SAMPLE CHAPTER

FLEX ON JAVA

Bernerd Allmon
Jeremy Anderson

FOREWORD BY JAMES WARD

MANNING

Flex on Java

by Bernerd Allmon
Jeremy Anderson

Chapter 2

Beginning with Java

2

This chapter covers

- Generating the application structure with Maven
- Building Java server-side domain objects and services
- Building a simple JSP UI

We'll begin by creating a Java application that will expose web services so we can later connect to them from a Flex client. We have attempted to avoid tying the book to a specific sample application by focusing more on the concepts and techniques of using various frameworks and tools. This should allow you to pick a topic in the book that interests you and get rolling on it. We'll demonstrate many topics by using an application built in this chapter called FlexBugs. If you want to follow the samples in the book you can download the full code listings on the book's website at http://manning.com/allmon. You could also replace the application contents with something that's more meaningful to you by changing the domain objects to manage whatever you want, like contacts or movie favorites.

Throughout the book, especially in this chapter, we leverage a few Java frameworks that help to lighten the amount of work required to build a fully functional web application. This chapter is a bit mechanical because we need to set up a development environment. A few downloads and installs must take place if you choose to use our samples. Feel free to browse through this chapter and skip what you already know.

The Java frameworks used will help keep development to a minimum while creating a sample application to work with for integration purposes. This will allow us to focus on teaching and demonstrating how to build synergy between Flex and Java.

We're building a Java application first as a basis for work in chapter 3, but you can start with Flex in chapter 3 if you'd like or move around the book as convenient. The Java application comes first because we expect most readers to be refactoring existing applications to include a Flex client and this will give you something to play with quickly.

We'll start by generating the project structure with Apache Maven, a convention-over-configuration project management framework. Maven will build the application for us and speed up the development process. After we have a project structure generated, we'll start building the server-side components while leveraging MySQL for the database.

For the Java server-side pieces, we'll start with creating plain old Java objects (POJOs), Data Access Objects (DAOs), and service objects that will be exposed to a web tier.

Let's write a simple Java server-side application using the AppFuse framework. AppFuse was created by Matt Raible of Raible Designs to simplify the construction of Java web applications through convention. Using AppFuse on the server side will allow us to focus on the integration of Flex with Java creating simple domain and service Java objects.

2.1 Working with AppFuse

Because the layers of architecture and complexity can make approaching the building of a Java web application a bit daunting, AppFuse is a great technology choice because it simplifies dealing with the layers and delivering value faster.

AppFuse allows a Java developer to quickly start focusing on business domain concerns. A typical Java application will be POJO-driven and wired together through Spring, the open source dependency injection (DI) framework. The DI design pattern helps to build applications with loosely coupled components making your application more flexible and testable. In addition, AppFuse comes stocked with Maven integration to make things even easier. Let's get things rolling by installing Maven.

2.2 Generating the application structure with Maven

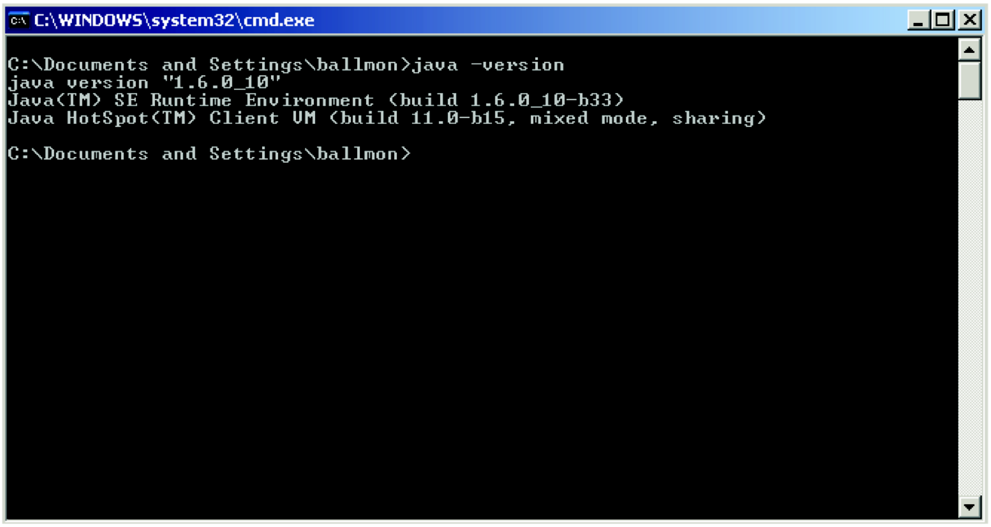
To pigeonhole Maven by calling it a build system doesn't do it justice. Apache Maven is a software project management and comprehension tool. What exactly does that mean? At the core of every Maven project is a project object model, more affectionately known as the POM, and from this POM Maven can build our application, generate reports, generate documentation, and more, all from a single description of the project. To learn more about Maven check out the Apache Maven project site at http://maven.apache.org or download the free ebook from Sonatype at http://www.sonatype.com/book.

Before moving ahead with Maven, be sure you have the Java Development Kit (JDK) version 1.5 or greater properly installed. You can follow the next section for that or skip it if you're ready to go. After you install the JDK, be sure to install the MySQL database as well. You'll need MySQL installed before generating the project with the AppFuse Maven archetype.

2.2.1 Download and install the JDK

To run any Java server-side environment, you must install and configure the JDK. Download and install JDK 1.5+ from the Sun website at http://java.sun.com/javase/downloads/index.jsp. Refer to the Java documentation for instructions on how to install Java on your specific platform. Set up an environment variable for JAVA_HOME that points to the JDK directory. It's also helpful to add the JDK's bin directory to the path. Open a command prompt and type in the Java version to verify that Java is installed correctly. The version information of the configured JDK should be presented as shown in figure 2.1.

