LAB EXERCISES

EXERCISES

TUTORIAL : SETUP ECLIPSE/WTP FOR SPRING TRAINING

Overview	
setup the Derby database	During this exercise we will prepare the Eclipse Spring and the various technologies that we wi
Objective	1. To illustrate the one-time setup that is helpful for developing Spring applications within Eclipse 2. To understand how to work with Derby
Builds on Previous Labs	None
Time to Complete	20-25 minutes

Task List

Overview of Steps

1. **Familiarize yourself with the assignment**
2. **Locate and examine the Spring Installation**
3. **Setup User Libraries for use with Eclipse Projects**
4. **Start up the Derby database server**
5. **Create a user library to access Derby databases from a stand-alone Java application**
6. **Create the database required for the course**
7. **Examine the setup of the labs and solutions**

Step 1: Familiarize yourself with the assignment

During this exercise you will gain experience setting up the Eclipse Web Tools Project (WTP) environment to effectively perform Spring development. We will explore the steps necessary to incorporate the Spring JAR files into an application.

During the second half of the exercise, we will run a Derby server and perform various database operations.

Note: All of the exercises included with this course use Derby as an example database. Derby was chosen as the database for several reasons:

- The SQL dialect supported by Derby is identical to the SQL dialect supported by IBM's DB2.
- It is freely available at: http://db.apache.org/derby .
- Derby has a very small memory and disk footprint, making it an excellent choice within the lab environment.

Step 2: Locate and examine the Spring Installation

Locate the folder where Spring was unzipped onto your local hard drive. We will refer to the directory as `${SPRING_ROOT}` in the rest of this document.

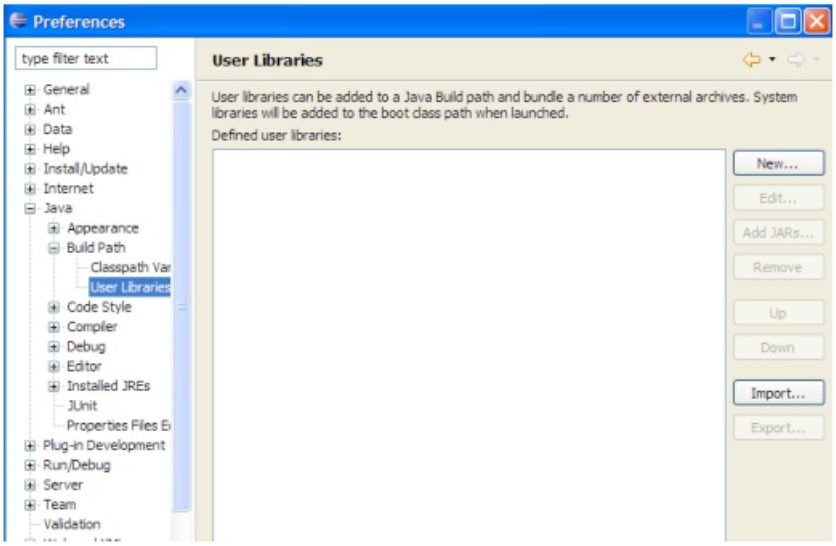
Take a few minutes to examine the contents of this folder and sub-folders. In particular, examine `${SPRING_ROOT}/dist;` this folder contains JAR files for the various Spring modules. In this course you will use many of these JAR files for both compiling and executing code.

This course focuses on Spring 3.0 as well as several other technologies that can be used in conjunction with Spring. In previous versions of Spring, these companion technologies were included in the Spring distribution. Spring has built a dependencies zip file that has most of the jar files that the components of Spring could depend on. That was the source of the jar files that we will be using for the dependencies in this class.

Step 3: Setup User Libraries for use with Eclipse Projects

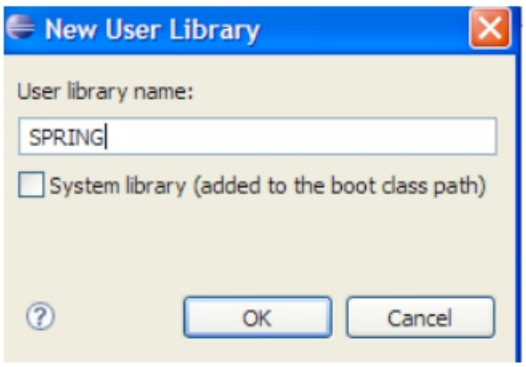
In this step you will define a *user library* in the Eclipse environment to provide your Spring applications access the commonly used classes in the various Spring JAR files.

On the *Window* | *Preferences* dialog box, navigate to the *Java* | *Build Path* | *User Libraries* section.



Click the *New* button.

Enter `SPRING` for the *User library name* and click *OK* .



With the *SPRING* user library selected in the *Preferences* dialog box, click the *Add JARs…* button.

The first set of JAR files that we are going to add to the library are those that come with the Spring distribution. We have included them in the lab code at:

~ / StudentW ork/ Tools/ Spring_ 3_ Base/ Distribution

Just for information purposes, the necessary Core jars are listed below, although the specific version numbers may be different.

Put them in the Spring User Library.

```
org.springframework.aop-3.0.2.RELEASE.jar
org.springframework.asm-3.0.2.RELEASE.jar
org.springframework.aspects-3.0.2.RELEASE.jar
org.springframework.beans-3.0.2.RELEASE.jar
org.springframework.context.support-3.0.2.RELEASE.jar
org.springframework.context-3.0.2.RELEASE.jar
org.springframework.core-3.0.2.RELEASE.jar
org.springframework.expression-3.0.2.RELEASE.jar
org.springframework.instrument.tomcat-3.0.2.RELEASE.jar
org.springframework.instrument-3.0.2.RELEASE.jar
org.springframework.jdbc-3.0.2.RELEASE.jar
org.springframework.jms-3.0.2.RELEASE.jar
org.springframework.orm-3.0.2.RELEASE.jar
org.springframework.oxm-3.0.2.RELEASE.jar
org.springframework.test-3.0.2.RELEASE.jar
org.springframework.transaction-3.0.2.RELEASE.jar
org.springframework.web.portlet-3.0.2.RELEASE.jar
org.springframework.web.servlet-3.0.2.RELEASE.jar
org.springframework.web.struts-3.0.2.RELEASE.jar
org.springframework.web-3.0.2.RELEASE.jar
```

In the next step, we add a second set of JAR files that are commonly used dependencies. These are normally included in the Spring Dependencies download and are either supporting technologies or technologies that are used in conjunction with Spring. Using the steps outlined above, add several JAR files to the SPRING user library. Navigate to the **~ / StudentW ork/ Tools/ Spring_ 3_ Base/ Dependencies** folder and work through adding each of these JAR files (again, the versions may be slightly different depending on the associated Spring 3 distribution):

```
com.springsource.org.apache.log4j-1.2.15.jar
com.springsource.org.apache.commons.beanutils-1.8.0.jar
com.springsource.org.apache.commons.collections-3.2.1.jar
com.springsource.org.apache.commons.lang-2.1.0.jar
com.springsource.org.apache.commons.logging-1.1.1.jar
com.springsource.org.apache.commons.pool-1.5.3.jar
com.springsource.org.quartz-1.6.2.jar
com.springsource.com.caucho-3.2.1.jar
com.springsource.org.apache.taglibs.standard-1.1.2.jar
com.springsource.javax.servlet.jsp.jstl-1.1.2.jar
```

Step 4: Start up the Derby Server

Navigate to the **~ / StudentW ork/ Tools/ Database** folder and double click on the **startup.bat** file. This will open an DOS window and start up the Derby server. Leave the resulting DOS window open to ensure that the server continues to run. Closing the DOS window will shut the server down in a non-orderly fashion.

- To shut the server down in an orderly fashion, double-click on the **shutdow n.bat** file. This sends a shutdown command to the server and the database, allowing it to shutdown in a safe fashion.

Step 5: Create a user library to access Derby databases from a stand-alone Java application

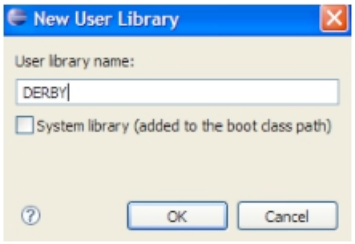
The purpose of this step is to ensure that the Derby drivers are available in the runtime environment. If you are deploying your project into JBoss, you do not need to perform this step, as JBoss includes the Derby drivers in its runtime environment. Skip this step if you are using JBoss.

Start Eclipse.

On the *Window* | *Preferences* dialog box, navigate to the *Java* | *Build Path* | *User Libraries* section.

Click the *New* button.

Enter `DERBY` for the *User library name* and click *OK* .



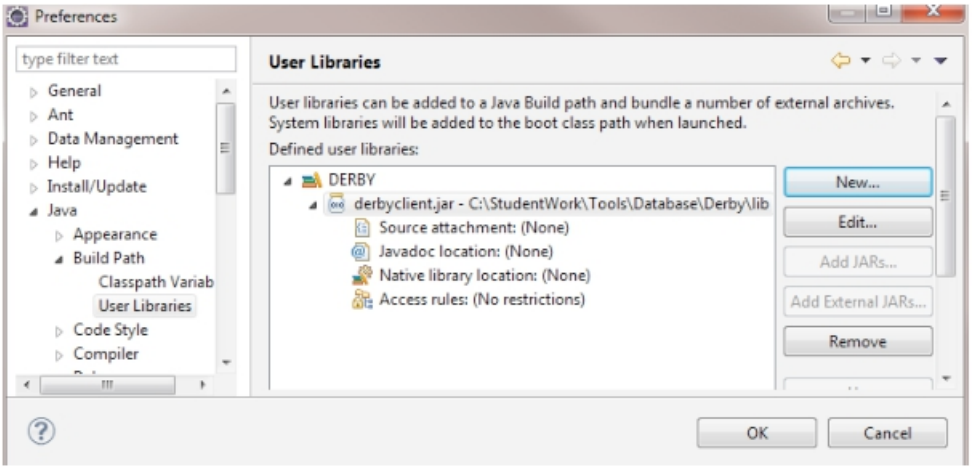
With the *DERBY* user library selected in the *Preferences* dialog box, click the *Add JARs…* button.

Navigate to the

~ / StudentW ork/ Tools/ Database/ Derby/ lib directory and select the `derbyclient.jar` files.

Click the *Open* button.

Click the *Open* button. The *Preferences* dialog box should look similar to the following:



Click the *OK* button to dismiss the *Preferences* dialog box.

Step 6: Setup a Eclipse Project to Setup, Test, and Reset the Database

If you have not done so already, startup the Derby server.

Open a DOS window and change directories to the **~ / StudentW ork/ Tools/ Database** folder.

Enter the **SpringBaseCreateDB** command in the DOS window. This command will cause a Derby utility to run the **SpringBaseCreateSQL.sql** script, which will, in turn, connect to the running Derby server and execute the SQL commands in the script.

If the server is not running, the script will fail since it will not be able to make a connection. If this is the first time that you have run the command, the database will be created in the **~ / StudentW ork/ Tools/ Database** folder. You will see an error in the script as the SQL attempts to drop a table that does not yet exist. After the initial dropping of the table, the remaining SQL creates the database.

You can use the **SpringBaseCreateDB** command to completely reset the database that we will be working against throughout this course. If you open the associated SQL script, you will see the SQL commands that are used to construct the tables that will be used later in the course.

At this time, run the **SpringBasePopulateDB** command and run the SQL scripts within it. This will take some time as it populates the database with a fair amount of data. Be patient and it will eventually complete and return.

There are two testing SQL files as well – **SpringBaseTestDVDTitlesDB** and **SpringBaseTestLoanDB** . These are fairly simple ones that you can use to perform basic checks on what is going on with the database. Feel free to examine and run them at any time.

If the server is running, the Derby utility will connect to it and execute the query commands against the database. You should see the results of those queries in the DOS window.

Note: You will be using the Derby database in one or more of the exercises in this course. In each case, the Derby library that you created will be used to facilitate operations.

Note that the Derby database server must be up and running before you can perform any database operations, so remember to restart it if you do shut it down.

Step 7: Examine the setup of the labs and solutions

Navigate to the ~ /StudentWork folder. You will see several folders there (some of which you have already worked with). The two folders of primary focus during the labs are the Labs folder, which contains the starting point for each lab and the Solutions folder, which contains the solution for each lab.

Expanding the Lab folder will reveal a separate folder for each lab. Each lab is standalone (although many of them do operate against the same database that we just set up). All of the labs are centered on a common data model and set of classes.

Generally speaking the labs are either Java applications or JEE applications. All of the labs have source code and various configuration files. One of the more challenging aspects of these types of development efforts is keeping track of all of the configuration files and where they go for the type of deployment that the application will eventually be run through. Part of the learning process associated with these labs is learning what the configuration files and where they need to go. The same is true with JAR files associated with Spring, Hibernate, JSF, Struts, etc.

Feel free to take a minute to look at a couple of representative labs. Spring-HelloWorld is the first of the Java applications that we will be working with.

