. As I've preached previously in this book, development of a unit test and the actual code works best when it is done in the same sitting. For example, I wrote the `TimesheetListControllerTest.java` and `TimesheetListController.java` in the same sitting; that is, I coded a little, compiled and tested a little, and then repeated these steps until my controller class provided all the functionality I needed. The obvious benefit of this approach was that my code was unit tested by the time I was done!

Furthermore, our controller class will now contain only the code we need—nothing more, nothing less.

Another notable benefit worth mentioning is that at times I find myself getting programmer's block (similar to writer's block*)*. But starting out with the unit test code helps me get going. Note that what I have mentioned here is a personal style of working, but hopefully you will find value in it and give the test-first approach a try (if you don't already do so).

One thing I do want to stress is that like everything else, you need to find the right balance. Although I believe in the test-first approach, there are times when it isn't feasible for me to write a unit test code that becomes more complicated than the actual code or is cumbersome to write. After all, you are writing Java code to test other Java code, which raises an obvious question—do we also test the test code? Of course, I'm kidding here, but my point is to find the right balance and in most cases, unit tests work out pretty well.

Last, unit testing works best if you write small methods that can be unit tested relatively easily.

Controller Code

Now it is time to review the code behind our controller class for the Timesheet List screen, `TimesheetListController.java`. We will create this in the `timex/src/java/com/visualpatterns/timex/controller` directory.

For starters, notice that we are implementing the `org.springframework.web.servlet.mvc.Controller` interface; this is perhaps the most basic type of controller you can develop using Spring.

```
public class TimesheetListController implements Controller
```

The next interesting thing to note is the `handleRequest` method; this is the only method we must implement to satisfy the requirements of the Controller interface.

```
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response)
```

The `handleRequest` method returns a `ModelAndView` object, which contains the view name and the model data (a `java.util.List`, in our case). The view name is resolved by `JstlView`, which we defined in the `timex-servlet.xml` file we saw earlier in this chapter.

```
        return new ModelAndView(VIEW_NAME,
                                MAP_KEY,
                                timesheetManager.getTimesheets(employeeId));
```

There are a few more variations to how you can construct the `ModelAndView` class, as shown in the following list (see the Spring Framework API Javadocs for details):

- `ModelAndView()`
- `ModelAndView(String viewName)`
- `ModelAndView(String viewName, Map model)`
- `ModelAndView(String viewName, String modelName, Object modelObject)`
- `ModelAndView(View view)`
- `ModelAndView(View view, Map model)`
- `ModelAndView(View view, String modelName, Object modelObject)`

View/JSP Code

We already prototyped the screens in Chapter 2, "The Sample Application: An Online Timesheet System," so we now need to add some code to the related view (`.jsp`) files. This book's code zip file contains the before file, `timesheetlist.html` (prototyped, static HTML), and the after file, `timesheetlist.jsp` (dynamic/JSP), versions of this file.

Let's review our `timesheetlist.jsp` a bit closer. For starters, we will create this in the `timex/src/java/com/visualpatterns/timex/view` directory. Now let's look at some JSP code.

The following excerpt from our `timesheetlist.jsp` file shows the dynamic code used for populating the HTML table on the Timesheet List screen; this is done in a loop using JSTLs `forEach` tag. Within each loop, we are generating the HTML table's rows and columns (and also formatting the hours) using the JSTL core library.

```
<c:forEach items="${timesheets}" var="timesheet">
    <tr>
        <td align="center"><a
            href='enterhours.htm?eid=<c:out
            value="${timesheet.employeeId}"/>&amp;tid=<c:out
            value="${timesheet.timesheetId}"/>'><fmt:formatDate
            value="${timesheet.periodEndingDate}" type="date"
```

Now let's look at another interesting piece of code from our view file, `timesheetlist.jsp`:

```
<c:if test="${not empty message}">
   <font color="green"><c:out value="${message}"/></font>
   <c:set var="message" value="" scope="session"/>
</c:if>
```

All this code does is check for any messages stored in the `message` session attribute. This message is set by the Enter Hours controller upon a successful save in the `onSubmit` method, as you will see later in the chapter.

