

STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

 HANNING

Chapter 9. Integrating with Spring and Hibernate/JPA.....	1
Section 9.1. Why use Spring with Struts 2?.....	2
Section 9.2. Adding Spring to Struts 2.....	8
Section 9.3. Why use the Java Persistence API with Struts 2?.....	14
Section 9.4. Summary.....	23



Integrating with Spring and Hibernate/JPA

This chapter covers

- Managing objects with Spring
- Knowing when to use dependency injection
- Adding Spring to Struts 2
- Integrating Struts 2 and the Java Persistence API

Now that we've finished the core chapters, we know how to build a basic Struts 2 application. We've even done so with our Struts 2 Portfolio. This chapter starts the part of the book dealing with how to finish the application with a variety of refinements that many developers consider best practices. The refinements that we introduce in this chapter are specifically related to integrating a Struts 2 application with a pair of popular third-party technologies, Spring and the Java Persistence API (JPA).

First, we learn how to use a Spring container to provide a more sophisticated means of managing our application resources. While Spring provides many different services to the application developer, we focus here on the use of Spring as a means of dependency injection. In short, we use Spring to intervene in the creation of both framework and application objects for the purpose of injecting dependencies into

those objects. We both explain what this means and demonstrate the details by upgrading our Struts 2 Portfolio to use Spring for such purposes as injecting our service object into the actions that require it.

After we get Spring in place, we move into a discussion of using JPA/Hibernate to handle our data persistence needs. Hibernate is a popular object-relational mapping (ORM) technology, and the Java Persistence API is a new standardized interface for working with persistence technologies. We'll be coding to the standard JPA, but we'll be using Hibernate beneath the covers. Though Struts 2 doesn't do anything specifically to support integration with the JPA, the framework has generally been designed to ease such tasks. We demonstrate the commonly accepted best practices for using JPA with a Struts 2 application. We also upgrade our Struts 2 Portfolio application to use a JPA PortfolioService object. To top it all off, we use the Spring techniques learned in the first half of the chapter to manage our JPA service.

Where to start? Since Spring serves as the foundation for integrating JPA into the application, we get started in the next section by clearing up what we mean by *dependency injection*. Right after that, we show you how to introduce Spring into the framework's object-creation and management mechanisms.

9.1 Why use Spring with Struts 2?

Good question. Many people ask this question when first confronted with the Spring + Struts 2 equation. (Other people might note that this is not actually an "equation," mathematically speaking.) The confusion here is that Spring is a large framework that contains solutions to many different aspects of a J2EE application. Spring even has its own MVC web application framework, but that's not the part we're going to learn how to use in this chapter. In fact, we won't learn much about Spring at all, at least in the big sense. We'll be focusing on the part of Spring that provides dependency injection. Many Struts 2 developers consider this resource management service to be an essential part of a well-built web application.

In this section, we start by introducing the concept of dependency injection and cover the fundamentals of how Spring handles this. If you're familiar with Spring, you'll probably be comfortable skipping past this introductory material and proceeding straight to section 9.2, where we describe the details of integrating Spring with Struts 2.

9.1.1 What can dependency injection do for me?

We start by describing the problem that we'll use Spring to solve. In a nutshell, a Java application consists of a set of objects. These objects cooperate to solve the problems facing the application. In a Struts 2 application, these objects include application objects like our PortfolioService object, as well as core framework components such as actions and interceptors. It's now time to think about how all these objects get instantiated, and how they come to have references to one another.

Some objects are created by the framework. For instance, when a request comes into the framework, Struts 2 must decide which action class maps to that request.

Once it determines the correct action class, it creates an instance of that class to handle the processing of that request. As a developer, you write the action class and map it to the appropriate URL with XML or Java annotations. You never create an action yourself. Creation of actions and other objects is one of the main jobs of the framework internals. In fact, one of the most important internal components of the framework is the `ObjectFactory`, wherein all framework objects come to life.

But other objects, such as our `PortfolioService` object, aren't automatically created by the framework. At least not yet! Many of our actions depend on this service object to do their work. One way or another, these actions must obtain a reference to a service object. So far, we've been manually creating these objects with a low-tech strategy: the `new` operator. This unsophisticated strategy has created a tight bond between our actions and this `PortfolioService`. It's time to make some improvements in the way we manage our objects and their dependencies upon one another.

One of the most popular technologies for managing the creation of objects in a Java application is Spring. What does Spring add to the management of object creation? As we've been suggesting throughout this book, a well-designed application minimizes the coupling between its objects. The Struts 2 framework itself follows this design imperative quite well. A Struts 2 action, for instance, doesn't contain any references to the Servlet API even though it runs on top of that API when it executes inside the framework. We should strive to continue this level of decoupling as we build our applications on the framework. Spring can help in our decoupling.

But before we look at how Spring does this, let's take a brief look at what it means for objects to be tightly coupled in the first place.

TIGHTLY COUPLED OBJECTS

We can't get around the fact that our objects depend upon each other to do their work. In the code, this means they must acquire references to each other at some point in time. If the acquisition of references is done in the wrong way, the objects become tightly coupled, introducing a variety of issues from complicated testing to nightmarish maintenance. In order to make this issue more clear, we'll explore the shortcomings of the current version of the Struts 2 Portfolio.

The main resource upon which all of our actions are dependent is the `PortfolioService` object. This object provides all the data persistence and business logic needs of our application. It's great that we've been wise enough to extract all of that into a tidy service class. Thanks to our foresight, the action code is clean. However, we've allowed a tight binding to sneak in. Consider our chapter 8 version of the `Register` action, seen in listing 9.1. In particular, take note of the method by which we acquire our reference to the service object.

