

Session 5 Readings (Spring A Developer's Notebook (SADN) Preface, Ch. 1-5)SADN Preface

1. (p. ix-xi) Several key shortcomings of EJB, which drove the initial development of the Spring framework, are discussed. The shortcomings seem serious to me (especially the difficulties in testing). In contrast, Spring makes testing straightforward, and uses POJOs (most of which contain no references to Spring classes) as a primary design pattern.

SADN Chapter 1 (Installation and Setup)

Note on discussion below: the chapters are divided up into sections, with associated examples. The authors call these "labs", and I'll follow that standard

1. (p. 1-5: ch1_lab1) The initial discussion is intended to expose shortcomings that can arise when objects are related (a "dependency" exists) but no external means exists to connect them ("wire them together", in Spring terms). The term "façade" is used to describe the class (RentABike) that acts as a front end to the application, creating a static set of Bike objects. The CommandLineView class is used to execute the application and print the bikes.
2. (p. 6-8: ch1_lab2) In the second example, the RentABike class becomes a Java interface (we will discuss interfaces in class) that is implemented by the ArrayListRentABike class. The notion of "RentABike" (renting a bike) is thus exposed as a service. Note the highlighted code in Examples 1-6 and 1-7: the data element in CommandLineView is of type RentABike, but the implementation in RentABikeAssembler (interface) is of type ArrayListRentABike (implementation). We will see this pattern repeatedly in the examples.
3. (p. 10-11) The third example automates the build process using Ant, and is otherwise identical to the second example. I used Ant for all the examples, so there was no need to implement this one. We'll look at Ant in class as part of the Spring discussion.
4. (p. 12-16: ch1_lab4) At last Spring enters the picture. The key to understand how Spring works is to look at examples 1-9 and 1-10 on p. 13. The xml file establishes the connection between the two objects (ArrayListRentABike and CommandLineView): look the "<ref bean=" tag to see the connection. When RentABikeAssembler is invoked, the .xml file is read, and the beans are automatically instantiated (note the lack of "new" statements in Example 1-10, other than the one to create the Spring context). This section also shows a unit test, using another amazing Java technology called JUnit. JUnit can be used to invoke unit tests, as can be seen in Example 1-11. It's important to note that Spring is not used in the unit tests; the classes can be tested from outside the Spring framework. This is a huge benefit over other competing technologies, e.g. EJB.

SADN Chapter 2 (Building a User Interface)

This chapter's focus is on adding a user interface to the RentABike application; unfortunately the Spring MVC framework (a competitor to JSF which is built into Spring) is used. This chapter also includes a discussion of deploying to Tomcat using Ant. Since we have a Tomcat installed already, we'll deploy to that rather than the built-in Tomcat that comes with NetBeans.

1. (p. 17-20: ch2_lab1) A simple JSP is deployed to Tomcat. Note the embedded JSP code at the top of Example 2-2 that creates the ArrayListRentABike object.
2. (p. 21-27: ch2_lab2) In this example, the Java Standard Tag Library (JSTL) is used along with Spring to decouple the model and view, as we've done with JSP. Example 2-6 is actually

include.jsp, not editBike.jsp (note include.jsp is included at the top of Example 2-7). The JSTL tags should look familiar; they're similar to the JSF tags.

The Controller classes (Examples 2-8 to 2-10) are the controller classes for each JSP page. Not that they all implement the Spring controller interface, each acts upon a RentABike (façade), and they each implement the `handleRequest` method (which looks suspiciously like a servlet request, by design) that returns a `ModelAndView` object. The `ModelAndView` is exactly what it sounds like—a combination of model and view that will be used next by the application.

Examples 2-11 and 2-12 show the “wiring” of the application. The web application is set up to pass all files with an extension of “.bikes” to the Spring dispatcher servlet, and the .xml document is used by spring to map everything when Spring starts up. Note the URL mapping at the bottom of p. 26, which shows a “.bikes” file and its associated controller.

3. (p. 28-34: ch2_lab3) Now the power of Spring's MVC framework starts to appear. Several new concepts are introduced (resolvers, forms, and validators). The effect of all this is to give complete programmatic control over how views are displayed and which view is chosen next (for example, what should the application do if a validation error occurs?). Spring bind variables, which are very similar to JSF bind variables, are also used.