Exercise 1. CREATE A SIMPLE "HELLO WORLD" SPRING PROGRAM

Overview	
	In this exercise you will write a basic Spring pr
output.	
The objectives of this exercise are:	
1.	Learn how to configure and run a Spring progra
2.	Become confident that you understand how "D
3.	Gain familiarity with the basics of the Spring X

Objective	Familiarity with creating and deploying Spring programs.
Builds on Previous Labs	Standalone. However, all the code for this exercise is directly covered in the course.
Time to Complete	30 minutes

Task List

Overview of Steps

1. **Start Eclipse and create a HelloWorld Project**
2. **Create the `Messager` Interface and its two subclasses**
3. **Create the `MessageLooper` class**
4. **Create the `HelloWorldMain` class**
5. **Create the Spring XML configuration file `spring-config.xml`**
6. **Run the program and inspect the output**
7. **Configure Log4j to get rid of the logging output**
8. **Add Annotation support and a new bean implementation**
9. **Run the program and inspect the output**

Step 1: Start Eclipse and create a HelloWorld Project

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Setup a new Java project in Eclipse called **HelloW orld** . You can perform this operation from a variety of starting points (right-click in the Project Explorer and select **New**).

Once you have finished creating the project, we need to set up the project so the compiler can find the Spring classes during the build process. Right-click on the **HelloW orld** project's entry and select **Properties** . A dialog box will appear. Select **J ava Build Path** in the left pane. Select the **Libraries** tab and then **Add Library -> User Library -> Next -> SPRI NG -> Finish -> OK** . This brings the Spring library classes and the companion technologies into play during editing and compilation. Each of the subsequent labs will require some set of libraries and JAR files to be added.

The next step is to create the com.springclass package. Right-click on the **HelloW orld** project's entry and select **New -> Package** . Type in **com.springclass** as the Name and click **Finish.**

Step 2: Create the `Messager` Interface and its two subclasses.

Right click on the **com.springclass** package and select **New -> I nterface** . Name it `Messager` . Click **Finish** .

Add the method `greet()` . (Hint: **public void greet( ) ;**) Save your work.

We're now going to create two classes which *implement* our new Messenger interface.

Right click on the **com.springclass** package and select **New -> Class** . Add the interface **`com.springclass.Messager`** as an interface and name the new class `HelloMessager` . Click **Finish.**

Modify the auto-generated `greet()` method to use **`System.out.println()`** to write out "Hello World". Save your work.

Right click on the **com.springclass** package and select **New -> Class** . Add the interface **`com.springclass.Messager`** as an interface and name the new class `GoodbyeMessager` . Click **Finish.**

Modify the auto-generated `greet()` method to use **`System.out.println()`** to write out an appropriate sign off message. Save your work.

Step 3: Create the `MessageLooper` class

Right click on the **com.springclass** package and select **New -> Class** . Name the new class `MessageLooper` . Click **Finish.**

Add two variables: one for a variable of type `Messager` called `messager` ; and the other of type `int` titled `numTimes` .

Make sure that each variable is a private variable and has a setter and getter. Hint: After creating the private variable, you can right click on the private variable and select **Source -> Generate Getters and Setters -> Select All -> OK.**

Add an additional method called `doIt` that loops `numtimes` invoking the `greet` method of the `messager`

Make sure to save your work.

Step 4: Create the `HelloWorldMain` class

Right click on the **com.springclass** package and select **New -> Class** . Name the new class `HelloWorldMain` . Click **Finish.**

Add the implementation of the `main()` method. The code to add is shown below:

```
public static void main(String[] args) {
  String springConfig = "com.springclass/spring-config.xml";
  ApplicationContext spring =
    new ClassPathXmlApplicationContext(springConfig);
  MessageLooper messageLooper =
   (MessageLooper)spring.getBean("messageLooper");
  messageLooper.doIt();
}
```

You will notice that you must create import statements for the various Spring classes. Eclipse makes this step easy – simply hover the cursor of each class that has a red line underneath and pause; a pop-up will be displayed which includes the option of creating the import statement. Save your work.

? Step 5: Create the Spring XML configuration file `spring-config.xml`

Right click on the **com.springclass** package and select **New -> File** . Name the new file spring-config.xml. Click **Finish.**

Click on the **Source** tab. Add the contents of the file as given in the course notes. If you would rather not type in the namespaces and Schema locations, you can copy and paste them from

~ \ StudentW ork\ Labs\ Spring-HelloW orld\ src\ com\ springclass\ readme.txt

Save your work.

Eclipse will attempt to validate the file by accessing the schema at its remote location as given in the `xsi:schemaLocation` attribute of the `bean`'s element. This will work, if the machine has Internet access.

Make sure to include the following bean definitions to the < beans> element of the file.

```
<bean id="messageLooper" class="com.springclass.MessageLooper">
 <property name="messager" ref="messager" />
 <property name="numTimes" value="5" />
</bean>

<bean id="messager" class="com.springclass.GoodbyeMessager" />
```

? Step 6: Configure Log4j

In our exercises we will have Spring use Log4J to produce logging information. Log4J is configured with the **log4j.properties** file.

Right click on the **src** icon and select **New -> File** . Name the new file log4j.properies and add the following contents:

```
# Set root logger level to ERROR and add an appender called A1.
```

```
log4j.rootLogger=ERROR, A1

# set A1 to be the console
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# Use the PatternLayout for A1
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-5p %c - %m%n
```

When you run the program in the next step you will see logging output. You can control the level of information displayed by changing the word `ERROR` to `INFO` or other valid logging level.

This step will not be repeated in later labs. You may optionally use the **log4j.properties** file in any of the other lab exercises.

Step 7: Run the program and inspect the output

Run the **HelloW orldMain** program. You can execute **HelloW orldMain** by right-clicking on it and selecting **Run As -> J ava Application.**

In the Eclipse console you may see some output of the logging system, followed by five occurrences of **Hello W orld!**

Optionally, change the XML config file to use the `GoodbyeMessager` or to change the number of times that the message is displayed. Run the changed application.

Step 8: Add Annotation support and a new bean implementation

Right click on the **com.springclass** package and select **New -> Class** . Add the interface **`com.springclass.Messager`** as an interface and name the new class Annotated `Messager` . Click **Finish.**

Modify the auto-generated `greet()` method to use **`System.out.println()`** to write out "Hello Annotated Class". Save your work.

Make sure to include the following element to the < beans> element of the spring-config file.

```
<?xml version= "1.0" encoding= "UTF-8" ?>
<beans xmlns= "http://www.springframework.org/schema/beans"
xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
xmlns:context= "http://www.springframework.org/schema/context"
 xsi:schemaLocation= "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd ">

<context:component-scan base-package= "com.springclass" />
<context:annotation-config/>
</beans>
```

Modify the AnnotatedMessenger to register it as a Spring Bean. Add the following annotation to before the class declaration:

```
@Component("annotatedmessager")
```

Modify the spring-config.xml to use the annotatedmessager to the messager property of the messagelooper

Step 9: Run the program and inspect the output

Run the **HelloW orldMain** program. You can execute **HelloW orldMain** by right-clicking on it and selecting **Run As -> J ava Application.**

In the Eclipse console you may see some output of the logging system, followed by five occurrences of **Hello Annotated Class!**

Optionally, change the XML config file to use the `GoodbyeMessager` or to change the number of times that the message is displayed. Run the changed application.

Exercise 2. CONFIGURING DEPENDENCIES

Overview	
	In this exercise you will modify the Spring conf between multiple classes, which will allow your The exercise focuses on the configuration file;
Objective	Become familiar with the Spring configuration file.
Builds on Previous Labs	Standalone
Time to Complete	30 minutes

Task List

Overview of Steps

1. **Inspect the pre-built classes**
2. **Take a closer look at `config.bo.KioskService`**
3. **Inspect the *`ApplicationContext.xml`***
4. **Add the DAO classes to the Configuration file**
5. **Complete the KioskService Definition**
6. **Validate `ApplicationContext.xml`**
7. **Run the TestClient**
8. **Inspect the output**

? Step 1: Create a Java project in Eclipse and inspect the pre-built classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

In this exercise you will configure the Spring XML configuration file.

No Java coding is required during this exercise; this is an excellent time to become familiar with the provided classes, as they will be used throughout the remainder of the class.

Create a new Java project in Eclipse called **Configuration** . You can do this operation in a number of ways (e.g., right-click in the Project Explorer and select **New**). Next, configure it so that the compiler can find the Spring classes during the build process. Right-click on the project's entry and select **Properties** . A dialog box will appear. Select **J ava Build Path** and then select the **Libraries** tab and then **Add Library -> User Library -> Next -> SPRI NG -> Finish -> OK** .

Next import the various resources required for this lab. The import operation is one of the most frustrating aspects of working with any Eclipse-based IDE — it simply takes

time to get used to its nuances. The frustration is further magnified by the fact that we have various files and resources that must be imported into the correct location in a project's structure for Eclipse to properly manage them. Luckily, things are a little easier with Java applications, which is what we are working with in this lab.

Right click on the **src** folder of the project's root node in the **Project Explorer** and select **I mport** , expand **General** , and then select **File System** . You are presented with a dialog box with two important parts — one to select where and what to import and the other being the folder to import into.

Use the top **Brow se** button to navigate to the **~ / StudentW ork/ Labs/ SpringConfiguration/ src** folder. This will set you up to import the Java source code, which is in the **src** folder. It is critical to import from that base folder because that ensures that the folder structure corresponding to the Java packages are also imported into the Java project. Select the check box beside the **src** folder. If you expand the **src** folder, you will eventually see the Java classes that are being imported.

For the purposes of Java projects, you can simply import these Java files into the base project **src** folder. If you set up the build path correctly and imported files correctly, you should not see any errors generated from the automatic compilation that Eclipse performs at the completion of the import process. If you see errors, try to diagnose the problem and/or ask for help. You will not be the only one…

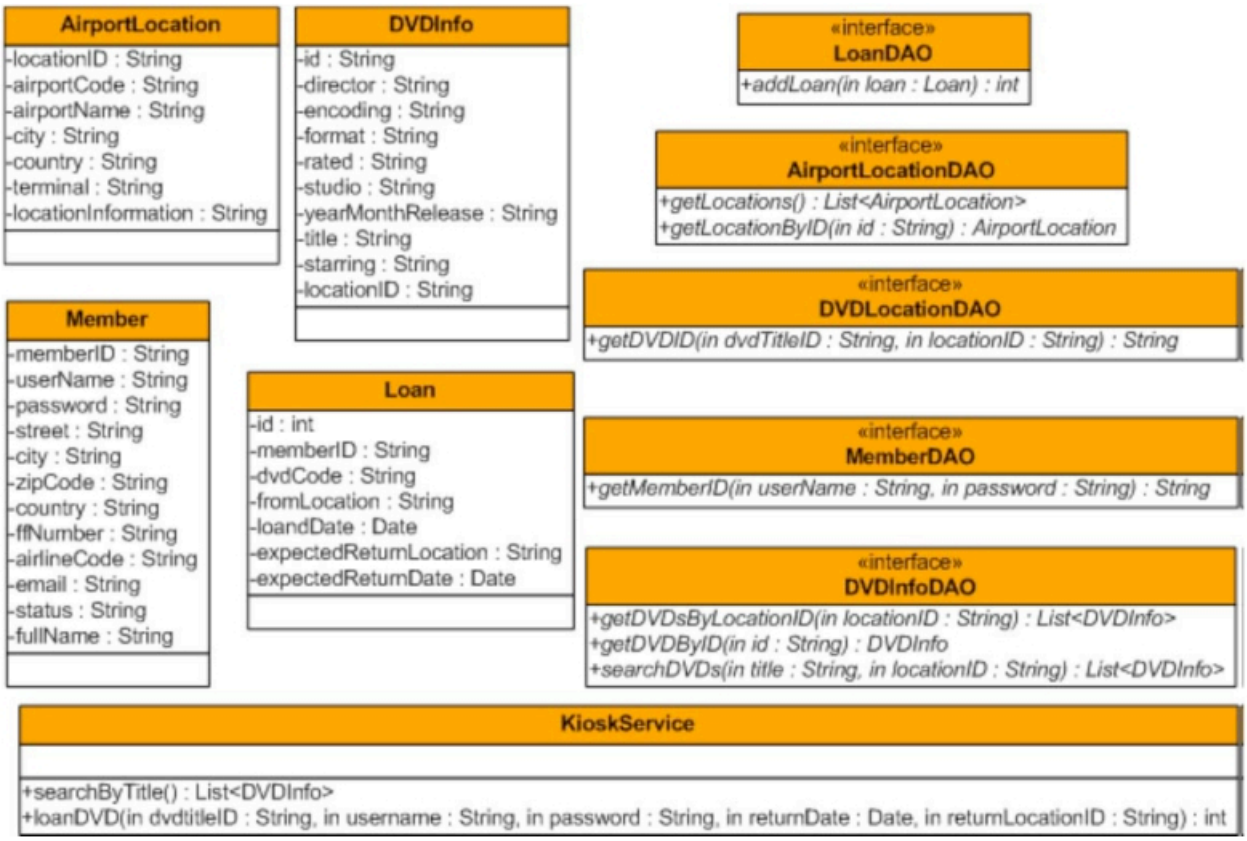
After importing the Java source code from **~ / StudentW ork/ Labs/ SpringConfiguration/ src** , import the contents of the **~ / StudentW ork/ Labs/ SpringConfiguration/ resources** into the project **src** directory by repeating the same steps. Once completed, the file **ApplicationContext.xml** will be in the **src** folder.

The case study used in this course is an application that allows customers to go to a Kiosk at a particular `AirportLocation` and search for DVDs, which can later be loaned from this location and dropped of at some destination.

All access to the DataSource is implemented using the DAO Patten and is implemented with the DAO classes, which (at least for the moment) are mock implementations. (In a subsequent exercise we will use DAO implementations that access the database that we set up earlier).

The main goal of this exercise is to configure the KioskService bean and its dependencies.

The pre-built TestClient will be used to check if the configuration has been completed successfully.