Listing 9.1 Using the new operator to construct the `PortfolioService` object

```
public class Register extends ActionSupport implements SessionAware {  
    public String execute() {  
        //Prepare the new user object
```

```

    . . .

    getPortfolioService().createAccount( user );
    session.put( Struts2PortfolioConstants.USER, user ); ❶

    return SUCCESS;
}

private String username;
private String password;
private String portfolioName;
private boolean receiveJunkMail;

// getters and setters omitted

public PortfolioService getPortfolioService( ) {
    return new PortfolioService();
}
}

```

**Properties to
receive data
transfer**

In case you've forgotten, this action simply collects the data from the registration form, creates a new `User` object with that data, then uses the `PortfolioService` object ❶ to persist that new user. But the real point of interest here is how we acquire our `PortfolioService` object. Our `Register` action clearly has a dependency upon the `PortfolioService` object. It must obtain a reference to a service object so that it can do its work. This service object is obtained with direct instantiation via the `new` operator ❷. While this works, it presents two major problems:

- 1 We're bound to a specific type, the old memory-based `PortfolioService` object.
- 2 We're bound to a specific means of acquisition, the `new` operator.

We now discuss each of these briefly.

The first problem is that we're bound, or coupled, to this implementation of the `PortfolioService` object because our code has a specific and naked type in it. To see how this is a problem, consider the task that we'll be faced with later in this chapter when we create a new portfolio service object that uses JPA for its data persistence. Let's say the new service class is called `JPAPortfolioService`. In order to integrate that back into the application, we'll have to go into every class that creates a specific instance of the `PortfolioService` object ❷ and change the code by hand to something like the following:

```

public JPAPortfolioService getJPAPortfolioService( )
    return new JPAPortfolioService();
}

```

The second problem is that we're bound to the `new` operator as a means of acquiring our object references. In some ways, this is the worst possible way to acquire your references. There are many other methods of acquisition that make various improvements upon the vulgar `new` operator, ranging from factories to service locators. But even these still suffer from the tight coupling inherent in any code-level means of acquisition. Basically, this means that we still have to touch code in order to change

the object being used, such as our service object. The most obvious headaches caused by this are maintenance and testing. Consider testing. If we wanted to plug in a mock service object to test our action, we'd be forced to intervene at the code level in every occurrence of the acquisition code. It'd be nice to make the change with a single line of declarative metadata, wouldn't it?

We'll now show a best-practice solution to these two problems, tight coupling to a specific implementation and tight coupling to an acquisition method. This solution utilizes a Spring container to inject dependencies into our objects, thus escaping the coupling to a specific acquisition method. Simultaneously, we'll also introduce an interface to provide a layer of separation from our specific service object implementation and our code that handles it. While the interface separation isn't required by Spring, many folks believe it to be a very powerful one-two punch of software design.

First, let's look at how Spring can resolve the issue of resource acquisition.

9.1.2 How Spring manages objects and injects dependencies

As we said, Spring is many things, but one of its most popular uses is the management of object creation and the injection of dependencies into those objects as they're created. Rather than using code to acquire our resources, we use metadata to declare what dependencies a given object requires. Spring reads this metadata, commonly located in an `applicationContext.xml` file, to learn about the dependencies of the objects it manages. Spring offers several means of injecting dependencies into managed objects, but one common method is via setter methods exposed by the managed object.

FYI Struts 2 itself uses a form of setter injection to acquire decoupled access to things such as the Servlet API attribute maps. Actions that want access to the session map, for instance, can implement the `SessionAware` interface, which exposes the `setSession(Map session)` method. The `servletConfig` interceptor, part of the `defaultStack`, will inject the session map into this setter for all actions that implement the `SessionAware` interface. Note that, like the Spring injection we're introducing, this interface allows the action to have decoupled access to the Servlet API session map. The action itself doesn't touch the Servlet API, and the type of the setter is the `Map` interface, thus keeping the action free of binding to a particular map. We could now easily inject a mock session map for testing.

Let's see how our `Register` action would look if we used Spring to inject our service object, rather than directly acquiring it in our code:

```
private PortfolioService portfolioService;

public PortfolioService getPortfolioService() {
    return portfolioService;
}

public void setPortfolioService(PortfolioService portfolioService) {
    this.portfolioService = portfolioService;
}
```

Instead of worrying about how to create our service object, we just let Spring create and inject it into a setter method. We completely get rid of the acquisition code and are left with only a simple JavaBeans property. This is all it takes, in the code, to have your resources injected. It doesn't look like much, but that's because everything's been removed to the Spring metadata layer, most likely `applicationContext.xml`. It is in that file where we tell Spring the details of the object's creation. We learn about that metadata in section 9.2, when we cover the details of integrating and using Spring.

First, though, we need to talk about the second part of that one-two punch. Remember the interface we're also going to introduce, the one that will solve the problem of being bound to a specific implementation of the service object? We look at that next.

9.1.3 *Using interfaces to hide implementations*

While the code snippet in the previous section has completely removed the resource acquisition code by allowing the service object to be externally injected by Spring, we still have a tight binding to the type of the object being injected, a memory-based `PortfolioService` class in this case. As you can see, our `Register` action's JavaBeans property is specifically typed to that class. When we want to upgrade to another implementation of our service object that uses JPA, we'll be forced to refactor all these properties across all the objects that depend on a service object. After that, we'll be just as bound to that JPA version of the service. To solve this binding problem, we'll introduce an interface for service objects that both the memory-based and the JPA-based versions can implement. Then, we'll change the property to that interfaced type.

We've done just this for the chapter 9 version of the Struts 2 Portfolio. Listing 9.2 shows the full source code of our new interface `manning.chapterNine.utils.PortfolioServiceInterface`.

Listing 9.2 The `PortfolioServiceInterface` provides a layer of separation.

```
public interface PortfolioServiceInterface {  
    public boolean userExists ( String username );  
    public void updateUser( User user ) ;  
    public void addImage ( File file ) ;  
    public User authenticateUser(String username, String password) ;  
    public Collection getUsers() ;  
    public Collection getAllPortfolios() ;  
    public User getUser( String username ) ;  
    public User getUser( Long id ) ;  
    public Portfolio getPortfolio ( Long id ) ;  
    public String getDefaultUser() ;  
    public void persistUser ( User user ) ;  
}
```



```

    public boolean contains ( User user );

    public void updatePortfolio( Portfolio port );

}

```

There shouldn't be any surprises here. This interface simply defines the core business methods that our application uses. We could implement this interface with native Hibernate, JPA, XML files, raw JDBC, or anything else you like. You could even implement a mock service object for testing. For this chapter, we've implemented a JPA version of the service.