We just looked at how to configure and code the Timesheet List screen. Now it is time to review more complex Spring MVC features.

Enter Hours Screen: A Form Controller Example

The Timesheet List screen example we just looked at demonstrated how to develop a simple no-form controller. Now let's look at a slightly more complex example using the Enter Hours screen shown in Figure 7.6.

As you can see from Figure 7.6, the Enter Hours screen enables users to enter their hours and select the department these hours should be charged to (using a drop-down list). This functionality will require us to get a list of department names, bind the HTML form fields to a Java object, validate data entered on the screen, and display error/status messages on the screen.

Step-by-Step Configuration

The following are steps required to configure the Enter Hours screen in our `timex-servlet.xml` file. For the sake of brevity, I will not provide detailed explanations for the same steps we covered previously for the Timesheet List screen.

Map Handler

The following line provides the mapping for the Enter Hours view to the controller class:

```
<prop key="/enterhours.htm">enterHoursController</prop>
```

Define Controller and Associated Classes

The configuration for the Enter Hours controller is a bit more involved than the Timesheet List controller, so let's take a closer look at it.

First, you will notice that we have two model classes and one security-related (utility) class; these are required for the Enter Hours screen to function, which are configured as follows:

```
<property name="timesheetManager">
    <ref bean="timesheetManager" />
</property>
<property name="departmentManager">
    <ref bean="departmentManager" />
</property>
<property name="applicationSecurityManager">
    <ref bean="applicationSecurityManager" />
</property>
```

The following lines configure the command class for the `EnterHoursController`:

```
<property name="commandClass">
    <value>com.visualpatterns.timex.model.Timesheet</value>
</property>
```

The remainder of the configuration for this controller is Spring specific. For example, you will notice the validator property, which is an optional configuration but one we will use to validate the input data from the screen. The `formView` is the name of the actual form view and `successView` is the view you want Spring to redirect to upon a successful form submittal. The `sessionForm` property allows us to keep the same instance of the command object in the session versus creating a new one each time.

```
<property name="formView">
    <value>enterhours</value>
</property>
<property name="successView">
    <value>redirect:timesheetlist.htm</value>
</property>
<property name="validator">
    <ref bean="enterHoursValidator" />
</property>
```

ResourceBundle

One other configuration item we should look at is related to externalizing string messages for internationalization and other purposes, as shown here:

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>messages</value>
        </list>
    </property>
</bean>
```

The `ResourceBundleMessageSource` Spring class relies on JDK's `java.util.ResourceBundle` class; we will use this to externalize our error and status messages in a file called `messages.properties` (placed in our `timex/src/conf` directory), which contains the following messages:

```
typeMismatch.int=Invalid number specified in a numeric field
error.enterhours.missingdepartment=Please select a department
error.login.invalid=Invalid employee id or password
message.enterhours.savesuccess=Timesheet saved successfully
```

Alternatively, Spring also provides a class named `ReloadableResourceBundleMessageSource`, which can be used to reload the properties periodically using its `cacheSeconds` parameter setting. This can come in handy during development, when the messages file can change often.

Step-by-Step Coding

The following is Spring-related Java code we need to write for our form controller. By the end of this Enter Hours example, we will end up with the following files (under our `timex/src` directory):

- `conf/messages.properties`
- `java/com/visualpatterns/timex/controller/EnterHoursController.java`
- `java/com/visualpatterns/timex/controller/EnterHoursValidator.java`
- `java/com/visualpatterns/timex/controller/MinutesPropertyEditor.java`
- `java/com/visualpatterns/timex/view/enterhours.jsp`

Controller Code

Let's start by developing the controller. For starters, notice that instead of implementing the `org.springframework.web.servlet.mvc.Controller` interface as we did for the `TimesheetListController`, we are extending Spring's `org.springframework.web.servlet.mvc.SimpleFormController` (concrete) class.