After Java is configured you can move on to setting up the open source MySQL database.



```
C:\Documents and Settings\ballmon>java -version
java version "1.6.0_10"
Java(TM) SE Runtime Environment (build 1.6.0_10-b33)
Java HotSpot(TM) Client VM (build 11.0-b15, mixed mode, sharing)

C:\Documents and Settings\ballmon>
```

Figure 2.1 Verify that Java is set up correctly by checking the version

2.2.2 Download and install MySQL

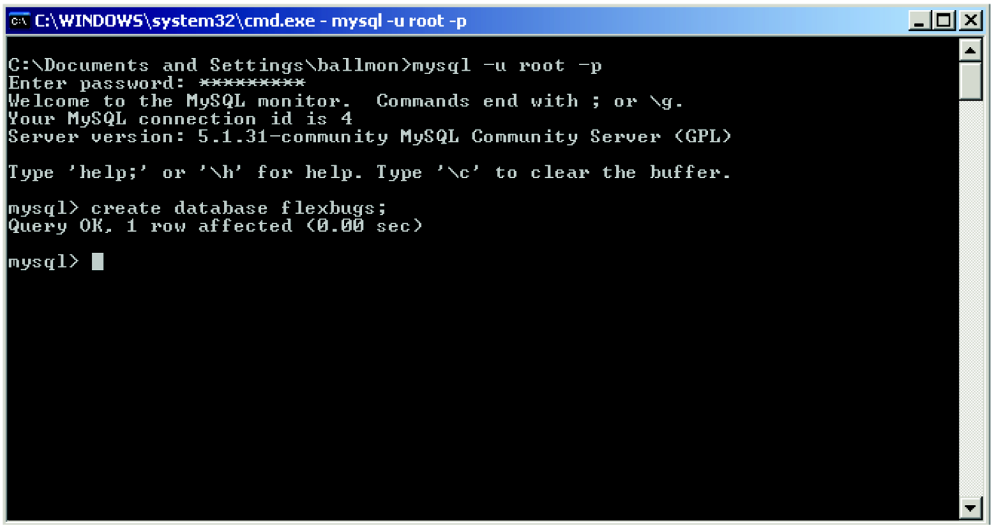
To demonstrate database integration and persistence you'll use MySQL, which is an open source database that is extremely lightweight. Download and install MySQL 5.x or higher from the MySQL website at http://dev.mysql.com/downloads/mysql.

Here you'll set up a database for the FlexBugs sample application. After you have MySQL installed pull up the command prompt and log in to MySQL using the root account, then create the flexbugs database as shown in figure 2.2. Using the command `mysql -u root -p` will instruct MySQL to log in to the local host instance of MySQL using the root account. It will ask for the password. Please record the admin account's user and password for later reference. Creating the database is as simple as executing the command `create database flexbugs`.

Let's move on to installing Maven to create the project structure, manage the dependencies, and build the application.

2.2.3 Download and install Maven

Maven can be downloaded at http://maven.apache.org/download.html. Be sure to download version 2.0.9 or above. After Maven is downloaded you should set up an M2_HOME environment variable that points to the directory where Maven was installed. The M2_HOME/bin directory will need to be set onto the path as well or exported for any UNIX platform. For more assistance on installing or configuring Maven refer to the Maven documentation at http://www.sonatype.com/books/mvnex-books/reference/installation-sect-maven-install.html.



```
C:\Documents and Settings\ballmon>mysql -u root -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.1.31-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database flexbugs;
Query OK, 1 row affected (0.00 sec)

mysql>
```

Figure 2.2 Using the MySQL commands to log into the database instance and create the flexbugs database

2.2.4 Create a Maven multimodule project

We're going to create a Maven multimodule project called FlexBugs. A multimodule project could be configured manually by creating a top-level *super POM*, adding projects under the super POM directory, and editing the super POM to include the modules with the modules element. We're going to use a technique that exploits a little known feature of the archetype:create plugin, and the Maven site archetype to kick-start the project.

Creating a multimodule project has many benefits, the two most important being (1) the ability to build every artifact in a project with a simple mvn compile command and (2) if you are using either the Maven eclipse:eclipse plugin or the idea:idea plugin, you can enter this command at the root of the project, and it will generate all the project files for all of the contained modules.

First you'll generate the top-level project using the `maven-archetype-site-simple` archetype:

```
mvn archetype:create
 -DgroupId=org.foj
 -DartifactId=flex-bugs
 -DarchetypeArtifactId=maven-archetype-site-simple
```

This generates a Maven project with the directory structure as shown in figure 2.3.

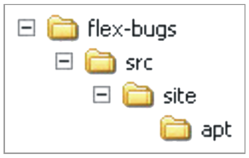


Figure 2.3 The generated top-level Maven project

The project generated is the minimum project setup necessary to generate site documentation. The index.apt file is the main index page for the site, and is written in the Almost Plain Text (APT) format, which is a wiki-like format. You can also generate a more complete site project using the `maven-archetype-site` archetype like this:

```
mvn archetype:create
 -DgroupId=[Java:the project's group id]
 -DartifactId=[Java:the project's artifact id]
 -DarchetypeArtifactId=maven-archetype-site
```

This will generate a project structure similar to figure 2.4.

After you have generated the site project, edit the pom.xml created from the site archetype plugin. Make sure that the packaging type is set to `pom`. We've left sections out (denoted by ...) to be brief.

Listing 2.1 Packaging of type `pom` indicates a multimodule project

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.foj</groupId>
    <artifactId>flex-bugs</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>                    ❶ Artifact type
     ...                                             (jar, war, ear)
</project>
```

Because you set the packaging type to `pom` ❶, any projects you generate from the root of the project directory will insert themselves into the project by creating an entry into the modules section of the pom.xml for the site.

AppFuse comes stocked with custom Maven archetypes, which allow AppFuse to create different flavors of Java web applications with varying technology stacks. You'll use the Struts 2 Basic archetype for the FlexBugs sample application.

In the root directory of your project that you created previously, type the command in listing 2.2.

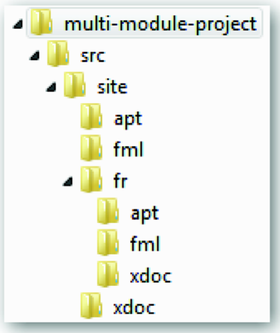


Figure 2.4 A fully dressed up Maven site project

Listing 2.2 Create the `flex-bugs-web` module for the Java server side

```
mvn archetype:create
-DarchetypeGroupId=org.appfuse.archetypes                    ❶
-DarchetypeArtifactId=appfuse-basic-struts                   ❷
-DremoteRepositories=http://static.appfuse.org/releases      ❸
-DarchetypeVersion=2.0.2                                     ❹
-DgroupId=org.foj.flex-bugs                                  ❺
-DartifactId=flex-bugs-web                                   ❻
```

The `appfuse-basic-struts` ❷ archetype isn't a built-in Maven resource. Instead, it's provided through a remote repository ❸. You provide Maven with coordinates to the archetype by also providing the `archetypeGroupId` ❶ and `archetypeVersion` ❹ along with the rest of the required details. The `groupId` ❺ points to the top-level project and the `artifactId` ❻ is the name of the module you are about to create.

After you've executed the command, look inside the top-level pom.xml from the main project. There should now be an entry toward the bottom of the file like the following.