Step 2: Study the constructor of `config.bo.KioskService.java`

The main point of this exercise is learning how to properly configure the constructor for this class within the Spring container. Notice that the constructor requires a single parameter of type `AirportLocation`.

The airport location is available from the `getLocationByID` method on the `AirportLocationDAO` . The trick is figuring out how to specify that this is a factory that provides the object you need within the Spring XML configuration file (**ApplicationContext.xml**) and *not* using any Java programming.

Step 3: Inspect the file *`ApplicationContext.xml`*

`ApplicationContext.xml` is located in the *`src`* directory. Notice that a bean called *KioskService* has already been defined; please do not alter the name of this bean because the TestClient will be using this particular name to lookup the Bean.

Step 4: Add the DAO classes to the Configuration file

Define the five (mocked) DAO implementation classes residing in the `config.dao . mock` package

Hint:

- **config.dao.mock.AirportLocationDAOImpl**
- **config.dao.mock.DVDInfoDAOImpl**
- **config.dao.mock.DVDLocationDAOImpl**
- **config.dao.mock.LoanDAOImpl**
- **config.dao.mock.MemberDAOImpl**

Step 5: Complete the KioskService Definition in the Configuration file

The KioskService requires a single parameter in the constructor of type AirportLocation which can be obtained by invoking the (non-static) `getLocationByID` method on the `AirportLocationDAO` bean (an example of an ID would be BUR-2)

In addition, the KioskService contains four properties that must be set dvdInfoDAO, dvdLocationDAO, loanDAO and memberDAO.

Step 6: Validate *`ApplicationContext.xml`*

Eclipse will attempt to validate the Spring configuration against the given schema. Eclipse will attempt to access the schema from the remote location that is declared in the schema declarations. This will work, if the machine has Internet access.

This validation takes place on initial import, as well as when you edit the file. Break the configuration file and see that error messages are generated. Return the configuration to a valid state prior to continuing.

Step 7: Run the TestClient

You can execute the TestClient by right-clicking on it and selecting **Run As -> J ava Application** .

Step 8: Carefully inspect the output

Spring produces a large amount of logging information, so take a close look at the information provided by Spring. In most cases the error messages provided should give you enough information about any problems that exist.

You can stop the logging information by copying the *log4j.properties* file that you created in the previous exercise to the *src* directory.

When everything works correctly, you should see that a Loan was added under Number 9494 for member 93947.

Exercise 3. ADVANCED CONFIGURATION

Overview	
relationship in the configuration file.	In this exercise you will configure multiple Kios You will also be using a properties file to define
Objective	Re-use bean definitions and make use of property files.
Builds on Previous Labs	Standalone
Time to Complete	30 minutes

Task List

Overview of Steps

1. **Introduction**
2. **Inspect *`KioskServices.properties`***
3. **Edit *`ApplicationContext.xml`***
4. **Define a new Bean called `KioskServiceUSA`**
5. **Define a new Bean called `KioskServiceNL`**
6. **Declare the PropertyConfigurer bean**
7. **Validate *`ApplicationContext.xml`***
8. **Run the TestClient**
9. **Carefully inspect the output**

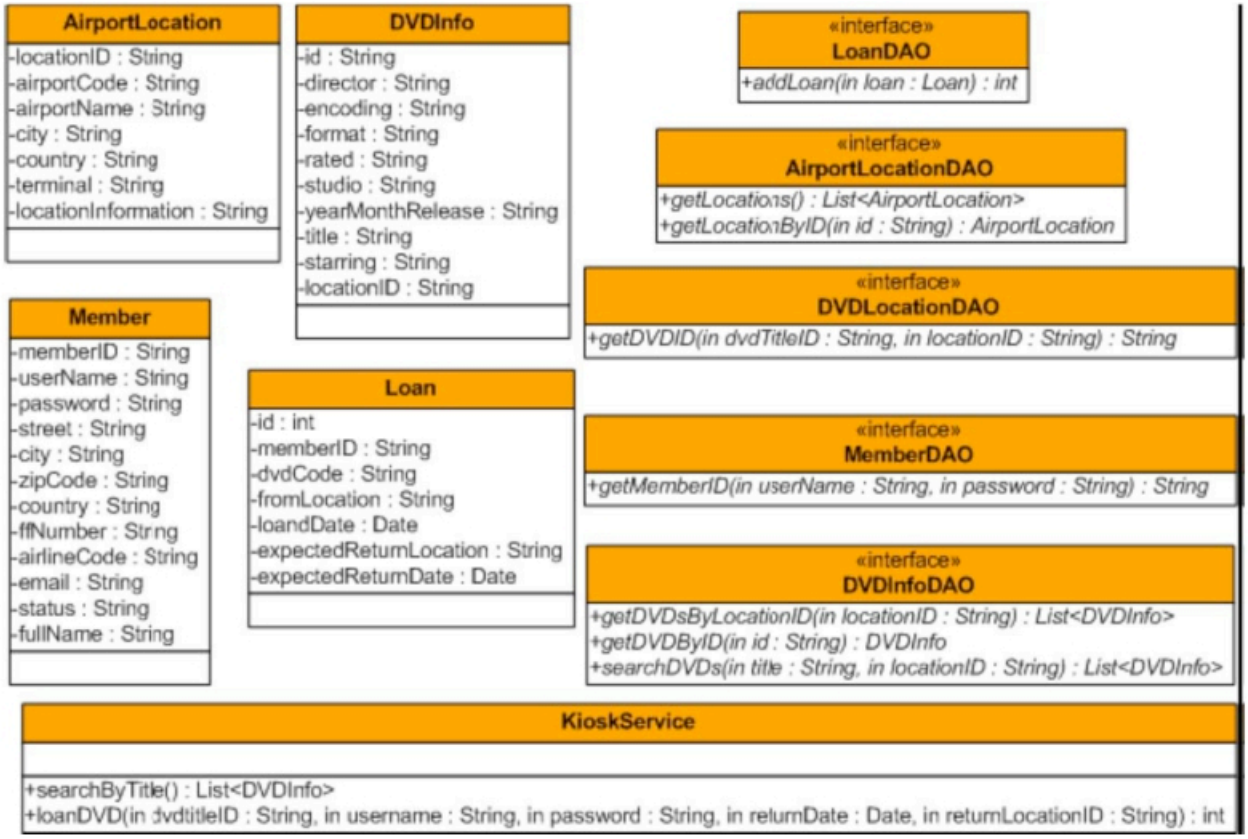
Step 1: Introduction

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Create a new Java project called **AdvConfiguration** , add the **SPRI NG** user library to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ AdvancedConfiguration** into the project's **src** directory as explained in the **Configuring Dependencies** lab.

During this exercise you will be adding a second KioskService to the application. In this case you will be creating a service located in the USA and one located in The Netherlands.



To minimize the amount of configuration needed, you will be re-factoring the Definition of the KioskService from the pervious exercise and make this an abstract (HINT!) bean, from which other KioskServices will inherit properties.

To make the configuration as easy as possible, you are going to make sure that all configurable settings that might need adjustments at deploy time are defined in a properties file. This helps ensure the configuration file does not have to be edited before an application goes into production.

? Step 2: Inspect *KioskServices.properties*

Located in the `advanced` package.

This properties file contains two entries:

- location.id.nl= AMS-1
- location.id.usa= BUR-2

You will de referencing these two values from the configuration file.

? Step 3: Edit *ApplicationContext.xml* and modify the KioskServiceTemplate

`ApplicationContext.xml` was located in the *resources* directory.

Notice that the name of the KioskService bean has been changed into *KioskServiceTemplate.*

Remove the `constructor-arg` from this definition. (If you don't, this will become the default location for ALL child definitions of this template). By removing the `constructor-arg` element, you are making an abstract definition of this bean.

Make KioskServiceTemplate into a template by adding the abstract= "true" attribute.

Step 4: Define a new Bean called `KioskServiceUSA`

Define a new Bean called `KioskServiceUSA` which is a child definition of the `KioskServiceTemplate` bean.

Make sure you add the constructor parameter.

When specifying the value of the locationID, instead of hard-coding a value (e.g. BUR-2), make a reference to an entry in a properties file called *location.id.usa.*

Step 5: Define a new Bean called `KioskServiceNL`

Define another KioskService bean called *KioskServiceNL* which is also a child definition of the `KioskServiceTemplate` bean.

Make sure you add the constructor parameter.

When specifying the value of the locationID, make a reference to an entry in a properties file called *location.id.nl.*

Step 6: Declare the PropertyConfigurer bean

Declare the *PropertyConfigurer* (`org.springframework.beans.factory.config.PropertyPlaceholderConfigurer`) and define the properties file location as *advanced/KioskServices.properties.*

Step 7: Validate *`ApplicationContext.xml`*

Make sure your XML file is valid against it's DTD before you continue to run the client by saving it and checking for errors.

Step 8: Run the TestClient

Run the TestClient as a Java application.

Step 9: Carefully inspect the output

As mentioned previously, get familiar with both the normal and the abnormal output from Spring.

When everything works correctly, you should see that two Loans were added under Number 9494, one in USA and one in NL.

Exercise 4. SPRING AOP – COMBINED ADVICE

Overview	
	In this exercise you will be creating advice that debugging tool. The advice adds the capability calls — reporting the method, the values returr By changing the Spring configuration this debu deployment and restoring the full performance
Objective	Create a useful debugging advice tool
Builds on Previous Labs	Standalone
Time to Complete	60 minutes

Overview of Steps

1. **Introduction**
2. **Implement `aop.advices.BeforeAfterLoggingAdvice – before()`.**
3. **Implement `aop.advices.BeforeAfterLoggingAdvice – buildMethodCall()`.**
4. **Implement `aop.advices.BeforeAfterLoggingAdvice – afterReturning()`.**
5. **Implement `aop.advices.BeforeAfterLoggingAdvice – afterThrowing()`.**
6. **Edit *`ApplicationContext.xml – to add this proxy as needed.`***
7. **Run the TestClient and carefully inspect the output.**

Step 1: Introduction

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

This lab uses a similar starting point to the previous project. You will be able to use the code you create in this project in your own projects.

In this project you will create advice that can be applied to monitor all the method calls of any classes you specify in the Spring configuration file. This advice is a valuable debugging tool that you can add to your toolkit. For example, you can peruse the output for Exceptions and see what the parameters were for methods that generated exceptions.

Create a new Java project called **SpringAOP-BeforeAfter** add the **SPRI NG** user library to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ SpringAOP-BeforeAfter** into the project's **src** .

This code uses AOP, so you will need to add the following to the build path:

com.springsource.org.aopalliance-1.0.0.jar

Navigate to the **~ / StudentW ork/ Tools/ SpringDependencies** folder and add the needed JAR file to your build path.

? Step 2: Implement `aop.advices.BeforeAfterLoggingAdvice – before()`.

This step and the next 3 steps will edit the `BeforeAfterLoggingAdvice` class.

The BeforeAfterLoggingAdvice class will intercept before method calls, after method calls and exceptions thrown by various methods. It will have three public methods

- `before()` for intercepting calls before they are executed,
- `afterReturning()` for intercepting method calls after they return normally, and
- an `afterThrowing()` method for intercepting exceptions that are thrown by a method.

In this step, you will create the method `before()`. Later steps will create the methods `afterReturning()` and `afterThrowing()`.

The trick to writing this advice is to write the code generically, so that the advice can be applied to any object. To do this you will need to use Java reflection (which is programmatic introspection on objects).

Next create the `before()` method with code that will output a String that corresponds to the method call that is being intercepted. Sample output could be:

Called: ShoppingCart.getTotalCharge("ups", 19);

A: Add to the Advice that it implements the interface `org.springframework.aop.MethodBeforeAdvice`. You can then select **SOURCE** and then **Override/ I mplement Methods** and select `before` to have Eclipse automatically generate the method signature:

```
public void before(Method method, Object[] args, Object target)
        throws Throwable
```

B: In this method use a **`java.lang.StringBuilder`** object to build the information about the method call. Use the helper method **`buildMethodCall()`** that you will create in the next step. The complete code for the method `before()` method is:

```
public void before(Method method, Object[] args, Object target)
            throws Throwable {
      StringBuilder buffer = new StringBuilder();
      buffer.append("Called: ");
      buildMethodCall(buffer, target, method, args);
      output(buffer);
}
```

In addition to the methods mentioned above, this class also needs the `buildMethodCall()` and an `output()` method (which we have provided for you). In the next step, you will create the helper method called `buildMethodCall()` that will

build a string representing a method call. This method is be used by the three public methods `before(), afterReturning()` and `afterThrowing()`.

? Step 3: Implement `buildMethodCall()`.

The signature of this method is:

```
void buildMethodCall
(StringBuilder buffer, Object target, Method method, Object[] args)
```

The first parameter is the StringBuilder used to build information about the method call.

The other 3 parameters are information about the method call. The **target** is the object being called, the **method** is a Method object containing reflected information about the method being called and the **object array** contains the arguments passed to the method. For example, consider a case where the method being called is: `getInfo( "Hello", 12)` of a class called `Person`. This method should append to the `StringBuilder` , the `String:` **Person.getI nfo(" hello" , 12) ;**

You can get the name of the class (Person in this example) using `target.getClass().getSimpleName()`. You can get the name of the method using `method.getName()` . You can convert each of the parameters to a string that represents it using `args[i].toString()`. The complete code for this method is:

```
void buildMethodCall(StringBuilder buffer,
                              Object target, Method method, Object[] args) {
   buffer.append(target.getClass().getSimpleName());
   buffer.append(".");
   buffer.append(method.getName());
   buffer.append("(");
   boolean firstArg = true;
   for (Object arg : args) {
      if (!firstArg) {
        buffer.append(", ");
      }
      firstArg = false;
      boolean isString = (arg.getClass() = = String.class);
      if (isString) buffer.append('"');
      buffer.append(arg.toString());
      if (isString) buffer.append('"');
   }
   buffer.append(");");
}
```

Step 4: Implement `afterReturning()`.