```
public class EnterHoursController extends SimpleFormController
```

`SimpleFormController` implements the Controller interface but also is part of a hierarchy of various abstract controller-related classes (as we saw in Figure 7.3). It can also automatically redirect the user to the default form view in case of errors and to a different (or same) view if the form submission is successful; this is controlled using the `successView` and `formView` properties we set in our `timex-servlet.xml` for the `enterHoursController` Spring bean, as we saw earlier.

Let's take a look at the various Spring-related methods for form processing. However, before looking at each method, let's look at the order in which these methods are called.

Figure 7.8 shows three boxes: the first box is essentially when the user first enters the screen; the second box is when the user submits the form with invalid fields (that is, validation fails), and the third/last box shows which methods are called when the validation is successful. Now let's review the type of code that goes into each of these methods.

The first method I will discuss is the `formBackingObject`, which returns a command object that is used to hold the input data from the HTML form fields. Notice that we fetch an existing `Timesheet` record from the database if parameters are passed into the controller, indicating it is an edit operation versus an add operation, in which case, we construct a new command object (which, incidentally, is a Time Expression domain/business object).

```
protected Object formBackingObject(HttpServletRequest request)
{
    if (request.getParameter(TID) != null
            && request.getParameter(TID).trim().length() > 0)
        return timesheetManager.getTimesheet(Integer.parseInt(request
                .getParameter(TID)), false);

    Timesheet timesheet = new Timesheet();
    Employee employee = (Employee) applicationSecurityManager
            .getEmployee(request);
    timesheet.setEmployeeId(employee.getEmployeeId());
    timesheet.setStatusCode("P");
    timesheet.setPeriodEndingDate(DateUtil.getCurrentPeriodEndingDate());
    return timesheet;
}
```

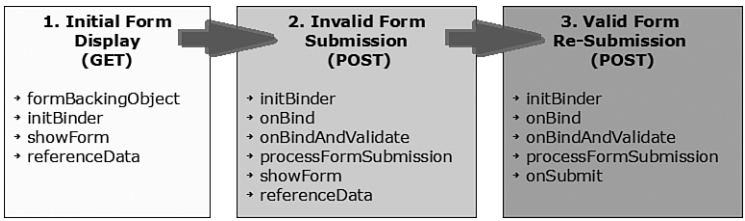

Figure 7.8 Life cycle of `EnterHoursController`.

Binding Directly to Domain (Business) Objects

One vital benefit of Spring MVC is the capability to bind the form fields directly to a domain object (Timesheet, for example)! This is one of the things that separates Spring from many other web frameworks.

Next up is the `initBinder` method, which provides a good place to register custom property editors (discussed shortly), as shown here:

```
binder.registerCustomEditor(int.class, new MinutesPropertyEditor());
```

The `referenceData` method is a good place to return read-only data for forms, typically for drop-down lists on the screen, as we have done by returning a list of departments for the Enter Hours screen:

```
model.put("departments", departmentManager.getDepartments());
```

Last, but not least, let's look at one of the most important methods in our controller class, the `onSubmit` method, shown next. As we saw in Figure 7.8, this method is called only after all validations have passed through successfully:

```
protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors)
{
    Timesheet timesheet = (Timesheet) command;
    timesheetManager.saveTimesheet(timesheet);
    request.getSession().setAttribute(
            "message",
            getMessageSourceAccessor().getMessage(
                    "message.enterhours.savesuccess"));
    return new ModelAndView(getSuccessView());
}
```

Also, notice the following code in the `onSubmit` method, which returns a successful message via the HTTP session. This message is extracted from the `messages.properties` file (using the `message.enterhours.savesuccess` key) and displayed on the Timesheet List screen.

This is about all we will cover for the controller class. Now, let's look at the other related classes used by this controller.

Custom Property Editor

As I mentioned earlier in this chapter, Spring makes heavy use of JavaBean style property editors (that is, `java.beans.PropertyEditorSupport`).

We will write a custom property editor class, `MinutesPropertyEditor`, to convert the hours entered on the screen to minutes because that is how our database is designed. The code for this class should be fairly straightforward because it performs the conversion from minutes to hours and vice versa (that is, multiplying or dividing by 60 minutes).

Validation

Our validation example is very also fairly straightforward. The main code really is in the `validate` method of this class, as shown in the following code excerpt:

```
Timesheet timesheet = (Timesheet)command;
if (timesheet.getDepartmentCode() == null ||
    timesheet.getDepartmentCode().trim().length() < 1)
    errors.reject("error.enterhours.missingdepartment");
```

The `error` variable shown here is of type `org.springframework.validation.Errors`, which provides several `reject` methods. The example I have shown here is useful for displaying global messages for the entire screen; I tend to use this method rather than the field-specific ones. For example, one of the field-specific `reject` methods has the following signature: `rejectValue(String field, String errorCode)`.

Also, you might have noticed an `onBindAndValidate` method in Figure 7.8. This method has the following signature:

```
onBindAndValidate(HttpServletRequest request,
                  Object command,
                  BindException errors)
```

This method is called by Spring automatically after the `Validator` object has been invoked. This is a great place to do additional validations—for example, validations based on parameters sent in via HTTP request or database validations using one of the injected model classes, perhaps to check for duplicate records in the database.

View/JSP Code

Now that we are done looking at Java classes for the Enter Hours screen, we can look at the corresponding view code, located in our `enterhours.jsp` file. We will inspect a few excerpts here.

The first interesting block of code in our view is the displaying of error messages set in our `EnterHoursValidator` class, as shown here:

```
<spring:bind path="command.*">
    <c:if test="${not empty status.errorMessages}">
      <c:forEach var="error" items="${status.errorMessages}">
                  <font color="red"><c:out value="${error}"
escapeXml="false"/></font><br/>
      </c:forEach>
    </c:if>
</spring:bind>
```

This is the first time we are seeing the `spring:bind` tag, so let me explain a few things about it.

The key class behind the spring bind tag library is `org.springframework.web.servlet.support.BindStatus`. This tag enables you to bind the HTML form fields to the command object (Timesheet, in our case). However, it also provides access to a special variable named *status*. The `status` object contains some of the following attributes, which can be used in the JSP code:

- status.value—The value of a given attribute in the command object
- status.expression—The name of a given attribute in the command object
- status.error—A Boolean flag indicating whether an error exists
- status.errorMessage—A field-specific error message
- status.errorMessages—Global error messages for the view
- status.displayValue—Get a string value suitable for display using `toString`

Now let's look at how fields are bound. The following code shows how the `departmentCode` JSP/HTML variable is bound to the matching variable in our Command object (that is, `Timesheet.departmentCode`).

```
<spring:bind path="command.departmentCode">
```

That is really all there is to `enterhours.jsp`; some of the code I have not explained here is because we already covered similar code for the Timesheet List screen example earlier in this chapter (such as looping through code using the JSTL `forEach` tag).

I wish I could tell you there is more to Spring's bind tag library, but as I mentioned earlier, this library is fairly simple; but what you can do with it is quite powerful.

Binding to Custom (Nonbusiness) Command Objects

One of the key benefits of Spring MVC is that it enables you to bind HTML form fields directly to your domain object. Spring refers to these objects as *command* objects, perhaps based on the "Command" design pattern, which basically involves encapsulation of a request in an object. Another way to view the concept of a command object is to view it as our *form object* because it can hold all the values entered on the HTML form. However, because we can bind our HTML form fields directly to our business objects or have other data stored in this object, the term *command* is more appropriate.

For the Time Expression screens, we bind directly to `Timesheet`, our domain object. However, you always have the option to create a custom Command class, which could, for example, extend or contain the `Timesheet` class and provide some additional methods. For instance, I worked on a project where I need to assemble and disassemble a `java.util.Date` object because the HTML form had separate drop-downs for month, date, and year. In that case, I used methods such as `assembleDate` and `disassembleDate` in a custom command class.

There are a couple of ways you can approach a custom command class. For example, we could have done something like the following:

```
public class TimesheetCommand extends Timesheet
```

By doing this, you can still bind directly to the setter/getter methods of our business object, but also extend it by adding additional methods, as needed. Also, to construct a custom command class, you would need to specify it in the `timex-servlet.xml` file and also construct/return an object of this type in the `formBackingObject` method.

The other approach is to have the `TimesheetCommand` class contain a reference to the `Timesheet` object. For example, this class could have a constructor as follows:

```
public TimesheetCommand(Timesheet timesheet) {...}
```

Using this approach, you would bind the HTML form fields to the *Timesheet* object using a notation similar to this:

```
command.timesheet.minutesMon
```

The one problem you run into with this approach is related to JavaScript validation checking because JavaScript gets confused with the dots in HTML field names. For example, `command.timesheet.minutesMon` would translate into `timesheet.minutesMon` for the HTML input text field name if we used `${status.expression}` to fill in the name of this input field.

DateUtil.java

The one other notable file is `DateUtil.java`; this file provides some utility type date methods. For example, our `EnterHoursController` class uses one of these methods in its `formBackingObject` method:

```
timesheet.setPeriodEndingDate(DateUtil.getCurrentPeriodEndingDate());
```

JSP Taglib Directives

The one thing I haven't pointed out explicitly until now are the following lines of code you might have noticed in our JSP files:

```
<%@ taglib prefix="c"      uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"    uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

These directives are required before using a JSP tag library. More information on this and other JSP features can be found on the java.sun.com website.

Views with No Controllers

There might be times when you do not need or want to write a controller. For example, suppose we want to implement a help screen for Time Expression. We want this help screen to be accessible as `/help.htm` and have the real file (`help.jsp`) hidden in `/WEB-INF/jsp`. In this case, we would first define `UrlFilenameViewController` in `timex-servlet.xml`, as shown next:

```
<bean id="urlFilenameController"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

Then we can reference `urlFilenameController` in our handler mapping (the `urlMap` bean in `timex-servlet.xml`, for example):

```
<prop key="/help.htm">urlFilenameController</prop>
```

Spring HandlerInterceptors

Until now, we developed our Timesheet List and Enter Hours screens without worrying about authentication. However, one of our fundamental requirements from Chapter 2 is that employees can see only their own timesheets, which brings us to our Sign In and Sign Out features.

Spring provides the concept of interceptors for web application development; these enable you to intercept HTTP requests. We will use this feature to provide authentication for Time Expression.

To implement our sign in/out features, we will need to create the following files under the `src/java/com/visualpatterns/timex` directory:

- `controller/HttpRequestInterceptor.java`
- `controller/SignInController.java`
- `controller/SignInValidator.java`
- `controller/SignOutController.java`
- `util/ApplicationSecurityManager.java`
- `util/DateUtil.java`
- `view/signin.jsp`

Authentication for Time Expression

The authentication for Time Expression is enabled by having all HTTP requests requiring authentication to be mapped as they go through our interceptor class, `HttpRequestInterceptor.java`. The following code excerpt demonstrates how an intercepted request can be preprocessed:

```
public class HttpRequestInterceptor extends HandlerInterceptorAdapter
{
   private ApplicationSecurityManager applicationSecurityManager;