Example 2-16 shows a Spring validator, which is used to inject validation code into the application. In Example 2-17, the validator is added to the Spring context (note the validator is just another POJO) and referenced from the `editBikeForm` bean.

4. (p. 35-36: ch2_lab3) The last section adds unit tests to test the application controller and do some test validation. The key issue here is that no servlet container (Tomcat etc.) is needed; the tests can be run from the command line, even though we're implementing a web application. I included this with lab3 rather than creating a separate lab because the source files were identical for both.

SADN Chapter 3 (Integrating other clients)

In this chapter, Struts and JSF are used to build client front ends to Spring. Because we've implemented using JSF for other projects, we'll skip the Struts content.

1. (p. 37-48) An example using Struts is developed. For whatever reason, the book's authors did not include this example in the downloads on their site.
2. (p. 49-55: ch3_lab1) The section starts off with some discussion of value bindings, which we've seen previously. These are available “out of the box” with JSF, a big advantage over Struts. There's also a mention of values that bind to methods (see an example at the bottom of p. 51), allowing code to be executed when, for example, a link is clicked. On p. 53, note the validation code in bold; this could show an error (in red) if a manufacturer is not available, if a model doesn't match the manufacturer, etc.

The `faces-config.xml` entries and the controller bean are shown on p. 54-55. The navigation is straightforward, but the controller is a little different from what we've seen previously; there's a nice separation of concerns, in the sense that all the controller does is manage navigation: it's much clearer what's going on.

3. (p. 56-58) The section starts with some discussion of value bindings and their associated `JavaBean` getters/setters. This is a nice quick review if this concept is still fuzzy. Finally, the magic of Spring is revealed yet again; since JSF and Spring both use POJOs, the Spring wiring logic runs without modification, other than to tell JSF to use Spring-managed beans, rather than “plain old `JavaBeans`” (see discussion in the middle of p. 57). That's all there is to it!

SADN Chapter 4 (Using JDBC)

1. (p. 59-62) This section covers MySQL setup, which we've seen previously.
2. (p. 63-67) Spring JDBC templates allow developers to let Spring take over some of the database "grunt work", as noted in p. 63. The use of templates, which allow a developer to say "run this SQL, and when you're done with it, run this method against the result set" is also very clever. Java "inner classes", which are just inline classes created for use by the templates, are discussed on p. 64. The benefits of this approach are noted on p. 66-67; if you compare the examples in this section with what we used for the traditional JDBC example, the difference is obvious.
3. (p. 68-69) A small amount of refactoring (extracting common code and calling it rather than repeating it) is performed in this section.
4. (p. 69-71) Spring includes classes that implement database "operational objects", which can be used to wrap stored procedures, update queries, etc. in Java classes. This is a little bit of Java trickery, but it does allow code like Example 4-9, where a query object is created on the fly to query customers and bikes (note the similar coding style for both; this is normally considered a benefit from a reusability perspective).
5. (p. 72-76) Mock objects allow testing of Java classes without requiring access to external resources, e.g. databases. They also allow consistent, repeatable testing behavior. Spring's architecture allows mock objects to be bolted on easily and used to run mock tests, in this case to simulate database activity.

SADN Chapter 5 (OR Persistence)

This chapter discusses Spring's integrations with persistence frameworks besides Hibernate (iBatis, JDO). We'll skip these sections, the methodology is fairly similar across the frameworks.

1. (p. 90-96) An example showing Hibernate mapping files is given first, followed by a servlet mapping file (Example 5-14; note the class name for the sessionFactory bean, indicating that Hibernate support is built in to Spring). The next example (5-15) shows a very clever overlap between Spring and Hibernate: a Hibernate class (HibernateDaoSupport) is inherited from, but the RentABike interface is extended, allowing Spring to wire everything together. Notice the various database queries that are run when the interface methods are implemented.

There's a nice "finale" on p. 95; a sample method shows what would be required to write the getBikes() method without Spring. Contrast that with the three lines of code to implement getBikes on p. 93, and you'll get a sense of what all the excitement is about.