Now, our actions that depend upon a service object provide setters typed to this interface, rather than a specific implementation, to receive the Spring injections. With this done, changing the service object used by the actions is as simple as flipping a switch in the Spring metadata. Listing 9.3 shows the full source of our chapter 9 Login action, in which we've changed the setter to use the interface.

Listing 9.3 Login exposes a setter into which Spring injects the service object

```

public class Login extends ActionSupport implements SessionAware {      ❶

    public String execute() {

        User user = getPortfolioService().authenticateUser( getUsername(),
                                                                getPassword() );      ❷

        if ( user == null )
        {
            return INPUT;
        }
        else {
            session.put( Struts2PortfolioConstants.USER, user );
        }
        return SUCCESS;
    }

    private String username;
    private String password;

    //getters and setters omitted

    PortfolioServiceInterface portfolioService;

    public PortfolioServiceInterface getPortfolioService() {
        return portfolioService;
    }

    public void setPortfolioService(PortfolioServiceInterface
                                    portfolioService) {      ❸
        this.portfolioService = portfolioService;
    }

    private Map session;

    public void setSession(Map session) {      ❹
        this.session = session;
    }
}

```

Not much to look at. The service setter now takes an object of type `PortfolioService-Interface` ④. We've worked with this action before, but let's revisit a couple of points in light of our new understanding of dependency injection. To recap this action's business logic, it checks whether or not the login credentials returned a valid user ③. If valid, the user is stored in the session scope. But if the credentials don't map to a valid user, we return the user back to the input page, a.k.a. the login page. As we mentioned earlier, the `Login` action needs to access the session-scoped map because a `User` object will be placed in that map when a user successfully logs in. We know that Struts 2 is probably running on the Servlet API, so this map is actually the Servlet session map. But the framework has a strong commitment to loose coupling. In order to get a reference to this map in a loosely coupled manner, `Login` implements the `SessionAware` interface ①. The contract of this interface is fully satisfied by the exposure of a single setter method ⑤ into which the framework will inject a session map. At runtime, this map is most likely going to be the Servlet API map. Thanks, however, to the loose coupling of the `SessionAware` interface, you could easily set a mock map for testing purposes. In general, the `Aware` interfaces offered by the framework provide a good form of dependency injection. Unfortunately, they mostly just handle servlet-related things.

Thankfully, we can use Spring for injecting stuff like our service object. With Spring, the dependent object doesn't need to implement any specific interface, like the `Aware` interfaces, in order to receive the injection. Spring tries hard to keep your code independent of Spring. When it's time to conduct the business, the service object is just there. In this case, the `Login` action uses the service object to authenticate the user ②. It could be a mock service object that returns `true` every time, or it could be our JPA service object that makes a live check against our database. This is what Spring dependency injection is all about.

Now it's time to see the details of adding Spring to our Struts 2 application.

9.2 Adding Spring to Struts 2

In this section, we see how to add Spring to a Struts 2 application. It's quite easy. There are a couple of strategies for the actual injection of dependencies into your objects. We cover those in this section. But first we need to show you how to get Spring set up. The basic idea is that we need to give Spring a chance to handle the objects that are created by Struts 2. One way to let Spring do this is to provide a Spring extension of the Struts 2 `ObjectFactory`, the class that creates each of the objects used in the framework. We do just this in this section.

First, we need to download and add the Spring plug-in to our application. As you'll see in chapter 12, plug-ins can modify or enhance the core structure of the framework. One such example is the Spring plug-in. This plug-in provides a Spring extension of the core `ObjectFactory`. With this plug-in in place, Spring has the opportunity to manage the creation of any objects that the framework creates. Note that while the Spring `ObjectFactory` adds the opportunity for Spring to manage the creation of objects, it's not necessary for Spring to be involved in all object creation. Basically, the Spring `ObjectFactory` only intervenes when you tell it to; all other objects get created in the normal fashion.

You can find the Spring plug-in in the Struts 2 plug-in registry at <http://cwiki.apache.org/S2PLUGINS/home.html>. The plug-in comes as a JAR file, `struts2-spring-plugin-2.0.9.jar`. You also need to get the Spring JAR, `spring.jar`, found at www.springframework.org. With these two added to your `lib` directory, you just need a way to create the Spring container itself. Since we're building web applications, we can use a Spring application-context listener. This comes with the Spring JAR and is set up with the following snippet from our `web.xml` file:

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

At this point, you're completely ready to start managing your objects with Spring. But, as we indicated earlier, Spring won't just start handling all of your objects. You must tell Spring to intervene. To have Spring manage your objects, you need to declare these objects as Spring beans in a Spring configuration file. By default, the Spring container created by the `ContextLoaderListener` looks for metadata in an XML file at `/WEB-INF/applicationContext.xml`. You can pass in a parameter to the listener to specify different locations, and even multiple files, if you like. Consult the Spring documentation for this listener if you'd like to know more. As for the structure and usage of the XML metadata, we'll start to explore that in the next two sections as we explore some basic strategies for managing dependencies with Spring.

9.2.1 *Letting Spring manage the creation of actions, interceptors, and results*

With the Spring plug-in set up, it's time to put it to use. We start by showing how to let Spring directly manage the creation of framework objects such as actions, interceptors, and results. First, we'll let Spring handle the creation of our `Login` action. As we've seen, this action depends on our `PortfolioService` object. In the last couple of sections, we've already prepared the `Login` class for Spring injection by adding the setter into which we can inject a `PortfolioServiceInterface`-typed object. To put that setter to use, we need to tell Spring how to manage the creation of our `Login` action.

As we've hinted, one common way to tell Spring about the objects it should manage is with metadata contained in XML files. In our case, we use a file called `applicationContext.xml`. Listing 9.4 shows what to put in `applicationContext.xml` to have Spring manage the creation of a `Login` action, complete with the injection of a portfolio service object.