```
...
<modules>
    <module>flex-bugs-web</module>
</modules>
...
```

Executing the command in listing 2.2 should generate the project structure shown in figure 2.5. Don't be concerned with the warnings while creating your project; they are expected. As long as you see BUILD SUCCESSFUL at the end, your project was created successfully.

As you can see from figure 2.5 Maven generated the project structure and added a couple of files for testing.

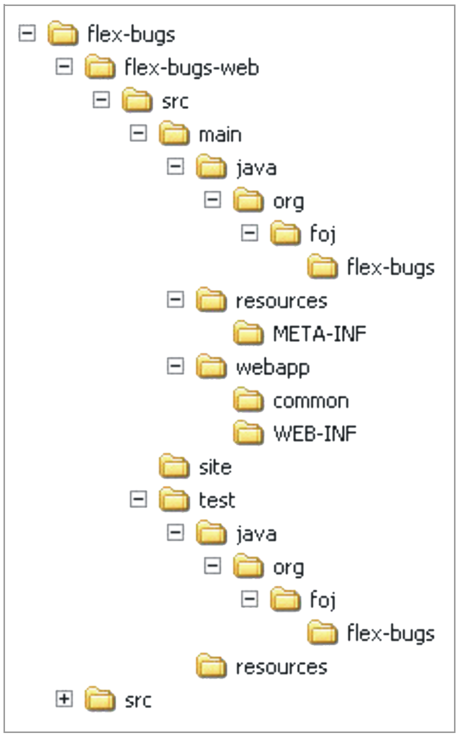


Figure 2.5
Generated module structure using the `appfuse-basic-struts` archetype

2.2.5 Maven provides a buildable project

If you look in the `src/main/java/org/foj` package you'll find a source file called App.java, and in the `src/test/java/org/foj` package you'll find a unit test called AppTest.java. Remove both files as you will not need them.

Notice that Maven appears to be building something. In fact, the `flex-bugs-web` POM tries to build a deployable Java Web Archive or *WAR* but will first choke on a configuration issue. If running the `mvn jetty:run-war` command without changing the configuration you'll most likely get this error.

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Error executing database operation: CLEAN_INSERT

Embedded error: Access denied for user 'root'@'localhost' (using password: NO)
[INFO] ------------------------------------------------------------------------
[INFO] For more information, run Maven with the -e switch
```

Let's first edit the POM for the `flex-bugs-web` module. This POM will be located at the root of that module. There's a good deal going but we're going to focus on the piece we need to change. At the bottom you need to specify your MySQL user and password with the values we specified when you set up MySQL earlier. Here's an example:

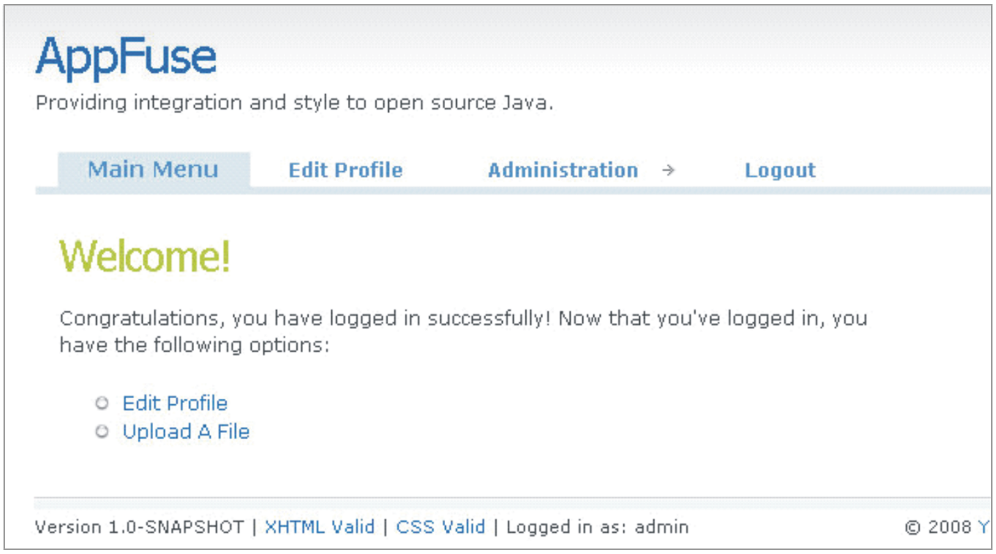
```
...
<jdbc.url><![CDATA[jdbc:mysql://localhost/
➥flex_bugs_web?createDatabaseIfNotExist=true&amp;useUnicode=true&amp;
➥characterEncoding=utf-8]]></jdbc.url>
<jdbc.username>root</jdbc.username>
<jdbc.password>java4ever</jdbc.password>
...
```

The Maven archetype we used, brought to us by AppFuse, made it extremely easy to get to this point—far easier than starting from scratch.

2.2.6 Running the FlexBugs web application

Maven equips a developer with the ability to use the application immediately without manually deploying it anywhere. Executing the Maven `jetty:run-war` goal from the `flex-bugs-web` module will gather all the resources, compile all the code and tests, execute the unit tests, generate test reports, build a deployable WAR file, and launch the WAR file in an embedded instance of the popular and lightweight Jetty servlet container. Using the `appfuse-basic-struts` archetype will also generate the default database for us and add configuration files to allow developers to quickly begin developing features.

After you've run the `jetty:run-war` command, you can go to http://localhost:8080/flex-bugs-1.0-SNAPSHOT and log in from there. By default, you can log in to the application using admin for both the username and password. After logging in, you are redirected to the administration panel as seen in figure 2.6. From there you can do basic things like editing your user profile and managing users.


AppFuse

Providing integration and style to open source Java.

Welcome!

Congratulations, you have logged in successfully! Now that you've logged in, you have the following options:

- Edit Profile
- Upload A File

Version 1.0-SNAPSHOT | XHTML Valid | CSS Valid | Logged in as: admin © 2008 Y

Figure 2.6 AppFuse default application

The application shows nothing glamorous at this point although everything you see and can do has required a minor setup effort. AppFuse does much under the covers for us from a framework and technology perspective. It's possible that getting a project together with help from Maven saved us a week or more of typical Java development time.

Before we start development of the FlexBugs sample application download the source code at https://flexonjava.googlecode.com/svn/flex-bugs/trunk.

2.3 Build the model objects

A model object is a POJO that is persistable and mapped to the database. In our example we're using AppFuse with the Spring framework and Hibernate to manage performing database operations for objects that are mapped to a database.

Let's start with `Issue.java` as seen in listing 2.3. For the FlexBugs application you need something to store issues and comments. An *issue* describes something that needs fixing to meet a requirement. This could be a bug, a new feature, a refactor, or an optimization. A single issue can have many comments so a relationship is built between the issue and comment objects.