The signature for this method is:

```
public void afterReturning
(Object result, Method method, Object[] args, Object target)
throws Throwable
```

You can again use Eclipse to generate a stub implementation by first adding the **AfterReturningAdvice** interface to the class signature. This method should work in a similar fashion to the `before()` method with the addition of showing the returned value. It should generate a StringBuilder that contains the return value as well as the method call that generated it. Sample output could be:

Returned: 45 = ShoppingCart.getTotalCharge("ups");

Step 5: Implement `afterThrowing()`.

The signature for this method is:

```
public void afterThrowing
(Method method, Object[] args, Object target, Exception e)
```

For completeness, add that this class implements the interface **Throw sAdvice** . Technically, this is not required as the interface is empty, however this is good practice to show the intention of your code. This method will work like the `afterReturning` method – except that it will show the Exception that was thrown instead of the returned value.

Sample output is:

```
Exception aop.InvalidMemberPasswordException thrown by
MemberDAOImpl.getMemberID("j.wirth", "wrongPassword");
```

Step 6: Edit *`ApplicationContext.xml – to add this proxy as needed.`*

There are several things that you will need to do to this file. You will need to add a bean for the advice class that you just created. You will also need to add the Proxy, and configure the AutoProxy class to add this `BeforeAfterLoggingAdvice` to those classes that you want to monitor for debugging purposes.

First, add a bean to the *`ApplicationContext.xml`* for the `BeforeAfterLoggingAdvice` class that you created. Call this bean: `BeforeAfterLoggingAdvice`.

Next, you will want to add an `AutoProxyCreator`.

```
< bean id= "BeforeAfterLoggingAutoProxy" class=
"org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    < property name= "beanNames" value= "KioskService,MemberDAO" />
    < property name= "interceptorNames">
        < list>
            < value> BeforeAfterLoggingAdvice< /value>
        < /list>
```

```
        < /property>
    < /bean>
```

You can put the names of whichever Beans you want to monitor with this proxy as the value of the `beanNames` property. We have put the two beans `KioskService` and `MemberDAO`.

Step 7: Run the TestClient and carefully inspect the output.

Run `aop.test.TestClient` as a Java application.

The output will be directed to both the console and a file called logging.txt. You can view the logging.txt file by using Windows Explorer to find the root directory of the project in your workspace.

The formatting of the output is being controlled by the log4j.properties file. The reason to use log4j is that it makes it easy to direct the output from a file to the Console or to any other destination.

Notice that you can easily change the set of beans that you are monitoring with this proxy. When you go into production, you can turn off the proxy completely – thereby getting the full-speed performance from the underlying unproxied original objects.

Exercise 5. SPRING AOP – AROUND ADVICE

Overview	
then remove it for production deployments.	In this exercise you will be creating advice that tool that you can use to help optimize the perfo This advice profiles method calls and reports th available granularity on most computers is abo all results are accurate to + /- 1/60 th of a seco will add a method to the test program that invo addition, we have added a 15ms sleep to one o As with all AOP advice in Spring, you can decla (and the generation of proxies). Spring makes
Objective	Create a useful profiling advice proxy.
Builds on Previous Labs	Standalone
Time to Complete	35 minutes

Overview of Steps

1. **Introduction**
2. **Implement `aop.advices.ProfilerAdvice – invoke()`.**
3. **Edit *`ApplicationContext.xml – to add this proxy as needed.`***
4. **Run the TestClient and carefully inspect the output.**

Step 1: Introduction

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

This lab uses a similar starting point to the previous project. You will be able to use the code you create in this project in your "real-world" projects.

In this project you will create advice that you can configure to monitor the elapsed time that a given method consumes. This can help you find which methods consume a lot of CPU time – and are therefore candidates for optimization.

Create a new Java project called **SpringAOP-Profiler** add the **SPRI NG** user library to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ SpringAOP-Profiler** into the project's **src** directory as explained in the **Configuring Dependencies** lab.

This code uses AOP, so you will need to add the following to the build path:

com.springsource.org.aopalliance-1.0.0.jar

Navigate to the **~ / StudentW ork/ Tools/ SpringDependencies** folder and add the needed JAR file to your build path.

This code uses log4j as its output mechanism. This is easily changed to use System.out.println() or some other mechanism. The log4j libraries and configuration files have been provided. You do not need to understand the details of the log4j.properties file to understand the aspect oriented part of this code. The log4j output routines are invoked simply: `logger.debug(String)`. If you desire – inspect the log4j.properties file; if you have any questions about it feel free to ask the instructor.

? Step 2: Implement `invoke()`.

Edit the `ProfilerAdvice` class. We have supplied a couple of the utility methods similar to those that you worked with previously. In this exercise, you are going to be focusing on the `invoke()` method.

The trick to writing this advice is to write the code totally generically, so that the advice can be applied to any object. You will need to do a certain basic amount of "reflection"

In this step, you will create the `invoke()` method. This method will output a String that corresponds to the amount of time that a method call took, together with the signature of the method call. Sample output could be:

```
15890ms: SomeClass.method(123, "Hello");
```

A: Modify the `ProfilerAdvice` class declaration so that it implements the `org.aopalliance.intercept.MethodInterceptor` interface.

B: Add the `invoke()` method that the `MethodIntercepto` r interface requires.

You should get a starting time by calling `System.currentTimeMillis()` .

You should then call `proceed()` with the original method invocation.

After the method invocation returns you want to get the end time by calling `System.currentTimeMillis()` .

The elapsed time is the difference between the beginning time and the end time.

You then want to display the output.

Notice that there is already a `buildMethodCall()` method to help you. That method call does not directly take the `MethodInvocation` parameter that the `invoke()` method has. You will need to get the parameters you need for `buildMethodCall()` from the `MethodInvocation` .

You can then display the resultant string using `output()`.

Make sure to take into consideration the following factors.

1. The `proceed`() method may return a value. This value has to be returned to the caller.
2. The `proceed`() method may throw an Exception. You must produce output in this case AND you must make sure that the Exception is thrown back to the caller.

Hint: Wrap the `proceed()` call in a try block, returning any value from there. Then use a `finally` clause to capture the elapsed time and produce the output no matter what happens in the invoked method.

? Step 3: Edit *ApplicationContext.xml – to add this proxy as needed.*

You will have to make three changes to the Spring config file.

A: Add the `TestClientMain` itself as a bean in the config file. Name it " `TestClient`". Examine the code for `TestClientMain` – you will see that the `main` routine gets a reference to an instance of itself from Spring using a static variable. This is needed so that the proxy can be applied to the test method.

B: Add the `ProfilerAdvice` class as a bean. Name it `ProfilerAdvice` .

C: Decide which beans you want to profile. Add the names of those beans to the `beanNames` property of the `ProfilerAutoProxy` . Clearly, you can change these at runtime – depending on what you want to profile.

? Step 4: Run the TestClient and carefully inspect the output.

Run `aop.test.TestClient` as a Java application.

The output will be directed to both the console and a file called logging.txt. You can view the logging.txt file by using Windows Explorer to find the root directory of the project in your workspace.

The formatting of the output is being controlled by the log4j.properties file. The reason to use log4j is that it makes it easy to direct the output from a file to the Console or to any other destination.

Here's a question for you to think about. When you add the TestClient as a class to be profiled, you see output for when the test() method is called. You do not see output for when the doWork() method is called.

Another question to think about. Could you have written this profiler proxy using the techniques of the prior lab – namely having separate `before()`, `afterReturning()` and `afterThrowing()` methods?

Exercise 6. S PRING IOC AND AOP

Overview	
customers are searching for. might be thrown.	In this exercise you will be adding statistics inf using Spring AOP to add functionality to the ap In addition you will be adding an Advice that w
Objective	Become familiar with the Spring containers and Spring AOP
Builds on Previous Labs	Standalone
Time to Complete	30 minutes

Overview of Steps

1. **Inspect the classes**
2. **Implement `aop.advices.MemberThrowsAdvice`**
3. **Implement `aop.advices.SearchStatisticsAdvice`**
4. **Examine *`ApplicationContext.xml`***
5. **Add the `MemberThrowsAdvice` advice to the application**
6. **Add the `DVDStatistics` component**
7. **Add the `SearchStatisticsAdvice` advice to the application**
8. **Validate *ApplicationContext.xml***
9. **Run the TestClient**
10. **Carefully inspect the output**

Step 1: Inspect the classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

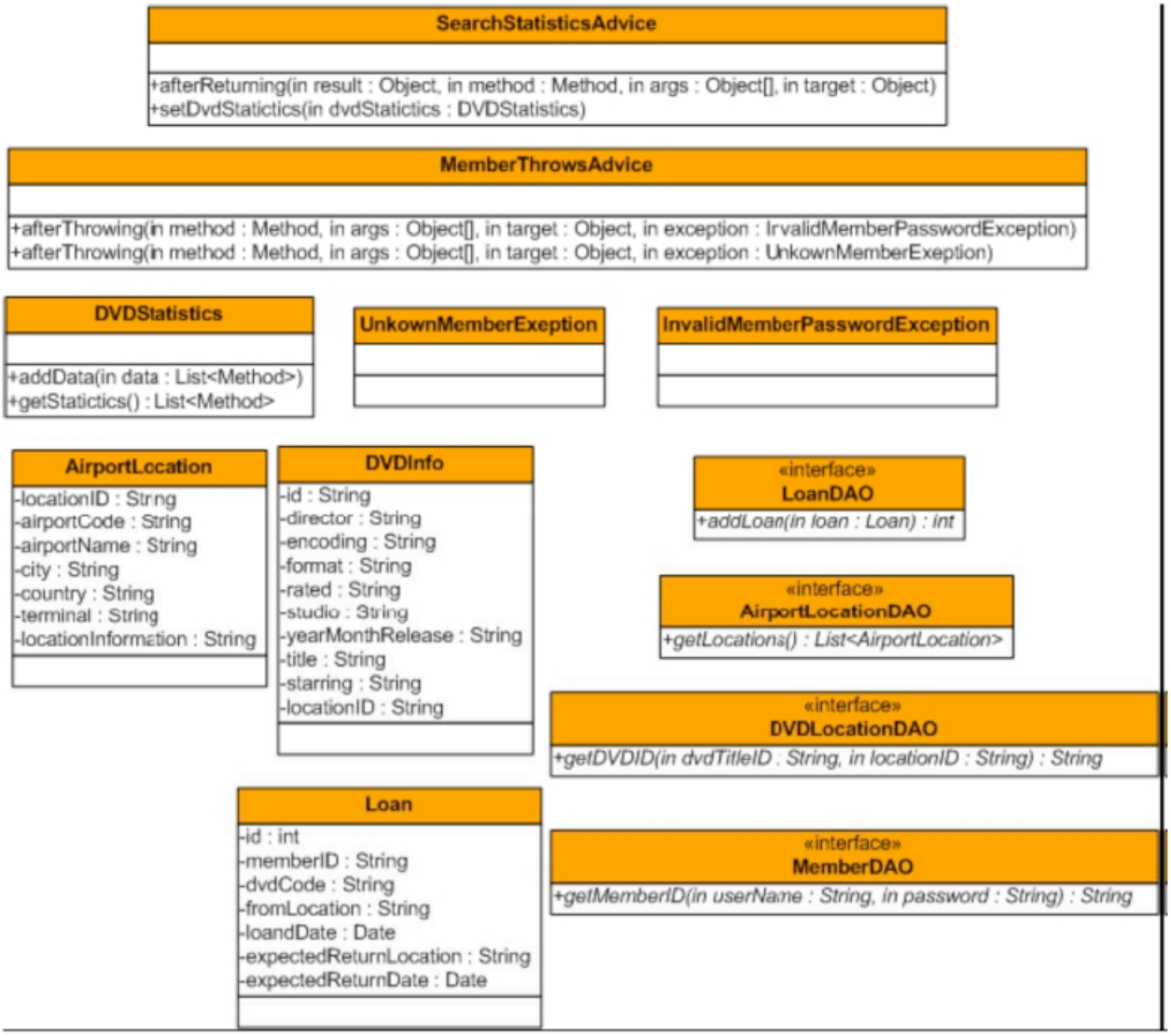
Create a new Java project called **SpringAOP** , add the **SPRI NG** and **SPRI NG-DEPENDENCI ES** user libraries to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ SpringAOP** into the project's **src** directory as explained in the **Configuring Dependencies** lab.

This code uses AOP, so you will need to add the following to the build path:

com.springsource.org.aopalliance-1.0.0.jar

Navigate to the **~ / StudentW ork/ Tools/ SpringDependencies** folder and add the needed JAR file to your build path.

Please take a moment and examine the diagram below, focusing on the `aop.advices.MemberThrowsAdvice` and the `aop.advices.SearchStatisticsAdvice` diagrams.



? Step 2: Implement `aop.advices.MemberThrowsAdvice`

Start by implementing the correct interface to make this an interceptor of Exceptions.

Implement the method(s) to make this an interceptor that listens for two types of exceptions: `InvalidMemberPasswordException` and `UnknownMemberException` .

When one of these Exceptions is intercepted, perform some kind of logging (`System.out` is fine), in which you write the method that was invoked when the exception occurred and the error message.