Listing 9.4 Telling Spring how to inject our service bean into our `Login` action

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

1

```

<bean id="portfolioService"
      class="manning.chapterNine.utils.PortfolioServiceJPAImpl"/>
<bean id="springManagedLoginAction" class="manning.chapterNine.Login"
      scope="prototype">
  <property name="portfolioService" ref="portfolioService"/>
</bean>
</beans>

```

At first glance, this might be an eyeful. But it's actually simple. The critical chunks are the declaration of two Spring beans, objects that Spring should manage for us. Before we get to those, let's look at that messiness at the start. We've got a huge bunch of namespace and schema stuff at the top ❶. Spring uses a lot of namespaces, so sometimes this stuff differs from one file to another depending upon the Spring functionality in play. You'll get by fine by copying ours unless you want to do something special with Spring. After that, we get to the good stuff, the declaration of two Spring beans. Notice that the root element of the `applicationContext.xml` file is the `<beans>` element ❶. All of the objects that Spring manages are known as *Spring beans*. We declare two.

The first of our beans is our JPA implementation of our `PortfolioServiceInterface` ❷. It's pretty accurate to just imagine this bean declaration as an instantiation of the object, or the definition of how to instantiate the object. In this case, we simply give the bean tag our fully qualified class name and an ID. The ID is used to identify the bean, much like a reference is used to identify an object in Java code. Next, we've told Spring to manage the creation of instances of our `Login` action. We've done this with another bean tag, giving it the class name and an ID ❸. We've also set the scope attribute to `prototype`. Why did we do this? By default, Spring beans are created as singletons. This won't work for Struts 2 actions, because they carry data related to each individual request. In order to make Spring create a new `Login` action bean each time one is needed, we must set the scope attribute to `prototype`. Our service object, like many application resources, works well as a singleton.

WARNING Make sure that your Spring-managed actions are configured to be created as new instances each time they're needed. By default, Spring creates singletons and reissues them when that bean is requested. You can force Spring to create a unique instance for each request if you set the scope attribute to `prototype`. Very important!

Now we get down to the business of wiring the portfolio service bean that we declared in the first bean tag into our `Login` action. We use Spring's property tag to do this ❹. The property tag looks for a setter for a property with the same name as the property tag's name attribute. The value to inject into this setter is then specified with the `ref` attribute, which takes a reference that points, in our case, to the ID we gave to our service object in the first bean tag.

Okay. We've told Spring how to create a couple of our objects for us. Whenever someone asks for the `portfolioService` or `springManagedLoginAction` bean, Spring

serves up a bean matching those IDs managed just as we've asked. If someone asks for a `springManagedLoginAction`, Spring creates a unique instance of our `manning.chapterNine.Login` class and injects a `portfolioService` Spring bean into it. But this doesn't mean that Spring will intervene any time a Login action is created. It won't do anything until someone asks, by ID, for one of the beans it manages. Every instance of a Login action in the system is not inherently the `springManagedLoginAction` Spring bean. So how do we make the framework ask Spring for this bean when it needs a Login action? Good question.

Normally, Struts 2 creates its action objects by instantiating the class defined in the declarative architecture metadata. If a request comes in with a URL that maps to the Login action, as defined in our declarative architecture XML or annotations, the framework consults the declaration of that action to find out which class should be instantiated. Here's how we've been declaring our Login action up until now:

```
<action name="Login" class="manning.chapterNine.Login">
  <result type="redirectAction">
    <param name="actionName">AdminPortfolio</param>
    <param name="namespace">/chapterEight/secure</param>
  </result>
  <result name="input">/chapterEight/Login.jsp</result>
</action>
```

This mapping tells the framework to create an instance of the `manning.chapterNine.Login` class to use as its action object for this request. As we've indicated, the framework has an `ObjectFactory` that normally handles all of this. Even with the `SpringObjectFactory` in place, via the Spring plug-in, this mapping is still handled in that standard fashion. If we want the framework to ask Spring to create one of its beans for us, we need to refer to that Spring bean's ID from within our Struts 2 action mapping, as follows:

```
<action name="Login" class="springManagedLoginAction">
  <result type="redirectAction">
    <param name="actionName">AdminPortfolio</param>
    <param name="namespace">/chapterEight/secure</param>
  </result>
  <result name="input">/chapterEight/Login.jsp</result>
</action>
```

Now, the framework asks Spring for a bean going by the name of `springManagedLoginAction` and Spring gladly returns that bean with the `PortfolioService` injected and ready to go.

That's about all you need to know to have Spring directly manage the creation of framework objects such as our Login action. You can do the same thing with interceptors or any other framework components. Oddly enough, we don't recommend using Spring in this manner for most situations. We showed this technique first because it's the most straightforward way of understanding what Spring does. But it's not always the best way to use Spring for dependency injections. If you want to take advantage of

some other Spring-fu, like some of its aspect-oriented features, then you'll need to use this heavy-handed, direct object management technique. But if all you want to do is inject dependencies, such as the `PortfolioService` object, there's a much easier method: autowiring! Let's have a look.

9.2.2 **Leveraging autowiring to inject dependencies into actions, interceptors, and results**

The direct management of your actions, interceptors, and results by Spring is a perfectly good way to do things, but it's verbose. To make things super easy, you'll want to take advantage of Spring's ability to autowire dependencies. Autowiring is a way to inject dependencies without explicitly declaring the wiring in your application-`Context.xml`. In other words, we can have the `PortfolioService` object automatically injected into all the actions that need it without actually having to generate any meta-data regarding those actions. You don't have to do anything to enable autowiring; it's on by default. There are several flavors of autowiring. You can autowire by name, type, constructor, or something called auto. We cover each of them in this section.

AUTOWIRING BY NAME

The default behavior of the Spring plug-in autowiring is by name. Autowiring by name works by matching the ID of a managed Spring bean with setter method names exposed on potential target objects. As it turns out, every object created by the framework is a potential target object. Since all of our actions are clearly created by the framework, they're all potential targets of autowiring without any explicit intervention on the behalf of the developer. Listing 9.5 shows what our `applicationContext.xml` would look like if we rely on name-based autowiring instead of direct Spring management of our dependencies.