Listing 2.3 The `Issue` model object

```
package org.foj.model;                                        <- 1

import org.apache.commons.lang.builder.EqualsBuilder;         <- 2
...

@Entity                                                       <- 3
public class Issue extends BaseObject implements Serializable {   <- 4

  private Long id;                                            <- 5
  private String project;
  private String description;
  private String type;
  private String severity;
  private String status;
  private String details;
  private String reportedBy;
  private Date reportedOn;
  private String assignedTo;
  private Double estimatedHours;

  @Id                                                         <- 6
  @GeneratedValue(strategy = GenerationType.AUTO)             <- 7
  public Long getId() {                                       <- 8
    return id;
  }

  public void setId(Long id) {                                <- 9
    this.id = id;
  }

    ...
```

1. Model Java package
2. Import declarations
3. Java persistence framework
4. Issue extends AppFuse BaseObject
5. Class instance variables
6. Declares database pk relationship
7. Indicates how to generate Id
8. "getter" method returns Id
9. "setter" method sets Id

```
  @Override
  public int hashCode() {                                          ⑩ hashCode
    return new HashCodeBuilder(11, 37).append(id).toHashCode();
  }

  @Override
  public boolean equals(Object o) {                      ⑪ equals
    if (null == o) return false;
    if (!(o instanceof Issue)) return false;
    if (this == o) return true;

    Issue input = (Issue) o;
    return new EqualsBuilder()
        .append(this.getId(), input.getId())
        .isEquals();

  }
                                                          ⑫ toString provides
  @Override                                                  object info
  public String toString() {
    return new ToStringBuilder(this, ToStringStyle.MULTI_LINE_STYLE)
        .append(id)
        .append(project)
        .append(description)
        .toString();
  }
}
```

You'll be storing the model objects in the `org.foj.model` Java package ❶ and will use the AppFuse framework in conjunction with the Spring Framework and Hibernate to simplify our application development. Spring provides DI and more while Hibernate is a database persistence framework that enables object relational mapping framework ❸. The `Id` ❻ and `GeneratedValue` ❼ annotation help to facilitate the persistence by designating a field as a database primary key.

The `Issue` object is a subclass of the AppFuse `BaseObject` ❹ and contains the instance variables ❺ you need to describe an issue. All of the instance variables or *fields* have the getters ❽ and setters ❾ required by the JavaBean specification.

NOTE Extending `BaseObject` requires us to override the `toString` ⓬, `equals` ⓫, and `hashCode` ❿ methods because they're defined as abstract in the `BaseObject` class. To implement these methods we're leveraging the Apache Commons Builder package ❷ for creating the elements for these methods. Whenever you're implementing the `Serializable` interface, it's a good idea to also implement the `equals` and `hashCode` methods and provide a `serialVersionUID` member.

Next you'll create a model object for a comment. The `Comment` will be another persistable object. There can be many comments to a single issue. For the remainder of the code snippets in this chapter we'll use "…" for trivial things like imports and getters and setters of similar objects.

Listing 2.4 The comment model object

```
...

@Entity
public class Comment extends BaseObject implements Serializable {

  private Long id;
  private Issue issue;
  private String author;
  private Date createdDate;
  private String commentText;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  @ManyToOne(fetch = FetchType.EAGER)
  public Issue getIssue() {
    return issue;
  }

  public void setIssue(Issue issue) {
    this.issue = issue;
  }

...

  @Override
  public int hashCode() {
    return new HashCodeBuilder(11, 37).append(id).toHashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (null == o) return false;
    if (!(o instanceof Issue)) return false;
    if (this == o) return true;

    Issue input = (Issue) o;
    return new EqualsBuilder()
        .append(this.getId(), input.getId())
        .isEquals();

  }

  @Override
  public String toString() {
    return new ToStringBuilder(this, ToStringStyle.MULTI_LINE_STYLE)
        .append(id)
        .toString();
  }

}
```

Comment declaration ❶

❷ **Comment has many-to-one relationship with Issue**

There's not much difference between an `Issue` and a `Comment` ❶. The class fields are related to comments and there is a many-to-one relationship ❷ with `Issue`. We've also told Hibernate that we'd like it to eagerly fetch the `Issue` when returning the `Comment`. In typical Java web development you would keep the session open to lazily load the `Issue` object only when it's referred to at runtime, but because your Flex application runs external to the JVM you cannot take advantage of this luxury.

Now that you have your model objects built you can create a set of DAOs. You'll need a DAO for issue and comment.

2.4 Build the DAOs

AppFuse provides generic implementations for DAOs that you can leverage if your DAOs do nothing more than the basic create, retrieve, update, delete (CRUD) operations. Because your `IssueDao` will do only basic operations, there is no need to define a concrete `IssueDao`. You can instead use the `GenericHibernateDao` which you'll see later when you wire the beans in the application context. The `CommentDao` needs to implement a couple of operations that go beyond the basic CRUD operations so you'll first create an interface for the `CommentDao`.

Listing 2.5 The CommentDao.java

```
...
public interface CommentDao extends GenericDao<Comment, Long> {

  List<Comment> getCommentsByIssueId(Long issueId);

  void deleteAllCommentsForIssueId(Long issueId);

}
```

The `CommentDao` has two simple methods, one that returns a list of `Comment` objects by passing in the `issueId` argument, and another to delete all of the `Comment` objects for an issue. The second method facilitates the deleting of `Issue` objects. Because `Comment` has a foreign key relationship with `Issue`, you cannot delete an `Issue` if any `Comments` refer to it.

NOTE We defined the relationship between `Comment` and `Issue` by annotating the field with a `@OneToOne` annotation, and could have also defined the reverse of that relationship in the `Issue` class by including a `Set` of `Comment` objects belonging to an `Issue`. Because we could not lazy load those objects, it would have forced us to eager load the `Comment` objects into the `Set`, which would force those `Comment` objects to eager load their `Issue` objects. This forces the `Issue` objects to eager load their `Comments`, and so on. This usually results in a stack overflow because you've effectively got a circular reference that causes an infinite loop of eager fetching.

Now let's implement the `CommentDaoImpl`.

Listing 2.6 The CommentDaoImpl.java

```
...
public class CommentDaoImpl extends GenericDaoHibernate<Comment, Long>
    implements CommentDao {

  public CommentDaoImpl() {
    super(Comment.class);
  }

  @Override
  @SuppressWarnings("unchecked")
  public List<Comment> getCommentsByIssueId(Long issueId) {
    return getHibernateTemplate().find
    ➥("from Comment where issue_id = ?", issueId);
  }
}
```

Much like the `IssueDaoImpl`, `CommentDaoImpl` extends `GenericDaoHibernate` but implements `CommentDao`. The only interesting thing happening here is that you have a method that returns a list of `Comment` objects by leveraging the Hibernate template and a query. Spring and Hibernate are a wonderful combination and make for clean and intuitive DAOs.

Now that you've constructed the DAOs you can build services.

2.5 Build the services

Now you need to expose services to the web tier. You'll be able to take advantage of these services for the Flex client you'll be building. Again you'll start with interfaces like `IssueManager` in listing 2.7.