    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler)
           throws Exception
    {
        Employee employee =
            (Employee)applicationSecurityManager.getEmployee(request);
        if (employee == null)
        {
            response.sendRedirect(this.signInPage);
            return false;
        }

        return true;
```

Notice the use of `ApplicationSecurityManager` here (and referenced several times earlier in this chapter). The complete code for this class should be fairly straightforward to follow because it essentially provides methods for seting, getting, and removing a HTTP session attribute named user (of type `Employee`, one of our domain objects), as demonstrated in the following code excerpt, which sets this attribute:

```
public static final String USER = "user";
public void setEmployee(HttpServletRequest request, Object employee)
{
    request.getSession(true).setAttribute(USER, employee);
}
```

The `SignInController` class validates the login and also sets the `Employee` domain object using the `ApplicationSecurityManager.setEmployee` method, as shown next:

```
Employee formEmployee = (Employee) command;
Employee dbEmployee = (Employee) command;
if ((dbEmployee = employeeManager.getEmployee(formEmployee
        .getEmployeeId())) == null)
    errors.reject("error.login.invalid");
else
    applicationSecurityManager.setEmployee(request, dbEmployee);
```

Our `SignOutController` class signs the user out by removing the Employee attribute from the session, as shown here:

```
applicationSecurityManager.removeEmployee(request);
```

Note

Our application uses a minimal index.jsp file, which will serve as our welcome file; this is placed under our `src/web` directory and forwards the request to the our `signin.htm` URL, as shown here:

```
<c:redirect url="signin.htm"/>
```

Our Sample Application—in Action!

Now that we have our web user-interface components (controller and view) and our model code developed, we have a completely functional application that can be built, deployed, and test driven!

For example, we can now type ant deploy on the command line and have it (hot) deploy to our Tomcat webapps directory. After deployment, the application can be accessed from a web browser using a URL such as http://localhost:8080/timex/. Figures 7.9 through 7.11 show our screens in action.

Figure 7.9 Sign In screen.

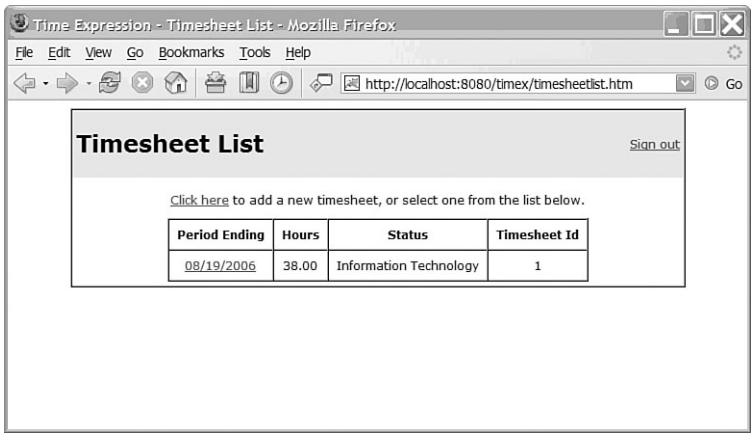

Figure 7.10 Timesheet List screen.

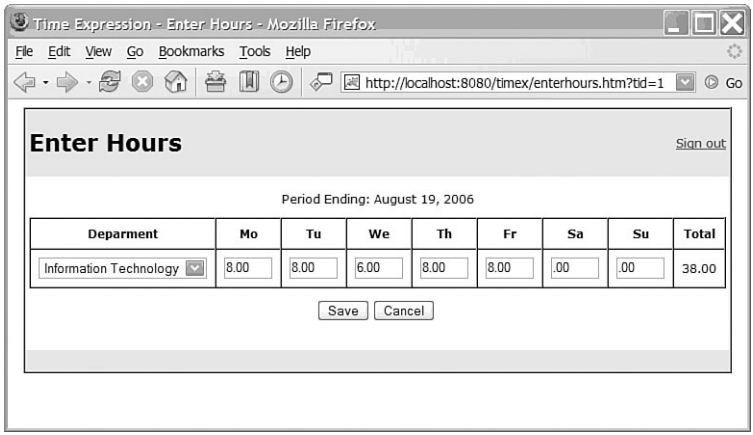

Figure 7.11 Enter Hours screen.

Personal Opinion: Designing and Coding with Interfaces

The Spring reference documentation and many articles on the web recommend designing and coding with interfaces. Spring supports both interface-based and class-based beans.

You might have noticed in Time Expression's class design that I have not used Java interfaces. This is related to my philosophy on when and where to use interfaces.

Let me start out by saying that I like programming with Java interfaces a lot! However, similar to the way many people jumped on the EJB bandwagon a few years ago, I see people jumping on the interface bandwagon recently. So, allow me to share my thoughts on this matter; you may agree or disagree with them. I would like to begin by telling you a little story on the topic of designing and coding with interfaces.

I have been using interfaces since 1996 and love the concept behind them. In 1997, I developed a 100% pure Java-based backup software named BackOnline (http://visualpatterns.com/backonline/). This product was mentioned in several well-known trade journals and won a Best Client award at JavaOne; it was even nominated by Scott McNealy (CEO, Sun Microsystems) for a Computerworld-Smithsonian award. BackOnline is a client-server product; the server is a multithreaded and multiuser server that essentially receives the files and stores them using an implementation class for an interface named DataStore. The DataStore interface has FTP-like methods, such as get, put, open, close, and so on; these, had to be implemented by concrete classes. The BackOnline software (which is no longer being sold) came prepackaged with two default DataStore implementation classes, DataStoreFileSystem and DataStoreJDBC (the fully qualified implementation class names were specified in a configuration file and dynamically loaded at runtime). DataStoreFileSystem essentially used the java.io package to store the files using the local file system. DataStoreJDBC used JDBC to store the file contents as Binary Large Objects (BLOBs) in a relational database.

I provided Javadoc and additional technical documentation for the DataStore interface, so Internet Service Providers (ISPs) and products vendors who signed an OEM (original equipment manufacturer) with my company could write their own custom implementations, if necessary. For example, an ISP might have wanted to take advantage of the native operating system's features, such as extended file permissions.

For the BackOnline example I just went through, using interfaces was an obvious choice. Also, many times I find that interfaces work well for lower-level APIs, such as the one I described for BackOnline or ones you find in frameworks such as the JDK or the Spring Framework (for example, `java.util.Collections` or `java.sql.Connection`). Furthermore, interfaces are great, if you think the underlying implementation can change (such as logging, authentication service, and OS specific functionality). Of course, with remote technologies (EJB, for example), you have no choice but to use interfaces.

For business applications, more times than not, especially on smaller projects, I have found that you need only one implementation of domain (business) objects or service objects (such as the TimesheetManager class for Time Expression). Furthermore, it doesn't make sense to have interfaces for domain objects (such as Timesheet.java, for example).

Creating one interface file for each implementation class amounts to unnecessary overhead, in my opinion. For large projects, this can amount to lots of extra `.java` (and `.class`) files without potentially adding much value. On the flip side, there are times when using interfaces makes sense. For example, in Chapter 2, we discussed multiple user types (roles) for the Time Expression application, such as Employee, Manager, and Executive. These could easily be created as concrete classes that implement an interface named Person or Role. On the other hand, given the common behavior in these objects, an abstract class would also make a lot of sense because the common methods could be pulled up into a super (parent) class (called Person, for example).