Listing 9.5 Autowiring requires less metadata in our `applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="portfolioService"
        class="manning.chapterNine.utils.PortfolioServiceJPAImpl"/>

</beans>
```

As you can see, we now do nothing more than declare the bean that we need to inject, our `portfolioService` bean, and give it an ID. There are no bean tags telling Spring how to manage our action objects. In fact, Spring won't handle the creation of our action objects in this case. Nonetheless, the framework lets Spring do a postcreation inspection of all objects that it creates, such as actions and interceptors, and try to autowire them with beans that it knows about. In our case, we've told Spring about our `portfolioService` bean. Spring tries to automatically inject this bean into any framework object that exposes a setter with a name that matches the ID of the service

bean. To receive the automatic injection of this `portfolioService` bean, a framework object just needs to expose a setter like the following:

```
PortfolioServiceInterface portfolioService;

public void setPortfolioService(PortfolioServiceInterface portfolioService)
{
    this.portfolioService = portfolioService;
}
```

The key here is that the setter naming convention matches the Spring bean ID. All objects created by the framework are subject to this injection. This isn't just actions. Our `manning.chapterNine.utils.AuthenticationInterceptor` also receives this injection. Due to the ease and elegance of autowiring by name, this'll probably be your preferred Spring usage as long as your need for Spring doesn't extend past dependency injection.

Before moving on to the next method of autowiring, we should make one point. In the previous section, where we had Spring actually create our action objects for us, we had to write our declarative architecture to point to Spring IDs rather than the normal class names. Since Spring is no longer creating our action objects, we can revert our declarative architecture back to the previous form. In other words, we no longer map our action element to a Spring ID name; we map it to the class name again, as seen in the following snippet:

```
<action name="Login" class="manning.chapterNine.Login">
  <result type="redirectAction">
    <param name="actionName">AdminPortfolio</param>
    <param name="namespace">/chapterNine/secure</param>
  </result>
  <result name="input">/chapterNine/Login.jsp</result>
</action>
```

The class attribute now points to an actual Java class rather than the ID of a Spring bean. With autowiring, the framework creates the action, then Spring inspects to see whether it can inject anything. This is a subtle but different mechanism than we discussed in the previous section.

Autowiring comes in several other varieties. Let's consider our options.

AUTOWIRING BY TYPE, CONSTRUCTOR, AND AUTO

Spring provides several other means of autowiring that are also available to us in the context of Struts 2. In the interest of space, we won't demonstrate these other strategies, but we'll give brief explanations of what they do. They're quite simple. The first step to using an alternative method of autowiring is configuration. If you want to change the method of autowiring, you need to set a Struts 2 configuration property. You need to add something like the following to your `struts.properties` file:

```
struts.objectFactory.spring.autoWire=type
```

Or, as with all Struts 2 configuration settings, you can set this property via a constant element in one of your XML files, such as `struts.xml`, or `chapterNine.xml` if you like. Here's what the constant element would look like:

```
<constant name="struts.objectFactory.spring.autoWire" value="type"/>
```


It doesn't matter which way you do it. Now, let's talk a bit about what each of these alternative autowiring methods means.

Essentially, they all do the same thing. They tell Spring to inspect the objects created by the framework for places where it can inject the beans that it knows about. If you choose to do autowiring by type, rather than the default name, Spring will try to match the type of the beans it knows about to the types of setter methods it finds on the objects created by the framework. The secret here is that you should only tell Spring about one bean per type; otherwise it can't figure out which bean to inject and it complains. The only thing faintly subtle about autowiring by type is that interfaces count. In other words, since the `manning.chapterNine.utils.PortfolioServiceJPAImpl` bean that we told Spring about implements `manning.chapterNine.utils.PortfolioServiceInterface`, it's autowired to setters of the interface type. Since the setter on our `Login` action takes type `PortfolioServiceInterface`, this injection will occur automatically if we change the `autoWire` property to `type`. This is another sign that interfaces are the right thing to do!

The next two alternative methods of autowiring are `constructor` and `auto`. If you choose `constructor`, your objects must have a constructor that takes its dependencies as parameters. Beans known to Spring will be injected into the constructor parameters as they match up by type. This has the same limitations as the type-based method discussed in the previous paragraph. If more than one Spring bean of the matching type is available, Spring will throw an exception rather than try to decide which one to inject. The last method of autowiring is `auto`. This method simply tries to inject by `constructor` first and then by `type`.

And that's it. Spring's not such a big deal, huh? Well, this is a small part of Spring. Moreover, it's a tribute to the power of Spring that the end result can look so effortless. Nonetheless, this simple use of Spring can make your code much more maintainable and testable, the twin joys of loose coupling. Keep in mind that we've only shown you how to use the dependency injection features of Spring, but it offers a lot more than just DI. Many of you will want to take advantage of some of that other stuff in your Struts 2 applications. With limited book space, we can only point the way. One thing to keep in mind is this. If you want to add further Spring management to your beans, such as Spring's aspect-oriented programming features, you need to let Spring directly manage your actions, interceptors, and results. We showed how to do this in section 9.2.1. Spring has a lot to offer and the Spring plug-in makes it easy to leverage these offerings from a Struts 2 application. Enjoy.

Now it's time to see how we can make another drastic improvement to your Struts 2 Portfolio. In the next section, we're going to introduce the powerful Java Persistence API (JPA) and show how to let it manage your application's data persistence needs.

9.3 **Why use the Java Persistence API with Struts 2?**

In this section, we show you how to integrate the Java Persistence API (JPA) into your application. This technology, wrapped around Hibernate, represents the bleeding

edge of enterprise Java data persistence. Coding to the JPA works a lot like coding to native Hibernate, but its interface-based architecture gives you the ability to switch out the vendor supplying the underlying implementation. In our examples, we use Hibernate. Still, the techniques we demonstrate are applicable regardless of which underlying engine you choose.

The topics of JPA and Hibernate go beyond the scope of this book. This section doesn't intend to teach you anything specific about JPA and Hibernate. The purpose of this section is to demonstrate a best practice of integration with those technologies. If you don't know anything about JPA and Hibernate, you'll need to seek basic instruction elsewhere. We recommend the Manning title *Java Persistence with Hibernate*. If you're already familiar with these topics, we build on that by walking you through one of the most successful solutions to integrating JPA into a Struts 2 web application. This solution solves two of the most immediate issues a developer faces when integrating JPA with a web application. First, we show how to use a servlet filter to solve the infamous Open Session In View problem. Second, we show how to use the well-respected Spring support for JPA to make management of the JPA all that much easier.

Before we get to the integration, we should start with the mundane, but essential, details of setting your project up to use JPA with Hibernate.

9.3.1 Setting your project up for JPA with Hibernate

First, there's no plug-in or anything for integrating JPA with your Struts 2 application. Setup includes adding a few JAR files, a configuration file or two, getting a database, and some other odd bits or two. We'll sail right through these in no time, starting with the issue of JAR files.

COLLECTING THE APPROPRIATE JAR FILES

There's quite a list here. Some of these resources belong to Hibernate and some belong to the JPA. There's also the issue of a driver for your choice of database. Some libraries are already in the project; some we need to add just for this chapter. As always, these kinds of things change over time; a trip to the Hibernate, JPA, or the MySQL website will yield the most accurate and up-to-date word on dependencies.

At the time of writing, here's what we're using:

- antlr.jar *
- asm.jar
- asm-attrs.jar
- c3po.jar
- cglib.jar
- commons-logging.jar *
- commons-collection.jar *
- dom4j.jar
- hibernate3.jar
- jta.jar
- hibernate-annotations.jar
- hibernate-commons-annotations.jar
- ejb3-persistence.jar
- hibernate-entity-manager.jar
- mysql-connector-java-5.1.5-bin.jar
- jboss-archive-browsing.jar
- javassist.jar

The ones with asterisks are the ones that would already be in a Struts 2 project. The rest are all specific requirements of our current persistence work.

CHOOSING A DATABASE

You can use just about any database you like. We use MySQL because it's familiar to many, including ourselves. If you opt to use another database, then you should switch out the MySQL driver in our list of resource dependencies. In addition, you'll need to change a couple of settings in the data source and Hibernate configuration found in the `applicationContext.xml`. We'll cover that file in a moment.

After you install a database, you just need to do the prerequisite admin stuff to set up the account by which you'll connect to the database. While you can change the settings, our current configuration expects to find a database account with username/password equal to `manning/action`. This database user must, of course, have the appropriate rights to create, update, alter, and so forth.

With the database and resources all in place, it's time to configure JPA. As we said, we're going to leverage Spring's built-in support for JPA to do this. Since we've taken the time to integrate the Spring container, we should take the opportunity to really benefit from it.

USING SPRING TO MANAGE OUR JPA DEPENDENCIES

After all the trouble of setting up the Spring plug-in, we might as well make the most of it. Actually, using Spring to manage JPA is a nice thing. Spring comes with packages dedicated to making the management of JPA easy and powerful. By the end of this chapter, we think you'll have to admit that the combination is elegant. Let's jump right in by examining our new `applicationContext.xml` to see just exactly what we're asking Spring to do for us. Listing 9.6 shows that Spring configuration file in full glory.

Listing 9.6 Letting Spring manage our JPA dependencies and transactions