Listing 2.7 The IssueManager.java

```
package org.foj.service;

import org.foj.model.Issue;
import javax.jws.WebService;
import java.util.List;

@WebService
public interface IssueManager {      ❶

  List<Issue> getAll();              ❷
  Issue get(Long id);                ❸
  Issue save(Issue issue);           ❹
  void remove(Long id);              ❺

}
```

❶ **IssueManager interface declaration with WebService annotation**

❷ **Return all issues**

❸ **Get specific issue by its ID**

❹ **Save issue**

❺ **Delete issue**

You define the `IssueManager` as a web service by annotating it using `@WebService` ❶. `IssueManager` contains methods defining your basic CRUD operations for reading ❷ and ❸, creating and updating ❹, and deleting ❺. Now let's take a look at the `CommentManager`.

Listing 2.8 The CommentManager.java

```
package org.foj.service;

import org.foj.model.Comment;
import javax.jws.WebService;
import java.util.List;

@WebService                                      (1) CommentManager interface declaration with WebService annotation
public interface CommentManager  {

  List<Comment> findCommentsByIssueId(Long issueId);      (2) Get comments for issue Id
  void deleteAllCommentsForIssueId(Long issueId);         (3) Delete all comments for issue
  Comment get(Long id);
  Comment save(Comment comment);                          (4) Save comment
  void remove(Long id);                                   (5) Remove comment

}
```

CommentManager is also a web service ❶ by virtue of it having the @WebService annotation just as with the IssueManager. It contains a method to return a list of Comment objects by providing an issueId ❷, a method for deleting all comments for an issue id ❸, a method for saving a comment ❹ and a method for deleting a comment ❺. Now let's provide implementation for the services like IssueManagerImpl.

Listing 2.9 The IssueManagerImpl.java

```
package org.foj.service.impl;

import org.AppFuse.dao.GenericDao;
import org.foj.model.Issue;
import org.foj.service.IssueManager;
import org.foj.service.CommentManager;
import java.util.List;
import javax.jws.WebService;

@WebService(serviceName = "IssueService",
    endpointInterface = "org.foj.service.IssueManager")   (1) IssueManagerImpl declaration with WebService annotation
public class IssueManagerImpl implements IssueManager {

  private GenericDao<Issue, Long> issueDao;
  private CommentManager commentManager;

  public IssueManagerImpl() {                              (2) Default no arg constructor
  }

  public IssueManagerImpl(GenericDao<Issue, Long> issueDao,
                          CommentManager commentManager) {   (3) Constructor
    this.issueDao = issueDao;
    this.commentManager = commentManager;
  }

  public List<Issue> getAll() {                           (4) Method that returns all Issues
    return issueDao.getAll();
  }
  public Issue get(Long id) {                          ❺ Get specific issue
    return issueDao.get(id);
  }

  public Issue save(Issue issue) {                     ❻ Save issue
    return issueDao.save(issue);
  }

  public void remove(Long id) {                        ❼ Remove an issue
    commentManager.deleteAllCommentsForIssueId(id);
    issueDao.remove(id);
  }

}
```

The `IssueManagerImpl` also uses the `@WebService` annotation just as in the interface, but provides the `serviceName` and `endpointInterface` attributes ❶. You provide a default no args constructor ❷ as well as one that will be used by Spring to inject the `IssueDao` and `CommentManager` ❸. Next implement the methods for returning the list of `Issue` objects ❹, returning a specific `Issue` ❺, and saving an `Issue` ❻ by delegating the calls to those methods to the `IssueDao`. The implementation for removing an issue ❼ first deletes any comments for the issue by calling the `CommentManager`, then removes the issue by calling the remove method on the `IssueDao`. Now let's look at the `CommentManager`.

Listing 2.10 The CommentManagerImpl.java

```
package org.foj.service.impl;

import org.foj.dao.CommentDao;
import org.foj.model.Comment;
import org.foj.service.CommentManager;
import java.util.List;
import javax.jws.WebService;

@WebService(serviceName = "CommentService",
            endpointInterface = "org.foj.service.CommentManager")
public class CommentManagerImpl implements CommentManager {
                                  ❶ CommentManagerImpl declaration with WebService annotation
  private CommentDao commentDao;

  public CommentManagerImpl() {                        ❷ Default no args constructor
  }

  public CommentManagerImpl(CommentDao commentDao) {   ❸ Constructor sets injected CommentDao instance to use
    this.commentDao = commentDao;
  }

  public List<Comment> findCommentsByIssueId(Long issueId) {   ❹ Find all comments for issue
    return commentDao.getCommentsByIssueId(issueId);
  }

  public void deleteAllCommentsForIssueId(Long issueId) {      ❺ Delete all comments for issue
    commentDao.deleteAllCommentsForIssueId(issueId);
  }
  public Comment get(Long id) {              ⊲ 6 Get specific comment
    return commentDao.get(id);
  }

  public Comment save(Comment comment) {     ⊲ 7 Save comment
    return commentDao.save(comment);
  }

  public void remove(Long id) {              ⊲ 8 Delete comment
    commentDao.remove(id);
  }

}
```

Like IssueManagerImpl, CommentManagerImpl declares itself to be a WebService ❶. Next using the @WebService annotation and defines its endpoint interface and service name you create a default no args constructor ❷ as well as one that will be used by Spring to inject your CommentDao ❸. You implement the methods to get the Comment objects for an issue ❹, deleting all the Comment objects for an issue ❺, getting a specific Comment ❻, saving a Comment ❼, and deleting a Comment ❽ by delegating to the CommentDAO.

NOTE AppFuse provides GenericManager implementation base classes just as it does for DAOs, but we chose not to use them here because certain WebService consumers such as Flex have difficulty dealing with web services that return objects such as ArrayOfAnyType, which is what AppFuse will return if we leverage the GenericManagers. To work around this issue you'll be defining and implementing your CRUD operations for the web services explicitly.

We're now officially done with the server-side objects and can wire things together with the Spring configuration and work on the web tier components.

2.6 Wiring things together with Spring

Spring enables developers to easily connect objects while keeping application components loosely coupled and testable. Notice how we've wired the model, DAO, and service objects together in the following listing. The applicationContext.xml is located in the src\main\webapp\WEB-INF directory, with other configuration files.

Listing 2.11 The applicationContext.xml

```
...
  <!-- Add new DAOs here -->
  <bean id="issueDao" class=                                    1 issueDao
➥"org.AppFuse.dao.hibernate.GenericDaoHibernate">
    <constructor-arg value="org.foj.model.Issue"/>
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <bean id="commentDao" class="org.foj.dao.impl.CommentDaoImpl">    ⊲ 2 commentDao
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
```

```
    <!-- Add new Managers here -->
    <bean id="issueManager" class="org.foj.service.impl.IssueManagerImpl">
      <constructor-arg ref="issueDao"/>
    </bean>

    <bean id="commentManager" class=
    ➥"org.foj.service.impl.CommentManagerImpl">
      <constructor-arg ref="commentDao"/>
    </bean>

...
```

3 issueManager

4 commentManager

The first bean you define is `GenericDao` for the `issueDao` **1**. The `commentDao` **2** is defined with your concrete implementation. Next, you create Spring beans for `issue-Manager` **3** and `commentManager` **4**. The `constructor-arg` element is used to inject the dependencies into the service class constructor.