? Step 3: Implement `aop.advices.SearchStatisticsAdvice`

Implement the correct interface to make this a valid interceptor that will intercept *`after`* the method returns.

Declare an instance variable of type DVDStatistics, and make sure that the value of this instance variable can be injected into the advice using setter injection (i.e. implement a setter method).

> DVDStatistics (implemented for you) is a helper component (implemented as a singleton) that contains two methods: `public void addData(List<DVDInfo> data) and public Map<String, Integer>getStatistics()`

Implement the required method for handling processing after returning. When the response is intercepted, cast the return value of the method into a List<DVDInfo> and add it to the DVDStatistics component.

? Step 4: Examine *ApplicationContext.xml*

Notice that the name of the KioskService bean: *KioskServiceTarget*.

Most of the work will have to take place in this file; you will have to define the advices, the advisors, the pointcuts and the proxies.

? Step 5: Add the `MemberThrowsAdvice` advice to the application

Add the aop.advices.MemberThrowsAdvice class to the configuration file as a bean.

To apply the advice to the application, you will be using the autowire functionality, provided by the BeanNameAutoProxyCreator class. Name the proxy `ThrowsProxy` and make sure that the bean wrapped by this proxy is the MemberDAO bean.

Make sure you register the MemberThrowsAdvice as an interceptor for this proxy.

? Step 6: Add the `DVDStatistics` component

Add the aop.bo.DVDStatistics component to the configuration file. Keep in mind that this class does not have a public constructor and is implemented as a singleton. An instance can be obtained using the static getInstance method.

? Step 7: Add the `SearchStatisticsAdvice` advice to the application

Add aop.advices.SearchStatisticsAdvice to the configuration file as a bean. Make sure you inject an instance of DVDStatistics using setter injection.

Add an Advisor for `SearchStatisticsAdvice` (you can give it a name of `DVDStatAdvisor`) and make sure this advice is only applied to all search methods of the `KioskService`. In order to accomplish this, you will need to set up a bean of type `org.springframework.aop.support.RegexpMethodPointcutAdvisor` with correct properties for the advice and the pattern to apply that advice.

Finally, you will need to create the `KioskService` proxy and apply the `DVDStatAdvisor` to the `KioskService`. This will require using the `ProxyFactoryBean` with properties set up for the interface you are setting up the proxy for, the target of the service, and the name of the interceptor that you just set up.

? Step 8: Validate *ApplicationContext.xml*

Make sure your XML file is valid against its schema before you continue to run the client by saving the file and checking for errors.

Step 9: Run the TestClient

Run aop.test.TestClient as a Java application.

Step 10: Carefully inspect the output

When everything works correctly, you should see an overview of the statistics obtained and that two Exceptions have been intercepted:

```
[java] Doing random searches to test DVD stats
[java] The Pianist (Widescreen Edition) - 11
[java] 2 Fast 2 Furious (Widescreen Edition) -11
[java] The League of Extraordinary Gentlemen (Widescreen Edition) - 18
[java] Bend It Like Beckham (Widescreen Edition) - 13
[java] The Life of David Gale (Widescreen Edition)- 14
[java] ----------------------------------
[java] Invalid Password when invoking public abstract java.lang.String aop.
dao.MemberDAO.getMemberID(java.lang.String,java.lang.String)
throws aop.MemberException
[java] With params:
[java] j.wirth
[java] wrongPassword
[java] Should have shown 'wrong password' message from ThrowsAdvice
[java]----------------------------------
[java] Unknown Member when invoking public abstract
java.lang.String aop.dao.MemberDAO.getMemberID(java.lang.String,java.lang.String)
throws aop.MemberException [java] With params:
[java] wrongName
[java] wrongPassword
[java] Should have shown 'wrong name' message from ThrowsAdvice
[java]----------------------------------
```

Exercise 7. Spring IoC and Annotation Based AOP

Overview	
Annotation based AOP. customers are searching for. might be thrown.	This exercise is identical in function to the prev In this exercise you will be adding statistics inf using Spring AOP to add functionality to the ap In addition you will be adding an Advice that w
Objective	Become familiar with the Spring containers and Annotation based Spring AOP
Builds on Previous Labs	Standalone
Time to Complete	45 minutes

Overview of Steps

1. **Create the project and inspect the classes**
2. **Create the needed pointcuts in `aop.advices.Pointcuts`**
3. **Implement `aop.advices.SearchStatisticsAdvice`**
4. **Implement `aop.advices.MemberThrowsAdvice`**
5. **Edit *`ApplicationContext.xml`***
6. **Add the `MemberThrowsAdvice` advice to the config file**
7. **Add the `DVDStatistics` component to the config file**
8. **Add the `SearchStatisticsAdvice` advice to the config file**
9. **Validate *ApplicationContext.xml***
10. **Run the TestClient**
11. **Carefully inspect the output**

[?] Step 1: Create the project and inspect the classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Create a new Java project called **Spring-DeclarativeAOP** , add the **SPRI NG** user library to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ Spring-DeclarativeAOP** into the project's **src** directory as explained in the **Configuring Dependencies** lab.

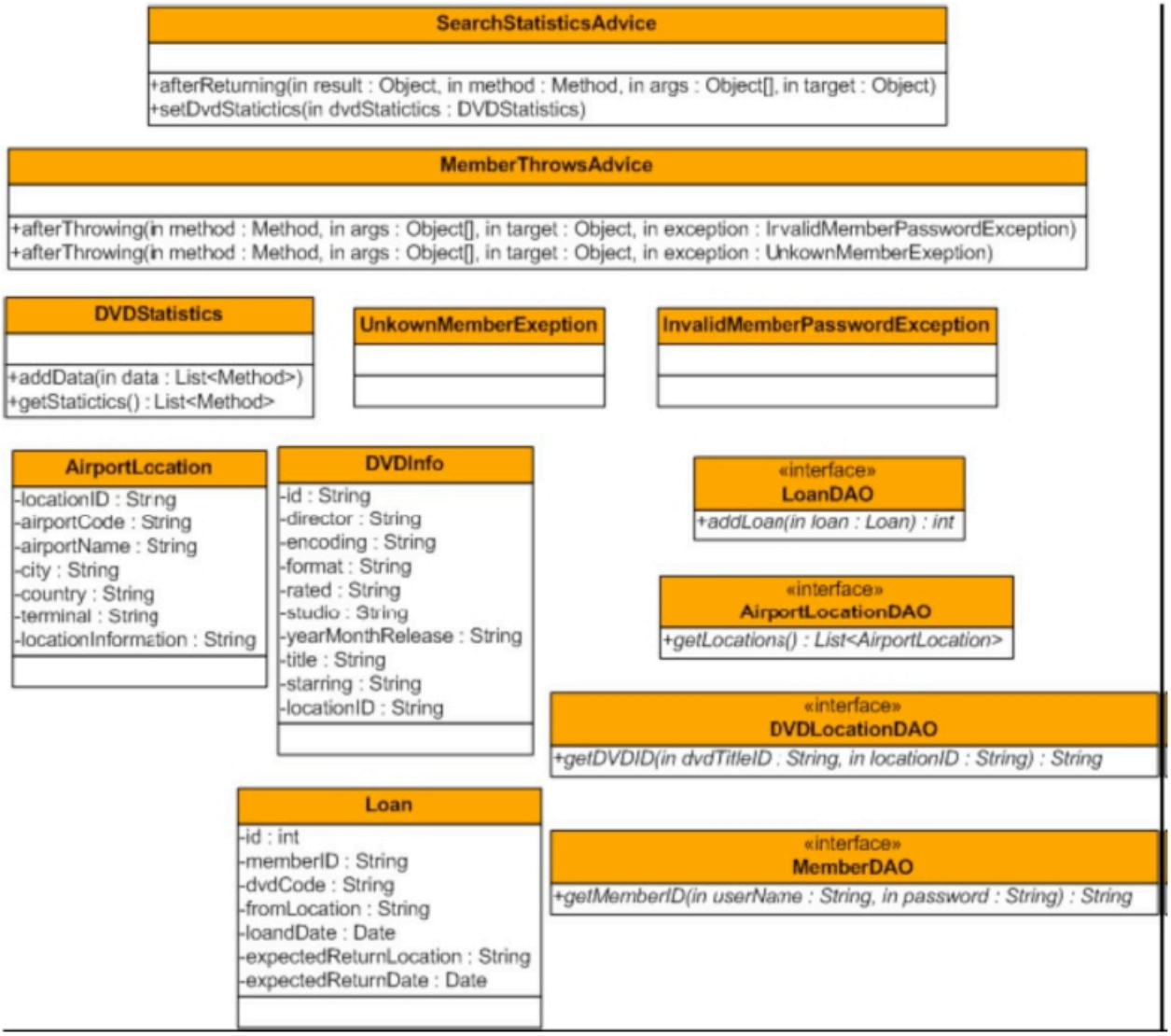
To use AspectJ in your Spring project you must add the AspectJ JAR files to the build path. The required JAR files are:

com.springsource.org.aspectj.tools-1.6.6.RELEASE.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

Navigate to the **~ / StudentW ork/ Tools/ SpringDependencies** folder and add the needed JAR files to your build path.



Step 2: Create the needed pointcuts in `aop.advices.Pointcuts`

Edit the class **aop.advices.Pointcuts.** Give it the annotation **@Aspect.**

Add a pointcut named inMemberDAO(). It should be annotated as the pointcut: `"execution(* aop.dao.MemberDAO.*(..))"`.

Add a pointcut named `inKioskSearchMethod()`. It should be annotated as the pointcut: `"execution(* aop.bo.KioskService.search*(..))"` .

Step 3: Implement `aop.advices.SearchStatisticsAdvice`

Add the `@Aspect` annotation to the class.

Declare an instance variable of type DVDStatistics, and make sure that the value of this instance variable can be injected into the advice using setter injection (i.e. implement a setter method).

> DVDStatistics (implemented for you) is a helper component (implemented as a singleton) that contains two methods: `public void addData(List<DVDInfo> data)` and `public Map<String, Integer>getStatistics()`

Add a method called `afterReturning` that will intercept returns from calls to search methods . Implement the required method with a single parameter: `List<DVDInfo> result`. When the response is intercepted, add the `result` to the injected DVDStatistics component.

Add an `AfterReturning` annotation to the `afterReturning()` method. The annotation should have two parameters, one to declare the pointcut, and the other to specify that the return value of the AOP'd method will be given in the `result` argument.

Step 4: Implement `aop.advices.MemberThrowsAdvice`

Add the `@Aspect` annotation to the class.

A: Examine the method `methodCall2String()` which creates a `String` that represents the joinpoint. You may want to look up the JavaDocs for `JoinPoint` and `Signature`.

B: Create a method:

```
afterThrowing
  (JoinPoint joinPoint, InvalidMemberPasswordException ex)
```

Have this method print the message: `"Invalid Password when invoking "` together with the result of invoking `methodCall2String()` on the joinpoint.

Add an `@AfterThrowing` annotation to this method. The annotation should have two parameters: the first specifying that the pointcut should be only when an exception in thrown from a method in the `MemberDAO` class. The second should specify that the exception should be passed in the variable `ex`.

C: Create a method:

```
afterThrowing
  (JoinPoint joinPoint, UnknownMemberException ex)
```

Have this method print the message: `"Unknown member when invoking "` together with the result of invoking `methodCall2String()` on the joinpoint.

Add an `@AfterThrowing` annotation to this method. The annotation should have two parameters: the first specifying that the pointcut should be only when an exception in thrown from a method in the `MemberDao` class. The second should specify that the exception should be passed in the variable `ex`.

Step 5: Edit *ApplicationContext.xml*

Since you are using annotations, there is very little work to do in the Spring config file.

Add a line:

```
<aop:aspectj-autoproxy/>
```

to the config file. This will instruct Spring to automatically read all the proxy related annotations and implement them as needed.

Step 6: Add the `MemberThrowsAdvice` advice to the config file

Add the aop.advices.MemberThrowsAdvice class to the configuration file as a bean.

You do not need to configure it with any advice or proxies. Spring will apply all the needed functionality based on the annotations that you have provided.

Step 7: Add the `DVDStatistics` component to the config file

Add the aop.bo.DVDStatistics component to the configuration file. Keep in mind that this class does not have a public constructor and is implemented as a singleton. An instance can be obtained using the static getInstance method.

Step 8: Add the `SearchStatisticsAdvice` advice to the config file

Add aop.advices.SearchStatisticsAdvice to the configuration file and make sure you inject an instance of DVDStatistics using setter injection.

You do not need to configure it with any advice or proxies. Spring will apply all the needed functionality based on the annotations that you have provided.

Step 9: Validate *ApplicationContext.xml*

Notice that you have added very little to the **ApplicationContext.xml** file. To create proxies you only had to do two things.

1: Add the `autoproxy` directive.

2: Add the beans that contain the advice – as regular Spring beans.

Make sure your XML file is valid against its Schema before you continue to run the client by saving the file and checking for errors.

Step 10: Run the TestClient

Run aop.test.Test Client as a Java application.