```
<?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:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.0.xsd"

  <bean class="org.springframework.orm.jpa.support.
    PersistenceAnnotationBeanPostProcessor" />

  <bean id="portfolioService" class="manning.chapterNine.utils.
    PortfolioServiceJPAImpl"/>

  <bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.
        HibernateJpaVendorAdapter">
```

```

        <property name="database" value="MYSQL" />
        <property name="showSql" value="true" />
    </bean>
</property>
</bean>

<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.gjt.mm.mysql.Driver" />
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/manning" />
    <property name="username" value="manning" />
    <property name="password" value="action" />
</bean>

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

</beans>

```

This might look like a lot at first, especially if you're unfamiliar with the JPA. But we can remedy all that quickly. Here's a nutshell version of what's happening. All we're really doing is having Spring create our JPA `EntityManager` and inject it into our service object. That's it. And, of course, we're still having the service object autowired into our actions. But everything here is all about creating the `EntityManager` and injecting it into our service object. Now, let's pick it apart line by line to take the sting out of it.

First, we've got a gigantic wad of namespace and schema stuff ❶. Too bad the printed word doesn't do code folding yet. Next, we declare a bean postprocessor that checks all the beans managed by Spring for persistence-related annotations ❷, such as the annotations that mark which setter methods should be injected with the `EntityManager`. We put one of these annotations in our `PortfolioServiceJPAImpl` class. Next, we have the familiar declaration of our service object ❸ as a Spring bean. As we've seen, this is autowired by name into our actions and interceptors. This hasn't changed in the slightest.

All of the rest of this mess sets up our JPA stuff. And it's not really messy at all. The main entry point into the JPA is something called the `EntityManager`. This object manages all of your persistent entities—our users and portfolio objects. Our `PortfolioServiceJPAImpl` uses an `EntityManager` to read, write, and update our objects. If you're familiar with Hibernate, the `EntityManager` is equivalent to the Session. Rather than telling Spring how to create an `EntityManager`, we need to tell Spring how to create an `EntityManagerFactory` ❹. This is also where we configure our JPA persistence unit. As you can see, we tell Spring which JPA vendor we're going to use as well as which data source we're going to use. The data source that we wire to the factory is another Spring bean ❺, which we've configured to work with our MySQL database and the database account we created.

Finally, since all JPA and Hibernate work must occur within transactional boundaries, we register a transaction manager with Spring ⑥, which we also wire to the `EntityManagerFactory`. We then tell Spring that we'll use annotations to tell the transaction manager about our transactional boundaries ⑦. You can also define transactional boundaries in the XML, but we're using transaction annotations in our `PortfolioServiceJPAImpl` class. Once you get familiar with it, it's fairly elegant.

That's the Spring part of it. Next, we look at how to handle the problem of lazy loading in the view layer.

HANDLING LAZY LOADING WITH SPRING'S `OPENENTITYMANAGERINVIEW` FILTER

If you've used Hibernate before, you're probably familiar with the view-layer lazy loading issue. To summarize the problem, when you retrieve Java objects from a persistence technology such as JPA, optimizations are made to reduce traffic with the database. One of the primary optimizations is the lazy loading of deeper elements in the data structure contained by the retrieved object. Let's say we retrieve a `User` from the JPA `EntityManager`. All of the `Portfolios`, for instance, might not be loaded when this `User` is first loaded. In fact, they might not be loaded until they're referenced. This is *lazy loading*.

In an MVC web application, the action classes generally load the data from the database, such as the `User` mentioned previously. Then they forward control over to the view layer, a JSP result, let's say. Many times, data such as a given `Portfolio` referenced within that `User` won't be read until a JSP tag iterates over that set of `Portfolios` while rendering the result page. If the persistence context, the `EntityManager` in the case of the JPA or the `Session` in the case of native Hibernate, has been closed, this attempt to read the unloaded `Portfolio` data will fail because lazy loading is no longer available.

A well-known fix to this has been around for some time. In Hibernate terms, the fix is known as the `OpenSessionInView` pattern. This fix typically uses a servlet filter or some kind of interceptor to wrap a Hibernate `Session` around the entire request-processing pipeline, including the view layer. You can find many examples of this on the Web. In JPA terms, we need a `OpenEntityManagerInView` fix. Fortunately, Spring provides a servlet filter implementation of this fix that's widely used: one of the key reasons for going with the Spring support for JPA. We don't need to do anything other than configure this filter in our `web.xml` file, as seen in Listing 9.7.

Listing 9.7 Configuring the `OpenEntityManagerInViewFilter`