Now that we've wired things up with Spring let's construct the web tier starting with Struts 2 framework action classes.

2.7 Constructing the web tier

Struts 2 applications implement the Model-View-Controller (MVC) design pattern, which is not to be confused with the MVP design pattern used to develop the Flex application. The pattern encourages separation between the data model, view elements, and controllers that sit between them. The MVC pattern, widely adopted in the Java community, has made its way into other languages and frameworks, like Flex.

2.7.1 Building Struts 2 action classes

You'll start by building controller or *action* classes first, like `IssueAction`.

Listing 2.12 The IssueAction.java

```
package org.foj.action;

import org.AppFuse.webapp.action.BaseAction;
...

public class IssueAction extends BaseAction {

  private IssueManager issueManager;
  private CommentManager commentManager;
  private List<Issue> issues;
  private List<Comment> comments;
  private Issue issue;
  private Long id;

  public void setIssueManager(IssueManager issueManager) {
    this.issueManager = issueManager;
  }

  public void setCommentManager(CommentManager commentManager) {
    this.commentManager = commentManager;
  }

  ...
```

1 IssueAction extends AppFuse BaseAction

2 Setters for IssueManager and CommentManager

```
  public String list() {                                   ❸ Returns list of
    issues = issueManager.getAll();                          Issue objects
    return SUCCESS;
  }

  public String delete() {                                 ❹ Deletes Issue
    issueManager.remove(issue.getId());
    saveMessage(getText("issue.deleted"));

    return SUCCESS;
  }

  public String edit() {                                   ❺ Edits by IssueId
    if (id != null) {
      issue = issueManager.get(id);
    } else {
      issue = new Issue();
    }

    comments = commentManager.findCommentsByIssueId(issue.getId());

    return SUCCESS;
  }

  public String save() throws Exception {                  ❻ Saves Issue
    if (cancel != null) {
      return CANCEL;
    }

    if (delete != null) {
      return delete();
    }

    boolean isNew = (issue.getId() == null);

    issue = issueManager.save(issue);

    String key = isNew ? "issue.added" : "issue.updated";
    saveMessage(getText(key));

    if (!isNew) {
      return INPUT;
    } else {
      return SUCCESS;
    }

  }

}
```

`IssueAction` extends the AppFuse `BaseAction` ❶ that contains many common methods that actions rely on. `IssueAction` has setters for the service objects ❷. These setters will be called by Spring, and their instances will be injected into the action class during runtime. `IssueAction` facilitates controlling communications to the server side from the web tier. It contains the methods for the view pages to list ❸, delete ❹, edit ❺ or, most importantly, save `Issue` objects ❻.

The `CommentAction` object serves the same purpose for the `Comment` object as the `IssueAction` object does for the `Issue` object. All the methods on `CommentAction` are facilitating CRUD for the `Comment` POJO by calling the `commentManager` service. The `CommentAction` class can be downloaded from the website if needed.

Now that the actions are in place, let's work on JSP files to create a simple UI for managing issues.

2.7.2 Editing the issue menu item

First you have to modify the menu.jsp to get to the issues list.

Listing 2.13 The menu.jsp

```
...
    <menu:displayMenu name="MainMenu"/>
    <menu:displayMenu name="UserMenu"/>
    <menu:displayMenu name="IssueMenu"/>
    <menu:displayMenu name="AdminMenu"/>
    <menu:displayMenu name="Logout"/>
...
```

1 Adding IssueMenu item to the JSP view file

The menu JSP file reads in the menu xml data. To add the `Issue` menu item you need only add a single line 1 to this file that is located in the flex-bugs-web/src/main/webapp/common directory. In the following listing you'll provide the xml data for that menu item.

Listing 2.14 The menu-config.xml

```
...
    <Menu name="IssueMenu" title="menu.issue"
    ➥description="Issues Menu"
        roles="ROLE_ADMIN,ROLE_USER" page="/issues.html">
      <Item name="ViewIssues" title="menu.viewIssues" page="/issues.html"/>
    </Menu>
...
```

1 Add Issue menu item to menu data xml file

By adding to the existing AppFuse plumbing that creates menu items 1, you quickly gain access to new features. Let's create the IssueList.jsp that will be displayed when you click the issues menu item.

2.7.3 Adding JSP resources

The issueList.jsp will display a list of issues and allow you to add or modify existing issues. The issue and comment JSP files will reside in the ../src/main/webapp/WEB-INF/pages directory.

Listing 2.15 The issueList.jsp

```
<%@ include file="/common/taglibs.jsp" %>

<head>
  <title><fmt:message key="issueList.title"/></title>
  <meta content="<fmt:message key='issueList.heading'/>" name="heading"/>
</head>
```

1 Essential tag libraries bundle

```
<c:set var="buttons">
  <input type="button" style="margin-right: 5px"
         onclick="location.href='<c:url value="editIssue.html"/>'"
         value="<fmt:message key="button.add"/>"/>
  <input type="button" onclick="location.href=
  ➥'<c:url value="/mainMenu.html"/>'"
         value="<fmt:message key="button.done"/>"/>
</c:set>
```

❷ Variable holds button data

```
<c:out value="${buttons}" escapeXml="false"/>
```

❸ Prints button data for display

```
<s:set name="issues" value="issues" scope="request"/>
```

❹ Variable represents issues list

```
<display:table name="issues" class="table" requestURI="" id="issueList"
    export="false" pagesize="25">
  <display:column property="id" sortable="true" href="editIssue.html"
                paramId="id" paramProperty="id" titleKey="issue.id"/>
  <display:column property="project" sortable="true"
     titleKey="issue.project"/>
  <display:column property="description" sortable="false"
     titleKey="issue.description"/>

  <display:setProperty name="paging.banner.item_name" value="issue"/>
  <display:setProperty name="paging.banner.items_name" value="issues"/>

</display:table>
```

Displays nicely formatted table ❺

```
<c:out value="${buttons}" escapeXml="false"/>

<script type="text/javascript">
  highlightTableRows("issueList");
</script>
```

❻ JavaScript highlights table rows

To make life easier, you include a JSP that in turn includes a bundle of tag libraries ❶ that are useful for the web application. You have button data that will be stored in a variable ❷ and a Java Standard Tag Library (JSTL) tag ❸ that will print the buttons. You create a variable that will hold a list of issues ❹ from the request scope and an HTML table that is formatted using the included display tag library ❺. A little JavaScript is used to highlight rows of data ❻. Now let's have a look at the issue-Form.jsp.