Step 11: Carefully inspect the output

When everything works correctly, you should see an overview of the statistics obtained and that two Exceptions have been intercepted:

```
[java] ------------------------------------
[java] Doing random searches to test DVD stats
[java] The League of Extraordinary Gentlemen (Widescreen Edition) - 18
[java] The Pianist (Widescreen Edition) - 11
```

```
[java] The Life of David Gale (Widescreen Edition) - 14
[java] 2 Fast 2 Furious (Widescreen Edition) - 11
[java] Bend It Like Beckham (Widescreen Edition) - 13
[java] -----------------------------------
[java] Invalid Password when invoking getMemberID(j.wirth, wrongPassword)
[java] Should have shown 'wrong password' message from ThrowsAdvice
[java] -----------------------------------
[java] Unknown Member when invoking getMemberID(wrongName, wrongPassword)
[java] Should have shown 'wrong name' message from ThrowsAdvice
```

Exercise 8. USING SPRING JDBC

Overview	
Helper classes provided by Spring.	In this exercise you will be implementing the D
Objective	Use the Spring DAO helper classes.
Builds on Previous Labs	Standalone
Time to Complete	75 minutes

Task List

Overview of Steps

1. **Inspect the classes**
2. **Implement `springjdbc.dao.jdbc.AirportLocationDAOImpl`**
3. **Implement `springjdbc.dao.jdbc.DVDInfoDAOImpl`**
4. **Implement `springjdbc.dao.jdbc.DVDLocationDAOImpl`**
5. **Inspect `springjdbc.dao.jdbc.query.LoanInsertQuery`**
6. **Implement `springjdbc.dao.jdbc.LoanDAOImpl`**
7. **Edit *ApplicationContext.xml***
8. **Add the datasource**
9. **Inject the datasource into the DAO classes**
10. **Validate *ApplicationContext.xml***
11. **Run the TestClient**
12. **Carefully inspect the output**
13. **Inspect the database**

Step 1: Create the project and inspect the classes

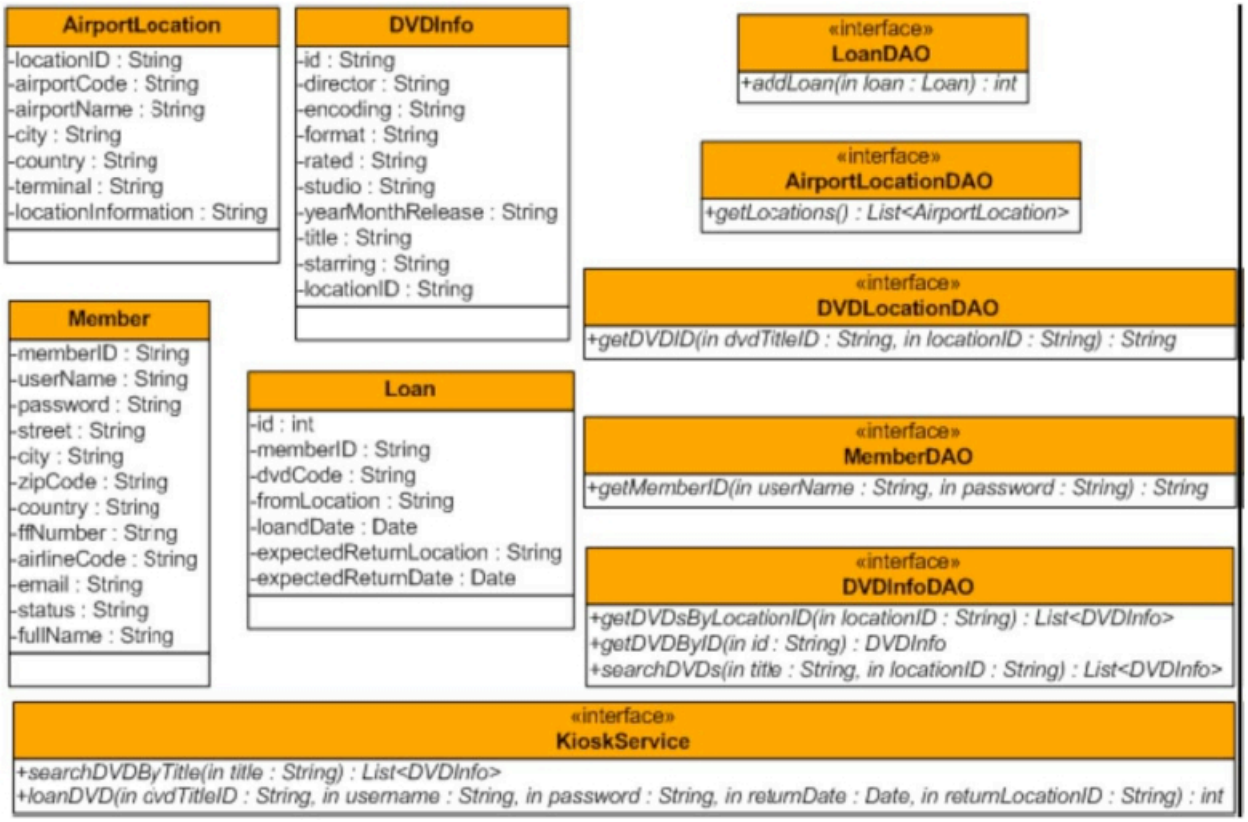
Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Create a new Java project called **SpringJ DBC** , add both the **SPRI NG** and the **DERBY** user libraries to the build path, and import the source code and resources from **~ / StudentW ork/ Labs/ Spring-J DBC** into the project's **src** directory as explained in the **Configuring Dependencies** lab.

Ensure that the database server is running. In addition, you may want to reset the database by running the appropriate script files. Refer to the first exercise for instructions on starting, resetting, populating, and testing the database.

Please take a moment to examine the `…dao.jdbc` classes that are shown on the diagram below.



? Step 2: Implement `springjdbc.dao.jdbc.AirportLocationDAOImpl`

Make sure your class extends the appropriate Spring helper class to make this a Spring JDBC DAO class.

Implement the `getLocations()` and `getLocationByID(String id)` methods of the `AirportLocationDAOImpl` as described below.

When implementing the `getLocations()` method, obtain a `JdbcTemplate` (later in the lab, the DataSource will be setter-injected). Use the template to execute the SQL statement provided in the `sql` variable.

The results of this query can be processed using a `RowCallbackHandler` .

To populate an instance of `AirportLocation` a helper method has been provided (`private AirportLocation populate(ResultSet rs) throws SQLException`).

The result of this query must be stored in the ArrayList called 'result'.

The code for this is:

```
//Obtain the JdbcTemplate

JdbcTemplate template = getJdbcTemplate();

//Execute the sql statement, using a RowCallbackHandler to obtain the
results.

template.query(sql, new RowCallbackHandler() {

    public void processRow(ResultSet resultSet)

                                          throws SQLException {
```

```
            AirportLocation location = populate(resultSet);
            result.add(location);
        }
});
```

When implementing the `getLocationByID(String id)` method, obtain a `JdbcTemplate` and use the template to execute the SQL statement provided.

The results of this query can be processed using a `RowMapper`.

The result of this query must be stored in the ArrayList called 'locations'.

The code for this is:

```
//Obtain the JdbcTemplate
JdbcTemplate template = getJdbcTemplate();
//Execute the sql statement, using a RowMapper to obtain the results.
locations = template.query(sql, args, new RowMapper() {
    public Object mapRow(ResultSet resultSet, int i) throws SQLException
{
       return populate(resultSet);
     }
 });
```

Step 3: Implement `springjdbc.dao.jdbc.DVDInfoDAOImpl`

Make sure your class extends the appropriate Spring helper class to make this a Spring JDBC DAO class.

This class contains three public methods for obtaining information from the database. All these public methods make use of one private helper method `private List<DVDInfo> execute(String sql, Object[] args)` , which performs the actual execution of the statement.

You will have to implement `execute(...)` method. Obtain a `JdbcTemplate` and use the template to execute the SQL statement provided as a parameter of this method.

The results of this query can be obtained using a `RowCallbackHandler` .

To populate an instance of `DVDInfo` a helper method has been provided (`private DVDInfo populate(ResultSet rs) throws SQLException`).

The result of this query must be stored in the ArrayList called 'result'.

Following the pattern of the previous step will assist in writing the code.

Step 4: Implement `springjdbc.dao.jdbc.DVDLocationDAOImpl`

Make sure your class extends the appropriate Spring helper class to make this a Spring JDBC DAO class.

Implement the `getDVDId(...)` method of `DVDLocationDAOImpl`.

Obtain a `JdbcTemplate` and set up an `Object` array holding the titleID and locationID arguments that were passed in.

Execute the sql statement. Notice that this sql statement returns a single Object of type String.

Catch the exception that might be thrown by this method invocation to check if exactly one row was returned. (Hint: `IncorrectResultSizeDataAccessException`)

Rethrow the exception as a DAOException. The code for this method is:

```
public String getDVDId(String dvdTitleID, String locationID) {
  String sql =
  "SELECT DVDCODE FROM DVD WHERE DVD_TITLE_ID=? AND LOCATION_ID=?";
  //Obtain the template
  JdbcTemplate template = getJdbcTemplate();
  //Execute the sql statement.
  //Notice sql statement returns a single Object of type String.
  Object[] args = new Object[]{dvdTitleID, locationID};
 //Catch the exception that might be thrown by this method
 //invocation to check if only one (and just one) row was returned
 //Rethrow exception as a DAOException
 String dvdID = null;
 try {
   dvdID =
       (String) template.queryForObject(sql, args, String.class);
  } catch (IncorrectResultSizeDataAccessException e) {
    if (e.getActualSize() == 0) {
       throw new DAOException(
               "No records found when trying to obtain DVDID");
    }
       throw new DAOException(
            "Multiple records found when trying to obtain DVDID");
    }
 return dvdID;
}
```

? Step 5: Inspect `springjdbc.dao.jdbc.query.LoanInsertQuery`

This class does not require any coding.

This class is of type SqlUpdate and will be used by springjdbc.dao.jdbc.LoanDAOImpl to perform the actual insert of the loan intro the database.

Step 6: Implement `springjdbc.dao.jdbc.LoanDAOImpl`

Implement the `addLoan(Loan loan)` method of `LoanDAOImpl` .

Create an instance of the pre-built LoanInsertQuery class.

Create an instance of `GeneratedKeyHolder` . This class will contain the key that was generated by the database upon insertion of a record.

Insert the record into the database using the `Object[]` provided and the `keyHolder` you just instantiated. Afterwards, obtain the key value from the `KeyHolder` and return its value to the caller. The code for this method is:

```
public int addLoan(Loan loan) {
   Object[] args = new Object[]{loan.getMemberID(),
      loan.getDvdCode(), loan.getFromLocation(),
      loan.getLoanDate(), loan.getExpectedReturnLocation(),
       loan.getExpectedReturnDate()};
   //Create an instance of the pre-build LoanInsertQuery class
   LoanInsertQuery insertQuery =
                    new LoanInsertQuery(getDataSource());
   //Create an instance of GeneratedKeyHolder, this class will
   //contain the key that was generated by the database upon
   //insertion of the record.
   KeyHolder keyHolder = new GeneratedKeyHolder();
   //Insert the record into the database, using Object[]
   //provided and the keyHolder you just instantiated
   insertQuery.update(args, keyHolder);
   //Obtain the value of the generated key and
   //return this value to the caller.
   return keyHolder.getKey().intValue();
}
```

Step 7: Edit *ApplicationContext.xml*

`ApplicationContext.xml` is located in the resources directory.

Notice the name of the *KioskService* bean: *`KioskService`*

Step 8: Add the datasource

For this exercise you will be using a Datasource that is obtained using a DriverManagerDataSource.

Keep in mind that when this application is to be used in an enterprise container, the DataSource can be easily obtained from JNDI, just by making changes to the Spring configuration file. The Java code no longer needs to be edited.

The `driverClassName`, `url`, `username` and `password` have been defined as entries in a properties file. When configuring the `DataSource`, make sure you reference the values in the properties file (see note in *ApplicationContext.xml*).

The properties will automatically be read from the properties file and added to the configuration by the PropertyPlaceholderConfigurer, which has already been configured at the bottom of the configuration file.

Step 9: Inject the datasource into the DAO classes

Inject the datasource into the DAO classes, using setter injection. The only DAO class that does not yet require a datasource is the MemberDAO. When Spring Hibernate is covered, the mapping for this class will be done using Hibernate.

Step 10: Validate *ApplicationContext.xml*

Make sure your XML file is valid against it's DTD before you continue to run the client by saving the file and checking for errors.

Step 11: Run the TestClient

Prior to running this application, make sure that your database server is up and running. Run the TestClient as a Java application.

Step 12: Carefully inspect the output

When everything works correctly, you should see that several DVDs were found that contain the letter x in the title. The first DVD in the result list will be used to create a new Loan and insert a record into the database.

Step 13: Inspect the database

You can use the testing SQL script files to examine for changes being made to the database.

Exercise 9. USING HIBERNATE WITH SPRING (OPTIONAL)

Overview	
Spring.	In this exercise you will implement and test the
Objective	Use Spring's Hibernate classes.
Builds on Previous Labs	Standalone
Time to Complete	60 minutes

Overview of Steps

1. **Start an Eclipse project and inspect the classes**
2. **Inspect `springhibernate.model.MemberTO` and `Member.hbm.xml`**
3. **Complete `springhibernate.model.dao.hibernate.MemberDAOImpl`**
4. **Define the SessionFactory (`context.xml`)**
5. **Define the MemberDAO (`context.xml`)**
6. **Run the Test application**
7. **Define the transactionManager (`context.xml`)**
8. **Add the Transaction Advice (context.xml)**
9. **Define the AOP Auto Configurer (context.xml)**
10. **Run the Test application**

? Step 1: Start an Eclipse project and inspect the classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

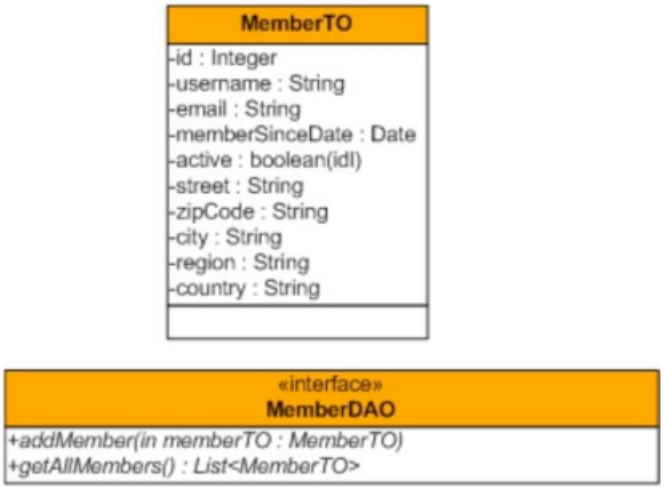
Now that you have completed that foundational setup, let's move on.