```
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
</filter>

<filter>
  <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
  <filter-class>
```

```

    org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>struts2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

If you're familiar with web.xml files, there's nothing complicated here. There's one critical thing, though. The mapping of the Spring filter must come before the mapping of the struts2 filter; otherwise nothing will work. Once in place, this filter creates and binds an EntityManager to the thread that's processing the request. It uses the factory declared in our Spring container to do this. This threadbound EntityManager will always be used by our other Spring-controlled JPA code. You don't have to do anything in your code. It's very convenient.

With all of that in place, we're now ready to look at how it all works. The next section walks us through the code that uses the JPA as we've configured here.

9.3.2 Coding Spring-managed JPA

In the previous section, we showed you how to set up your Struts 2 application to use JPA. In addition to adding the resources for JPA and Hibernate, we showed you how to leverage the Spring plug-in we added earlier in this chapter to manage your JPA resources. With all that in place, we're now ready to examine the code-level implementation of a JPA persistence layer. In our Struts 2 Portfolio, this takes place inside the PortfolioServiceJPAImpl class.

We start with a quick discussion of the JPA persistence unit.

THE PERSISTENCE UNIT

When you use the JPA, the entirety of your persistence-related pieces is known as a *persistence unit*. The EntityManagerFactory managed by Spring encapsulates most of the details of the persistence unit. These details include everything from the metadata that describes how your Java classes map to database tables to the database connection details. The main entry point into configuring your persistence unit is the persistence.xml file. Just as a web application must have a web.xml file, a JPA project must have a persistence.xml file. Listing 9.8 shows the full details of this important file, which can be found in the project at /WEB-INF/classes/META-INF/persistence.xml.

Listing 9.8 The mandatory configuration entry point for a JPA persistence unit

```

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
    java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

```

```

<persistence-unit name="struts2InAction">
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create"/>
  </properties>
</persistence-unit>

</persistence>

```

It's of interest that such an important file contains so little. Mostly, this is because we're deferring the management of our JPA details to Spring. The important thing here is the name of the persistence unit and the fact that we pass in a Hibernate property to control the autocreation of our database schema. We're using the development-friendly create setting. Every time the application is started, the database schema will be created anew. This means that your data won't persist over multiple application startups. If you want to preserve the database from a preceding state, just take this out before restarting the application.

The next critical element of a JPA project is the metadata that maps your Java classes to database tables. The next section covers this important topic.

USING ANNOTATIONS TO MAP OUR JAVA CLASSES TO DATABASE TABLES

One of the great things about using JPA is that we can use Java annotations to map our Java classes to the database. Though people, including myself, can debate whether annotations are fluff or stuff, it's hard to deny some of the benefits of using them. A few of the benefits of using the JPA annotations in this context include IDE support, less verbosity, automatic scanning for annotated classes, and type checking—annotations are actual Java types after all.

We're going to keep it simple, so let's start by cracking the source code to our persistent entities, a.k.a. our `User` and `Portfolio` classes. Listing 9.9 shows the source for our `manning.chapterNine.utils.User` class.

Listing 9.9 This version of the `User` class is a JPA persistent entity.

```

@Entity ❶
public class User {

    private String username;
    private String password;
    private String firstName;
    private String lastName;

    @OneToMany ( cascade={CascadeType.ALL }, mappedBy="owner") ❷
    private Set<Portfolio> portfolios = new HashSet<Portfolio>();

    @Id @GeneratedValue ❸
    private Long id;

    //getters and setters omitted

    public Set getPortfolios()
    {
        return portfolios;
    }

    public void setPortfolios( Set portfolios ) {
        this.portfolios = portfolios;
    }
}


```

```

    }

    public void addPortfolio ( Portfolio portfolio ) {
        portfolio.setOwner( this );
        portfolios.add( portfolio );
    }
}

```



This class uses annotations to tell JPA how to map its properties to the database. The entity annotation ❶ marks this class as one that should be scanned by the JPA and mapped to the database. This is required. Next, note the absence of any metadata describing mappings between the properties and the database. We could annotate these, but we can also sit back and let JPA generate sensible database column names for each of these properties. They will each be columns in our User table. We then map an association to our collection of Portfolio objects that belong to this user ❷, and annotate our id property ❸. This id property is the only actual change we have to make to our class to make it work as a persistent entity managed by the JPA Entity-Manager. As a final point of interest, we'd also like to point out that while we can rely on the JPA to manage our persistence, we still have to manage our Java relationships. In the interest of this, we have an addPortfolio method that manages the directional references between a Portfolio and the owning User ❹. This is required so that we can track the User from the Portfolio, if need be.

If you check out the `manning.chapterNine.utils.Portfolio` class, you'll find similar annotations. Again, these annotations are automatically located and scanned by JPA when it fires up. You don't have to register the annotated classes anywhere.

At this point, we've done everything we need to do. We have annotation metadata that describes how we want to map our Java classes to the database. We have a database. We've configured the Spring container to create our JPA EntityManagerFactory with all the necessary settings. When the EntityManagerFactory starts up, it scans our annotated classes and automatically creates the schema in the database. The only thing left is to get our hands on an EntityManager and start writing the JPA persistence code in the `PortfolioServiceJPAImpl` class. So that's what we'll do now.