Listing 2.16 The issueForm.jsp

```
<%@ include file="/common/taglibs.jsp" %>

<head>
  <title><fmt:message key="issueDetail.title"/></title>
  <meta content="<fmt:message key='issueDetail.heading'/>"/>
</head>

<s:form id="issueForm" action="saveIssue" method="post" validate="true">
  <s:hidden name="issue.id" value="%{issue.id}"/>
```

"s" Struts 2 form tag in action ❶

```
  <s:textfield key="issue.project" required=
  ➥"true" cssClass="text medium"/>
  <s:textfield key="issue.description" required="true"
  ➥cssClass="text medium"/>
```

❷ Form text input fields

```
  <s:textfield key="issue.type" required="true" cssClass="text medium"/>
  <s:textfield key="issue.severity" required="true" cssClass="text medium"/>
  <s:textfield key="issue.status" required="true" cssClass="text medium"/>
  <s:textarea key="issue.details" required="true" cssClass="text medium"/>

  <li class="buttonBar bottom">                                   <┐
    <s:submit cssClass="button" method="save"                      ❸ CRUD Button bar
    ➥key="button.save" theme="simple"/>
    <c:if test="${not empty issue.id}">
      <s:submit cssClass="button" method="delete"
      ➥key="button.delete" onclick="return confirmDelete('issue')"
              theme="simple"/>
    </c:if>
    <s:submit cssClass="button" method="cancel"
    ➥key="button.cancel" theme="simple"/>
  </li>

</s:form>

<c:if test="${not empty issue.id}">

  <s:form id="commentsForm" action="editComment"               | Comments
  ➥method="post" validate="true">                              | Struts 2 form

    <s:set name="comments" value="comments" scope="request"/>
    <s:hidden name="issue.id" value="%{issue.id}"/>

    <display:table name="comments" class="table"
    ➥requestURI="" id="commentList" export="false" pagesize="25">
      <display:column property="id" sortable="true" href="editComment.html"
                      paramId="id" paramProperty="id" titleKey="comment.id"/>
      <display:column property="author"
      ➥sortable="true" titleKey="comment.author"/>
      <display:column property="commentText"
      ➥sortable="false" titleKey="comment.commentText"/>

      <display:setProperty name="paging.banner.item_name" value="comment"/>
      <display:setProperty name="paging.banner.items_name" value="comments"/>

    </display:table>

    <s:submit cssClass="button" key="button.add" theme="simple"/>

  </s:form>

</c:if>

<script type="text/javascript">
  highlightTableRows("commentList");
</script>                                                    | JavaScript
                                                             | assigns focus
<script type="text/javascript">                             <┘
  Form.focusFirstElement($("issueForm"));

</script>
```

Obviously, the issueForm.jsp will allow a user to add or edit an issue. If you peek into the included src/main/webapp/common/taglibs.jsp you'll notice that the Struts 2 tag libraries are included and the letter "s" was used for the tag prefix ❶. The Struts 2 `textfield` elements ❷ map to an `Issue` object. The button bar created will contain

Save, Delete, and Cancel buttons ❸. The Delete button will display only if the issue has an id or already exists. Let's keep moving and build the commentForm.jsp.

Listing 2.17 The commentForm.jsp

```
<%@ include file="/common/taglibs.jsp" %>

<head>
  <title><fmt:message key="commentDetail.title"/></title>
  <meta content="<fmt:message key='commentDetail.heading'/>"/>
</head>

<s:form id="commentForm" action="saveComment" method="post" validate="true">
  <s:hidden name="comment.id" value="%{comment.id}"/>
  <s:hidden name="issue.id" value="%{issue.id}"/>
  <s:textfield key="comment.author" required="true" cssClass="text medium"/>
  <s:textfield key="comment.createdDate" required="false"
  ➥cssClass="text medium"/>
  <s:textarea key="comment.commentText" required="false"
  ➥cssClass="text medium"/>

  <li class="buttonBar bottom">
    <s:submit cssClass="button" method="save"
    ➥key="button.save" theme="simple"/>
    <c:if test="${not empty comment.id}">
      <s:submit cssClass="button" method="delete"
      ➥key="button.delete" onclick="return confirmDelete('comment')"
              theme="simple"/>
    </c:if>
    <s:submit cssClass="button" method="cancel"
    ➥key="button.cancel" theme="simple"/>
  </li>

</s:form>
```

As the name suggests, the commentForm.jsp provides a Struts 2 form for updating new or existing comments. When submitted, the form will call the comment manager's `saveComment` method. Now that you have the JSP files in place we'll need to add those properties so that they have real values.

2.7.4 Adding property resources

For the application's messages to be localized, we've leveraged the Java resource bundle framework. Add the properties shown in the following listing to the ApplicationResources.properties file located in the flex-bugs-web/src/main/resources directory.

Listing 2.18 The ApplicationResources.properties

```
# -- menu/link messages --
menu.issue=Issues
menu.viewIssues=View Issues

# -- issue form --
issue.id=Id
issue.project=Project
issue.description=Description
issue.added=Issue has been added successfully.
issue.updated=Issue has been updated successfully.
issue.deleted=Issue has been deleted successfully.

# -- issue list page --
issueList.title=Issue List
issueList.heading=Issues

# -- issue detail page --
issueDetail.title=Issue Detail
issueDetail.heading=Issue Information

# -- comment form --
comment.id=Id
comment.author=Author
comment.issueId=Issue Id
comment.createdDate=Created Date
comment.commentText=Details
comment.added=Comment has been added successfully.
comment.updated=Comment has been updated successfully.
comment.deleted=Comment has been deleted successfully.

# -- issue list page --
commentList.title=Comment List
commentList.heading=Comments

# -- issue detail page --
commentDetail.title=Comment Detail
commentDetail.heading=Comment Information
```

If more language support is needed, add the same properties with the respective translation to the appropriate properties file in the same directory. Now let's wire up the view components with Struts 2.

2.7.5 Configuring the struts.xml

To wire up the JSP view components to the controller objects, you can use the struts.xml located in the src/main/resources directory. This listing demonstrates the wiring you need for the issues management.