Create a new Java project called **SpringHibernate** . Add the **SPRI NG** and the **DERBY** user libraries to the build path. Since this lab is using Hibernate, several additional JAR files are required for both compilation and runtime. (In the case of Java projects, Eclipse makes sure that any JAR files that are visible for compilation are also visible for the runtime classloader.) All of these can be found under the `~StudentWork/Tools/` folder. At the project's property dialog box, select the **Libraries** tab and then repeatedly select the **Add External J ARs** button to add the following JAR files (note that the actual version may vary from what is listed):

- **hibernate/ hibernate3.jar**
- **hibernate/ lib/ required/ javassist-3.9.0.GA.jar**
- **hibernate/ lib/ required/ antlr-2.7.6.jar**
- all JAR files under the **dom4j-1.6.1** and **dom4j-1.6.1/ lib** folders
- **j2ee/ jta.jar** and **j2ee/ jsf-api.jar**

- **ehcache/ ehcache-core-1.7.2.jar**
- **cglib/ cglib-nodep-2.2.jar**
- **aspectj-1.6.8/ aspectjw eaver.jar**
- all JAR files under the **slf4j** folder

Finally, import the source code and resources from **~ / StudentW ork/ Labs/ Spring-Hibernate** into the project's **src** directory as explained in the **Configuring Dependencies** lab. Note that, in addition to the Spring context configuration file, you will import a Hibernate mapping file.



Step 2: Inspect `springhibernate.model.MemberTO` and `Member.hbm.xml`

The persistence layer of the MemberTO POJO will be implemented using Hibernate. The mapping file (*Member.hbm.xml*) has been created for you. This file defines the tables and columns in which the state of a MemberTO instance will be saved.

When looking at the *Member.hbm.xml* file, you should notice that the state of a single MemberTO will be separated over two database tables (MEMBER and ADDRESS). These tables will be created automatically by Hibernate during the first run of the application that attempts to access those tables in the database.

Step 3: Complete `springhibernate.model.dao.hibernate.MemberDAOImpl`

This class must implement the MemberDAO interface. Modify the class declaration to include the interface.

Before you proceed further, notice the instance variable named: *`HibernateTemplate`* . The value of this variable will be set using Dependency Injection by setting the sessionFactory property of this bean.

Implement the addMember() method. The HibernateTemplate provides you with utility methods which allow you to save or update an instance into a datasource.

Implement the getAllMembers method. The HibernateTemplate contains utility methods to find instances. However, the method needs an HQL query to be executed

against the datasource. In this case, the statement to be executed is " *From MemberTO".*

Step 4: Define the SessionFactory (context.xml)

You will have to start by defining the LocalSessionFactoryBean bean, which will represent the Hibernate SessionFactory.

Define a bean named *MySessionFactory* and class type *org.springframework.orm.hibernate3.LocalSessionFactoryBean.*

This bean requires two properties to be defined: *mappingResources* and *hibernateProperties* . The mapping resources are a list of *hbm.xml* files, containing the Hibernate mapping files. (*Member.hbm.xml*).

```
<property name= "mappingResources" >
  <list>
    <value>Member.hbm.xml</value>
  </list>
</property>
```

The second property (hibernateProperties) contains several properties which define the settings of the SessionFactory:

```
<props>
  <prop key="someKey">someValue</prop>
</props>
```

Define the properties according to the information below.

Property	Value
hibernate.connection.driver_ class	org.apache.derby.jdbc.ClientDriver
hibernate.connection.url	jdbc:derby://localhost/springclass
hibernate.connection.username	sa
hibernate.connection.password	password
hibernate.connection.pool_ size	2
hibernate.dialect	org.hibernate.dialect.DerbyDialect
hibernate.show_ sql	true
hibernate.hbm2ddl.auto	update

Step 5: Define the MemberDAO (context.xml)

Define a bean called MemberDAO that represents the implementation class you completed earlier.

Make sure you resolve the reference to the SessionFactory. (Remember the setSessionFactory method).

Step 6: Run the Test application

After ensuring that your database server is running, run the TestClient as a Java application.

The prebuilt testclient (springhibernate.test.TestClient) will insert a new Member into the database and then will display all available members in the database.

When executing your client, you should see output, similar to the output shown below (lines have been truncated):

```
Hibernate: insert into MEMBER (....
Hibernate: insert into ADDRESS (....
Member added with ID: 1
Hibernate: select memberto0_.MEMBER_ID as MEMBER1_, ....
Member{id=1, username='user1149846271843', email='u....
```

However, if you check the actual database using a testing script, you will see that no records were actually added to the database !

To complete this exercise, you will have to add transaction control to the application.

Step 7: Define a Transaction Manager (context.xml)

Create a bean called *myTxManager* that is an instance of `org.springframework.orm.hibernate3.HibernateTransactionManager`.

The transaction manager requires a single property named sessionFactory to be set. This property references the SessionFactory you defined earlier.

Step 8: Add the Transaction Advice (context.xml)

Applying transactions will be done by creating proxy objects. Because we do not want to make changes to the client applications, the client should be unaware of the fact that it is talking to a proxy. When the client does a lookup for *MemberDAO*, Spring will return a reference to a proxy.

Add a < *tx:advice*> element. Name it *txAdvice.*

It's *transaction-manager* should be the name of the transaction manager you created in the previous step.

Give the advice a *tx:attributes* element. That element should have two *tx:method* elements.

The first should specify that all methods named *add** should have their transaction propagation set to *REQUIRED.*

The second should specify that all other methods *, should have their transaction propagation set to *REQUIRED* and also set to *read-only to* true.

Step 9: Define the AOP Auto Configurer (context.xml)

Define an *aop:config* element.

Set its *aop:pointcut* **to** :

```
execution(* springhibernate.model.dao.MemberDAO.*(..))
```

Set its *aop:advisor* to refer to the transaction advice you created in previously, and to the pointcut you created earlier in this step.

Step 10: Run the Test application

The records will now be added to the database. Since the username is created using the currentTimeMillis method from the System class, you can run the client multiple times. You should see a member being added to the database each time you run the TestClient.

Exercise 10. USING STRUTS WITH SPRING (OPTIONAL)

Overview	
Struts web application.	In this exercise you will add the functionality of
Objective	Create a front-end using Struts and Spring
Builds on Previous Labs	Standalone
Time to Complete	75 minutes

Overview of Steps

1. **Inspect the classes**
2. **Configure the Derby Database**
3. **Create the Database tables**
4. **Complete `springstruts.controller.SearchDVDAction`**
5. **Complete `springstruts.model.DVDBO`**
6. **Inspect `springstruts.model.dao.hibernate.DVDDAOImpl`**
7. **Complete *web.xml***
8. **Complete *struts-config.xml***
9. **Complete *applicationContext.xml***
10. **Complete *action-servlet.xml***
11. **Deploy and Test the Web application**

Step 1: Inspect the classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Just about the time that you have figured out all of the nuisances of working with Spring and Hibernate with Java Projects and Applications, we shift to J2EE applications and, more specifically, web applications. There are several dimensions to the issues of developing and working with Spring and these other technologies within the IDE and servers.

First, turn to the tutorial at the end of labs to learn how to setup your IDE to work with your server.

Next, create a new Dynamic Web Project called **SpringStruts.**

Three of the challenging aspects of web applications are visibility of classes at compile time, visibility of classes at runtime, and the placement of resources into the proper location so that they get deployed into the proper place in the WAR file to be properly managed by the server and server environments.

Visibility of classes at compile time

This is a continuation of what we have been dealing with in previous labs relative to placing JAR files and libraries into a project's build path. For this lab, add the **SPRI NG** and the **DERBY** user libraries to the build path. Since this lab is using Hibernate and Struts, several additional JAR files are required for compilation. Most of these can be found under the `~StudentWork/Tools/` folder. At the project's property dialog box, select the **Libraries** tab and then repeatedly select the **Add External J ARs** button to add the following JAR or sets of JAR files:

- **hibernate/ hibernate3.jar**
- **hibernate/ lib/ required/ javassist-3.9.0.GA.jar**
- **hibernate/ lib/ required/ antlr-2.7.6.jar**
- **hibernate/ lib/ required/ jta-1.1.jar** (for Tomcat only)
- **dom4j-1.6.1/ dom4j-1.6.1.jar** (for Tomcat only)
- **j2ee/ jsf-api.jar**
- **struts/ struts-core-1.3.10.jar**
- **j2ee/ servlet-api.jar**

For Derby you must copy the derbyclient.jar file (mentioned in the earlier setup lab) to a specific location in your application server installation. For JBoss, this location is ${ JBOSS_ INSTALL} /server/default/lib and for Tomcat, this location is ${ TOMCAT} /lib.

Visibility of classes at runtime

This visibility is more complex than with Java applications because the J2EE application is getting deployed out into a web container (for example) that Eclipse has virtually no control over. In a container, there are, potentially, many different ways that classloaders can be configured. It is important to get the versions of classes that we want into the proper location so that other versions used by the container are not loaded prior to the environment even seeing them. Our goal here is get the JAR files deployed into the WEB-INF/lib directory in the WAR file that is generated by EclipseWTP. It turns out that the files that are needed at runtime are sometimes different than what we need at compile time and it also turns out that Eclipse WTP does not simply take the JAR files that are in the build path and automatically put them into the WEB-INF/lib directory of the generated WAR file. When we designate JAR files for deployment, we want to be very explicit about what we deliver to minimize the size of what is deployed as well as strictly control exactly what gets placed into the web container.

In the IDE, we add jars to the WEB-INF/lib folder by defining a Deployment Assembly. Right click on the SpringMVC web project and choose Properties. Select Deployment Assembly.

In the Deployment Assembly section use the add button to add items from the Java Build Path into this assembly. Add the following entries:

- **SPRI NG**
- **DERBY**
- **struts/ struts-core-1.3.10.jar**
- **hibernate/ hibernate3.jar**
- **hibernate/ lib/ required/ javassist-3.9.0.GA.jar**

- **hibernate/ lib/ required/ antlr-2.7.6.jar**
- **j2ee/ jsf-api.jar**

Repeatedly select **Add External J ARs** to add the following JAR files from the `~StudentWork/Tools/` folder:

- **ehcache/ ehcache-core-1.7.2.jar**
- **log4j-1.2.15.jar**
- **slf4j/ slf4j-api-1.5.0.jar**
- **slf4j/ slf4j-log4j12-1.5.0.jar**

After adding these to the list, ensure that each one is also check marked for deployment and select **Apply** and **OK.**

If using Tomcat or the STS's tc Server (which is based on Tomcat) or JBoss, do NOT add javaee.jar or servlet-api.jar to the Deployment Assembly folder. They will conflict with Tomcat libraries.

Placement of resources in proper location

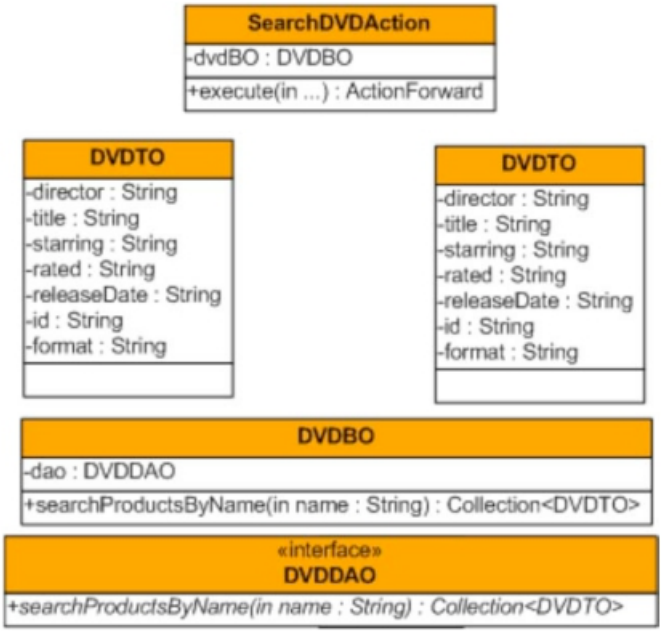
With all of the technologies that we are now deploying in our web applications, proper placement of configuration and mapping files is critical to successful deployment. Web containers are expecting specific files to be in specific places and, if they are not there, the application will probably not deploy successfully.

This is, of course, further complicated by Eclipse's somewhat obtuse import mechanism. A best practice is to always check the destination for any import to make absolutely sure that it is going where you expect it to. We try to make the import instructions as explicit as possible to assist you in this process.

Import the source code (**~ / StudentW ork/ Labs/ Spring-Struts/ W eb/ src**) into the project's **src** folder. Note that this also imports the Hibernate mapping file (**DVDTO.hbm.xml**) into the base directory of the **src** folder. Eventually, this will cause the mapping file to be deployed into the WAR file's WEB-INF/classes folder, which is where it needs to be.