USING THE JPA ENTITYMANAGER TO IMPLEMENT OUR SERVICE OBJECT

We're finally ready to look at the `PortfolioServiceJPAImpl` class itself and see what the actual JPA code looks like. Be forewarned, it doesn't look like much. As we indicated earlier, the actual work of JPA code will mostly involve an instance of the EntityManager class. This is why we've asked Spring to set up an EntityManagerFactory for us. Somewhere along the line, we need to get a reference to an real EntityManager in our code. As you might've guessed, we're planning to have Spring inject it. But how? Autowiring? Not quite, but close.

You might've noticed that we didn't actually declare an EntityManager bean in our Spring `applicationContext.xml` file. So we can't use autowiring to inject one into our service object. But we can take advantage of the Spring support for JPA. We can use annotations to tell Spring where to inject an EntityManager. The following snippet shows the annotation on our `PortfolioServiceJPAImpl`'s entityManager setter:

```

@PersistenceContext
public void setEntityManager(EntityManager entityManager) {
    this.entityManager = entityManager;
}

```

The `PersistenceContext` annotation indicates that Spring should inject an entity-manager at this setter. The naming of the setter is unimportant. It's not autowiring by name. This works because of two things. First, we declared the following bean postprocessor in our `applicationContext.xml`:

```

<bean class="org.springframework.orm.jpa.support.
    PersistenceAnnotationBeanPostProcessor" />

```

This postprocessor looks for the `PersistenceContext` annotations. Second, we declared an `EntityManagerFactory` in our `applicationContext.xml`. Without this factory, Spring wouldn't know where to get the `EntityManager` to inject in the annotated setters.

Now we have the `EntityManager` in hand and we're ready to write code. Listing 9.10 shows how we get down to the business of using the `EntityManager` in our `PortfolioServiceJPAImpl` class.

Listing 9.10 Using the JPA to manage the persistence of our Users and Portfolios

```

@Transactional
public class PortfolioServiceJPAImpl
    implements PortfolioServiceInterface {

    public boolean userExists ( String username ) {

        Query query = entityManager.createQuery ( "from User where
            username = :username" ).setParameter("username", username);
        List result = query.getResultList();

        return !result.isEmpty();
    }

    public void updateUser( User user ) {
        entityManager.merge( user );
    }

    public Collection getUsers() {

        Query query = entityManager.createQuery ( "from User" );
        return query.getResultList();
    }

    public Portfolio getPortfolio ( Long id ) {
        Portfolio port = entityManager.find(Portfolio.class, id);
        return port;
    }

    public void persistUser ( User user ) {
        entityManager.persist(user);
    }

    private EntityManager entityManager;

```



```

@PersistenceContext
public void setEntityManager(EntityManager entityManager) {
    this.entityManager = entityManager;
}
}

```

As we've indicated, we don't want to present a primer on JPA. We'll just point out the highlights of how we're using the injected `EntityManager`. First, we take advantage of Spring transaction management and the accompanying annotations to declare that this class is transactional ❶. This means that every method in the class will be transactional. Typically, you'd want to pursue a more fine-grained approach to the description of your transactions, but this works for us. In JPA, as in native Hibernate, all data access has to occur within the bounds of a transaction. You can open and commit these transactions programmatically, but you can also let Spring do it by configuring a transaction manager as we did in our `applicationContext.xml`. We also chose to use annotations to describe our transactions.

With transactional control out of the way, we can start persisting, loading, and updating data. The JPA offers a full API to make this work easy and clean. We can write queries against the database with a high-level query language that allows us to use our Java names instead of the database names that'd be required by native SQL ❷. We can update the database by merging new data objects with the persistence context ❸. We can, of course, retrieve an object ❹ and persist a new object ❺. While this can't teach you JPA, it does serve as a clean demonstration of a full set of CRUD functionality implemented in JPA.

This object, injected with the `EntityManager`, is then injected via autowiring into all of our actions that need to use it. They simply call these methods to persist their objects. Furthermore, the actions only handle an interface. You could easily switch this JPA implementation out for any other implementation, including test mockups.

Oh my, that was a fast run through a rather sophisticated technology. Again, our point here is not to teach JPA, but to demonstrate a best-practice integration technique. Based on our experience and our conversations with others doing Struts 2 web applications, this combination of JPA with Spring support is hard to beat. We hope you enjoy it.

9.4 Summary

In this chapter, we've seen how to integrate a Struts 2 application with two of the more commonly used third-party technologies. Both of these technologies, Spring and JPA, offer much more than we could've shown you in this brief chapter. Some JPA books are 800 pages long. Rather than try to tell you about those technologies, we've focused instead on showing you some best practices concerning the integration itself. Let's review what we've seen.

First, we saw how to use the Spring plug-in to extend the framework's `ObjectFactory` with a `SpringObjectFactory` that gives you the opportunity to manage framework objects in a Spring container. While we hinted that this management could

offer you more bonuses than we have the time to discuss, such as AOP, we did show you what many people consider to be the bottom-line in Spring usage for a Struts 2 application: dependency injection. We saw how to inject dependencies into managed beans and how to use several varieties of autowiring to inject dependencies into objects that the framework creates in the normal fashion. Dependency injection is great, hence all the hoopla. But we encourage you to check out all the other stuff that Spring has to offer its managed beans.

Finally, we took on the large task of upgrading our service object to use JPA. We went on a whirlwind tour of using Spring's support for JPA to make management of the JPA...well...manageable. Our fast coverage of JPA covered the critical issues of integrating with a persistence technology such as JPA. We showed you how to set up the `OpenEntityManagerInView` filter. We also showed you how to use Spring to inject the `PortfolioService` into the actions that depend on it. Somewhere in the midst of all that, we gave you a peek at using JPA annotations to map Java classes to the database, as well as Spring-managed transactions.

With all of this dense information crammed into a single chapter, we have no doubt that some of you are breathless. We fully expect that you might need to consult a Hibernate or JPA resource. They are complex topics that offer a lot to those willing to come to grips with their full breadth and depth. However, once you have the persistence technology background, we think this chapter will provide you with a good case study for getting that technology into your Struts 2 application.

With our core application shaping up, we can pick off a couple more points of refinement before moving into the advanced topics section of the book. Next up, we look at improving our validation code in chapter 10.