Listing 2.19 The struts.xml

```
<package>
    ...
 <!-- Add additional actions here -->
    <action name="issues"
    ➥class="org.foj.action.IssueAction" method="list">           ❶
      <result>/WEB-INF/pages/issueList.jsp</result>
    </action>

    <action name="editIssue"
    ➥class="org.foj.action.IssueAction" method="edit">           ❷
      <result>/WEB-INF/pages/issueForm.jsp</result>
```

❶ **issues action loads issueList.jsp**

❷ **editIssue loads issueForm.jsp**

```
    <result name="error">/WEB-INF/pages/issueList.jsp</result>
  </action>

  <action name="saveIssue"
  ➥class="org.foj.action.IssueAction" method="save">          ❸ saveIssue loads
    <result name="input">/WEB-INF/pages/issueForm.jsp</result>        issueForm
    <result name="cancel" type="redirect-action">issues</result>
    <result name="delete" type="redirect-action">issues</result>
    <result name="success" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
  </action>

  <action name="comments" class="org.foj.action.CommentAction"
  ➥method="list">
    <result>/WEB-INF/pages/commentList.jsp</result>
  </action>

  <action name="editComment" class="org.foj.action.CommentAction"
  ➥method="edit">
    <result>/WEB-INF/pages/commentForm.jsp</result>
    <result name="error">/WEB-INF/pages/commentList.jsp</result>
  </action>

  <action name="saveComment" class="org.foj.action.CommentAction"
  ➥method="save">
    <result name="input">/WEB-INF/pages/commentForm.jsp</result>
    <result name="cancel" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
    <result name="delete" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
    <result name="success" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
  </action>

</package>
```

Struts 2 makes it simple to wire up the view components quickly and make changes. As you can see, the `issues` action ❶ will load the `issueList.jsp` whenever the `list()` method is invoked. In the same way, `editIssue` ❷ will load the `issueForm.jsp` when the `edit()` method is called and if that doesn't work, it will go back to the list page. The `saveIssue` action ❸ will persist an issue by taking the input from the `issueForm.jsp`.

The remainder of the `IssueAction` is more of the same but pertains to issue comments.

2.7.6 Configuring Hibernate

The final step is to configure the POJOs with the Hibernate session factory. That way when the app is loaded into memory, Hibernate recognizes these objects. You do this through the hibernate.cfg.xml located in the src/main/resources directory.

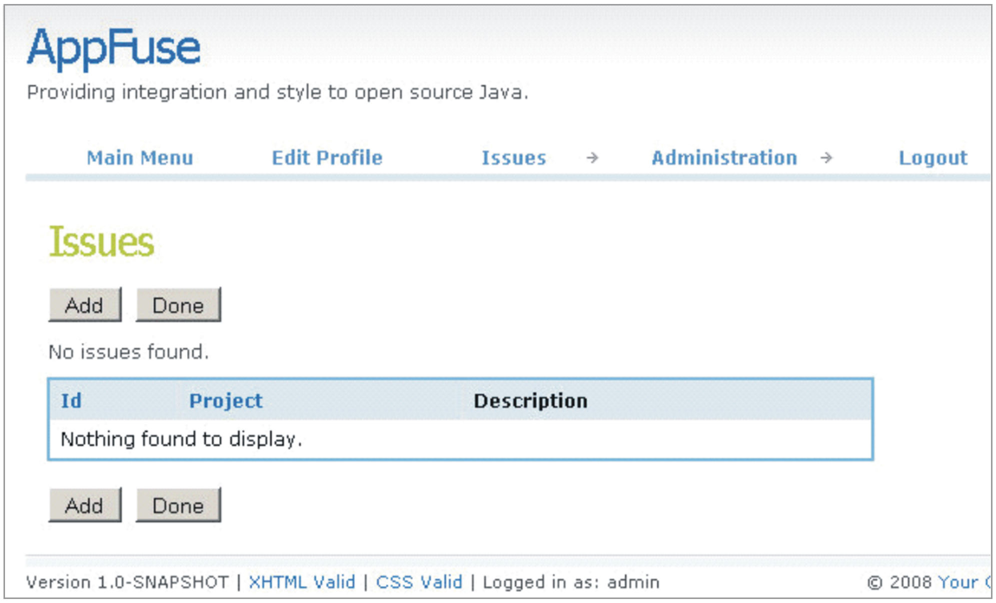
Listing 2.20 The hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
➥Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <mapping class="org.AppFuse.model.User"/>
    <mapping class="org.AppFuse.model.Role"/>
    <mapping class="org.foj.model.Issue"/>
    <mapping class="org.foj.model.Comment"/>
  </session-factory>
</hibernate-configuration>
```

❶ **Adding issue and comment**

In this simple configuration, you create a class mapping ❶ for each of the model objects, `Issue` and `Comment`. Rebuild the application with the `mvn jetty:run-war` command, then refresh your browser. The Issues button should be available as seen in figure 2.7.


AppFuse
Providing integration and style to open source Java.
Main Menu
Edit Profile
Issues
Administration
Logout
Issues
Add
Done
No issues found.

Id	Project	Description
Nothing found to display.		

Add
Done
Version 1.0-SNAPSHOT | XHTML Valid | CSS Valid | Logged in as: admin
© 2008 Your

Figure 2.7 The issues list page with the integrated Issues menu button

2.8 Summary

In this chapter we set up a Java web application using the AppFuse framework. AppFuse simplified the plumbing involved in building a typical Java web application by using many popular frameworks, for example Struts 2, Spring, and Hibernate.

In the next chapter we'll start building the rich interface for the sample application in Flex. In the following chapters we'll begin to connect the Flex front end to the Java application using web services and BlazeDS.

WEB/JAVA

FLEX ON JAVA

Bernerd Allmon • Jeremy Anderson

Together, Flex and Java make a powerful web development platform—they blend the strengths of Java on the server with the richness of Flex on the frontend. *Flex on Java* is a unique book that teaches you how to work with Flex in concert with the full array of Java technologies: Spring, POJOs, JMS, and other standard tools. You also learn how to integrate Flex with server-side Java via BlazeDS remoting. Almost all the carefully annotated examples use free or open source software.

What's Inside

- Build rich Flex 4 clients over Java backend systems
- Detailed examples using standard Java components
- Unit testing, charting, personalization, and other real-world techniques

This book is written for Java developers—no prior Flex experience is assumed.

BJ Allmon is a software craftsman, a polyglot developer, and team coach for Pillar Technology Group, an agile business and technology consulting firm. **Jeremy Anderson** is a Java developer and agile consultant with a passion for Groovy, Grails, and... Flex.

For online access to the authors and a free ebook for owners of this book, go to manning.com/FlexonJava

Free ebook
SEE INSERT

"Teaches a holistic approach to building great software."
—From the Foreword by James Ward, Flex Evangelist at Adobe

"Go from novice to expert with just this one book!"
—Peter Pavlovich
Kronos Incorporated

"A veritable tour de force."
—John S. Griffin, coauthor of *Hibernate Search in Action*

"Leverages your existing Java knowledge... a must-read.
—Brian Curnow
Gordon Food Service.

"Fantastic... extremely focused... packed with practical examples."
—Doug Warren
Java Web Services

ISBN 13: 978-1-933988-79-5
ISBN 10: 1-933988-79-7
53999
9 781933 988795

MANNING $39.99 / Can $45.99 [INCLUDING eBOOK]