In summary, given the right opportunity, you should use interfaces—but do not use them because it has been preached in some book or article as the right thing to do. Furthermore, you should not feel at fault for not using interfaces for each and every concrete class you write. Focus more on having a sound design for your application—for example, clean separation of layers, good database design, easy-to-follow code, appropriate use of architecture/design patterns, and so on. I hope I do not sound dismissive about interfaces because that is certainly not my intention; my point is to use everything in moderation and appropriately.

New Tag Libraries in Spring Framework 2.0

At the time of this writing, the Spring team was getting close to releasing additional tag libraries to make it simpler to work with Spring with JSP. However, the design of these new tag libraries was still evolving, so I was unable to cover this with accuracy.

A Word About Spring Web Flow and Portlet API

Two additional user-interface Spring technologies might be of interest to you, if you have the need for features they provide.

Spring Web Flow

Spring Web Flow, based on similar concepts as Spring Web MVC Framework, essentially provides wizardlike screen functionality to implement a business process. Good examples of such applications include an online shopping site such as amazon.com, which requires you to go through several screens before the transaction is considered complete. This type of functionality requires session/state management, which provides the capability to go back and forth through the screens without losing the information you have already entered. This is precisely the type of functionality Web Flow eases. However, our application, Time Expression, does not require such a feature and would not be a good example for the Spring's Web Flow.

Even though Web Flow is not covered in this book, given the scope of this book, I highly recommend that you give this technology a serious look if your requirements call for this type of functionality.

Spring Portlet API

The Spring Portlet API is a new addition to Spring 2.0. It essentially implements the JSR-168 Portlet Specification defined on the Java Community Process (JCP) website (http://www.jcp.org/en/jsr/detail?id=168). According to this, the Portlet API can be used for "Portal computing addressing the areas of aggregation, personalization, presentation and security." Another way to look at this is that *portlets* are part of a *portal* website, which might contain several portlets. Portlets are different from servlets in that they do not redirect or forward any requests from or to the browser; instead, they are managed by a portlet container.

If you are interested in this API, you should check out the JCP website. Also, you might want to check out Apache's Pluto, a reference implementation for the Portlet API.

Summary

In this chapter, we

- Looked at the various benefits of using Spring MVC
- Took an in-depth look at the Spring Web MVC Framework
- Built three of the screens in Time Expression using Spring MVC: one as a no-form screen, the others as form screens

We covered a lot material in this chapter, but we aren't done with Spring just yet. In the next few chapters, we will touch on various additional facets of the Spring Framework, including

- The Spring IDE plug-in for Eclipse
- Job Scheduling
- Emailing

Meanwhile, if you want to dig into more Spring, take a look at Spring's JPetstore example and Reference Documentation, both part of the Spring distribution software.

In the next chapter, we will look at Eclipse, which will completely change the way we have been working in this book! In other words, we will change from command-line programming to a sophisticated Integrated Development Environment (IDE), which will make coding, unit testing, and debugging much easier—in short, agile Java development!

Recommended Resources

The following websites are relevant to or provide additional information on the topics discussed in this chapter:

Websites for competing technologies to ones discussed in this chapter:

- Apache Jakarta Tapestry http://jakarta.apache.org/tapestry/
- Apache Jakarta Turbine http://jakarta.apache.org/turbine/
- Apache Struts http://struts.apache.org/
- Apache Tapestry http://jakarta.apache.org/tapestry/
- Apache Tomcat http://tomcat.apache.org/
- Apache Tomcat `antiJARLocking` and `antiResourceLocking` configuration attribute http://tomcat.apache.org/tomcat-5.5-doc/config/context.html#Standard%20Implementation
- Apache Tomcat Ant Tasks http://tomcat.apache.org/tomcat-5.0-doc/catalina/docs/api/org/apache/catalina/ant/package-summary.html
- Apache Velocity http://jakarta.apache.org/velocity/
- FreeMarker http://www.freemarker.org/
- JavaServer Faces http://java.sun.com/j2ee/javaserverfaces/
- Jetty Servlet Container http://jetty.mortbay.org/jetty/
- Mock Objects http://mockobjects.com
- OpenSymphony WebWork http://www.opensymphony.com/webwork/
- Spring Discussion Forums http://forum.springframework.org/
- Spring Framework http://springframework.org
- The original MVC http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html