Import the web resources

(**~ / StudentW ork/ Labs/ Spring-Struts/ W eb/ W ebContent**) into the project's **W ebContent** folder. This will place the servable resources (HTML and JSP files) into the WAR file's base folder. It will place (after you allow files to be overwritten) various configuration files into the WEB-INF folder (such as web.xml, applicationContext.xml, struts-config.xml, etc).



Step 2: Configure the Derby Database

You will need to modify the database to support the various view-oriented labs that are coming up. Run the **SpringStrutsJ SFCreateDB** command and then populate the new database table by running the **SpringStrutsJ SFPopulateDB** command. You can examine both of the associated scripts to see what was created. Running the **SpringBaseTestDV DTitlesDB** command will also show you what is in the database.

Step 3: Complete `springstruts.controller.SearchDVDAction`

This class has been implemented to function within a Struts application that does not rely on Spring, you will have to make the necessary changes to add the Spring functionality.

Make sure the class extends the appropriate Spring Support class, instead of the default Action class. Changing this will make this Action class aware of the WebApplicationContext.

Define an instance field of type DVDBO.

Create the appropriate setter method that allows Spring to use dependency injection to inject the business object into this action class before the Struts framework invokes the execute method.

IMPORTANT: Within the execute method you will find a comment saying 'TODO remove the next line!'. You will want to remove the line 'DVDBO dvdBO = new DVDBO();' from within the execute method, to make sure the execute method uses the business object injected by Spring instead of the mocked class that is used right now.

Step 4: Complete `springstruts.model.DVDBO`

You will change this class to use Springs dependency injection.

Declare an instance variable of type DVDDAO and provide the required setter methods that allow Spring to inject the appropriate dao class into this business object.

Remove the current implementation from the searchProductsByName method, instead use the dao class (injected by Spring) to perform the search and return the result.

Step 5: Inspect `springstruts.model.dao.hibernate.DVDDAOImpl`

No coding is required in this class

The DVDDAOImpl uses Springs HibernateTemplate class to simplify the use of hibernate within the DAO class. This template class is created within the setSessionFactory method that, in turn, is invoked by spring when injecting this class with a reference to Hibernates SessionFactory.

Step 6: Complete *web.xml*

You will have to register `org.springframework.web.context.ContextLoaderListener` as a

listener of the web application, to accomplish this, you will have to use the < listener> and the < listener-class> elements.

While you are here, note the name under which the ActionServlet has been registered. This name will be used in the Spring-Struts configuration file (in this case : `action-servlet.xml`) that you will be working with in a few minutes.

Step 7: Complete *struts-config.xml*

Your will have to register the `org.springframework.web.struts.DelegatingRequestProcessor` controller in the struts configuration file. This custom controller allows Spring to intercept a request to an action and inject the dependencies before invoking the execute method. You will need to use the `controller` tag with the `processorClass` attribute set to the Spring class listed above.

Also you will have to register the `org.springframework.web.struts.ContextLoaderPlugIn` plugin. This plugin is responsible for loading *action-servlet.xml*. You will need to use the `plug-in` tag with the `className` attribute set to the Spring plugin class listed above.

While you are here, note the action mapping in this file. This action mapping will have to be duplicated into *action-servlet.xml*. (you will be working on this file later).

Step 8: Complete *applicationContext.xml*

Remember, the *applicationContext.xml* defines the Spring mappings of the model and should be completely independent of the presentation layer used.

Take some time to inspect the bean definitions that are already present in the configuration file.

The `highviewDataSource` bean (of type `JndiObjectFactoryBean)` uses the JNDI name *`jdbc/HighView`* to lookup the datasource, which will later be used by the Hibernate SessionFactory.

Naturally, this datasource must also be defined in the *web.xml* (done for you) and the datasource must also be registered with the application server. This registration mechanism is application server-specific. In the case of JBoss, this is accomplished with a combination of a jboss-web.xml file in the WAR file's WEB-INF folder and a second configuration file that is placed into JBoss's folders. (Note that the jboss-web.xml file is ignored by other application servers such as Tomcat or WebSphere.)

You will find a file **spring-ds.xml** in the **~ / StudentW ork/ Labs/ Spring-Struts/ W eb** folder. This file defines the database-specific connection information for the Derby database that we have set up. Copy this file into `${JBOSS_ROOT}/server/default/deploy`. You will need to restart JBoss to get the server to pick up the new datasource, which will be available for this and subsequent labs.

The MySessionFactory bean (of type `org.springframework.orm.hibernate3.LocalSessionFactoryBean`) has been defined to configure the Hibernate sessions, which are used by the DAO layer of this exercise.

Define the *DVDDAO* bean (of type springstruts.model.dao.hibernate.DVDDAOImpl).
This bean needs a single reference to be resolved, called sessionFactory. Make sure this property

references the SessionFactory registered earlier.

Define the *DVDBusinessObject* bean (of type springstruts.model.DVDBO). This bean also needs a single reference to be resolved (called ' *dao*'). Make sure this property references the DAO bean declared above.

? Step 9: Complete *action-servlet.xml*

Remember that the filename of this configuration file depends on the name under which the ActionServlet has been registered in the *web.xml.*

Define the springstruts.controller.SearchDVDAction action class in this file.
Make sure the *name* of this class is identical to the *path*, under which it was registered in the `struts-config.xml`.

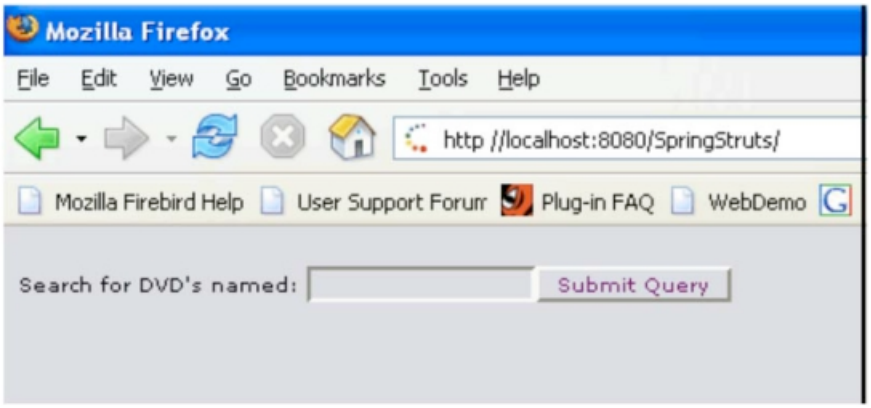
Resolve the reference of this action class (called dvdBO) and make sure this property references the business object you declared in the *applicationContext.xml* in the previous task.

? Step 10: Deploy and Test the Web application

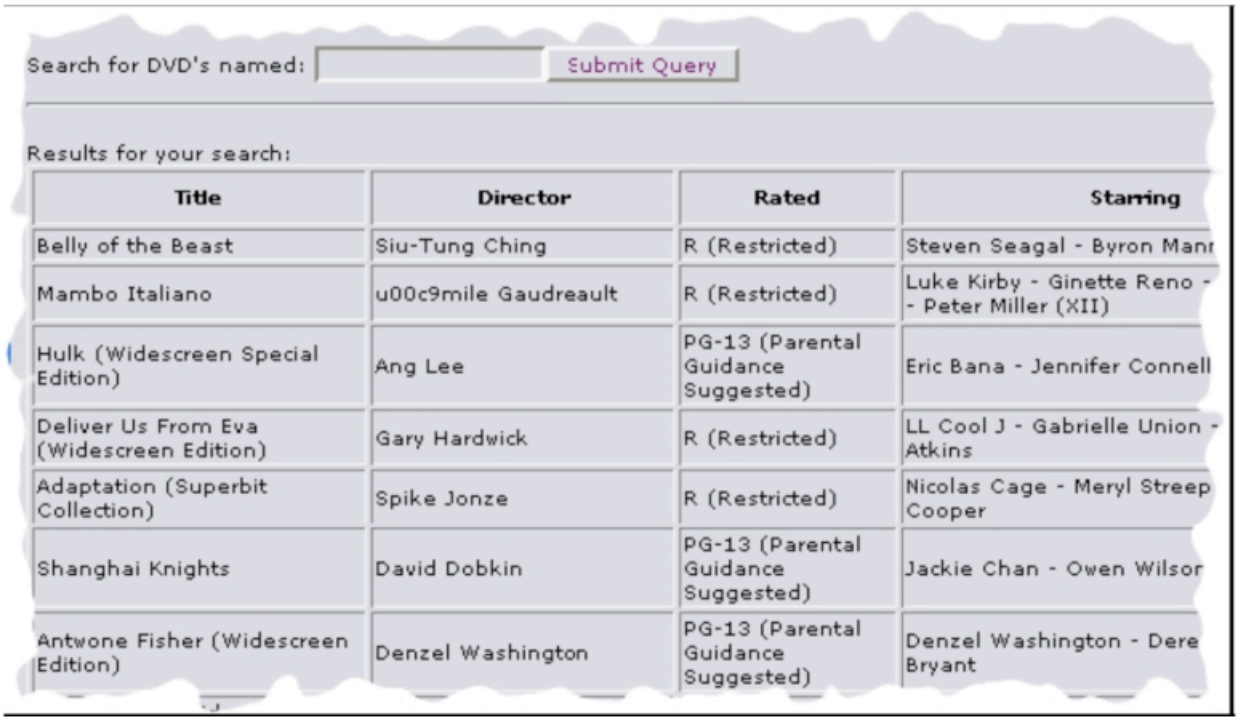
Deploy the **SpringStruts** project into your server. If everything is properly configured, you should see an indication that the application was deployed and is running. You can test it as described further down on this page. If you encountered deployment errors, there could be many sources for this, often caused by not getting the resources into the right location or the incorrect JAR files deployed out with the WAR file.

One debugging tool that can be extremely helpful now and in the future is to have examples of WAR files that have been successfully deployed. You can use these to compare what and how things are being deployed. For this and each of the subsequent J2EE labs, we have provided a working deployment package in the corresponding lab's solution folder. Feel free to use this in your debugging efforts.

Once deployed, start your web browser and navigate to http://localhost:[port]/SpringStruts, this should bring up the search page again.



This time you can perform an actual search against the database. For example enter just the letter 'a' and click the *Submit Query* button. The result page will display all DVDs that contain the letter 'a' anywhere in the title.



Title	Director	Rated	Starring
Belly of the Beast	Siu-Tung Ching	R (Restricted)	Steven Seagal - Byron Mann
Mambo Italiano	u00c9mile Gaudreault	R (Restricted)	Luke Kirby - Ginette Reno - - Peter Miller (XII)
Hulk (Widescreen Special Edition)	Ang Lee	PG-13 (Parental Guidance Suggested)	Eric Bana - Jennifer Connell
Deliver Us From Eva (Widescreen Edition)	Gary Hardwick	R (Restricted)	LL Cool J - Gabrielle Union - Atkins
Adaptation (Superbit Collection)	Spike Jonze	R (Restricted)	Nicolas Cage - Meryl Streep Cooper
Shanghai Knights	David Dobkin	PG-13 (Parental Guidance Suggested)	Jackie Chan - Owen Wilson
Antwone Fisher (Widescreen Edition)	Denzel Washington	PG-13 (Parental Guidance Suggested)	Denzel Washington - Dere Bryant

After you have exercised this implementation to your satisfaction, we suggest that you undeploy the project from your server to minimize its impact on future labs.

Exercise 11. USING SPRING MVC

Overview	
	To learn how to build a web application using S
Objective	Add functionality to list all the DVDs Add functionality to display details of a single DVD Add the Add DVD form
Builds on Previous Labs	Standalone
Time to Complete	75 minutes

Overview of Steps

1. **Inspect the Classes**
2. **Set up the Web Application**
3. **Add functionality to list all the DVDs**
4. **Add functionality to display details of a single DVD**
5. **Add the Add DVD Form**
6. **Deploy and Test**

Step 1: Inspect the Classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Create a new Dynamic Web Project called **SpringMVC.** Set the Target Runtime to your server.

Visibility of classes at compile time

For this lab, add the **SPRING** user library to the build path.

Add the jars within the **~/StudentWork/tools/JEE** folder to the Build path of the project

Visibility of classes at runtime time

In your IDE, we add jars to the WEB-INF/lib folder by defining a Deployment Assembly. Right click on the SpringMVC web project and choose Properties. Select Deployment Assembly.

In the Deployment Assembly section use the add button to add items from the Java Build Path into this assembly. Add the following entries:

SPRING (user library)

Jstl-impl.jar

Jstl.jar

Jsf-impl.jar

Jsf-api.jar

Jta.jar

Javax.jms.jar

If using Tomcat or the STS's tc Server (which is based on Tomcat), do NOT add javaee.jar or servlet-api.jar to the Deployment Assembly folder. They will conflict with Tomcat libraries.

Placement of resources in proper location

Import the source code (under **~ / StudentW ork/ Spring-MVC/ W eb/ src**) into the project's **src** folder. Eventually, this will cause the mapping file to be deployed into the WAR file's WEB-INF/classes folder, which is where it needs to be.

Import the web resources (under **~ / StudentW ork/ Spring-MVC/ W eb/ W ebContent**) into the project's **W ebContent** folder. This will place the servable resources (HTML and JSP files) into the WAR file's base folder. It will place (after you allow files to be overwritten) various configuration files into the WEB-INF folder (such as web.xml, spring-servlet.xml, etc).

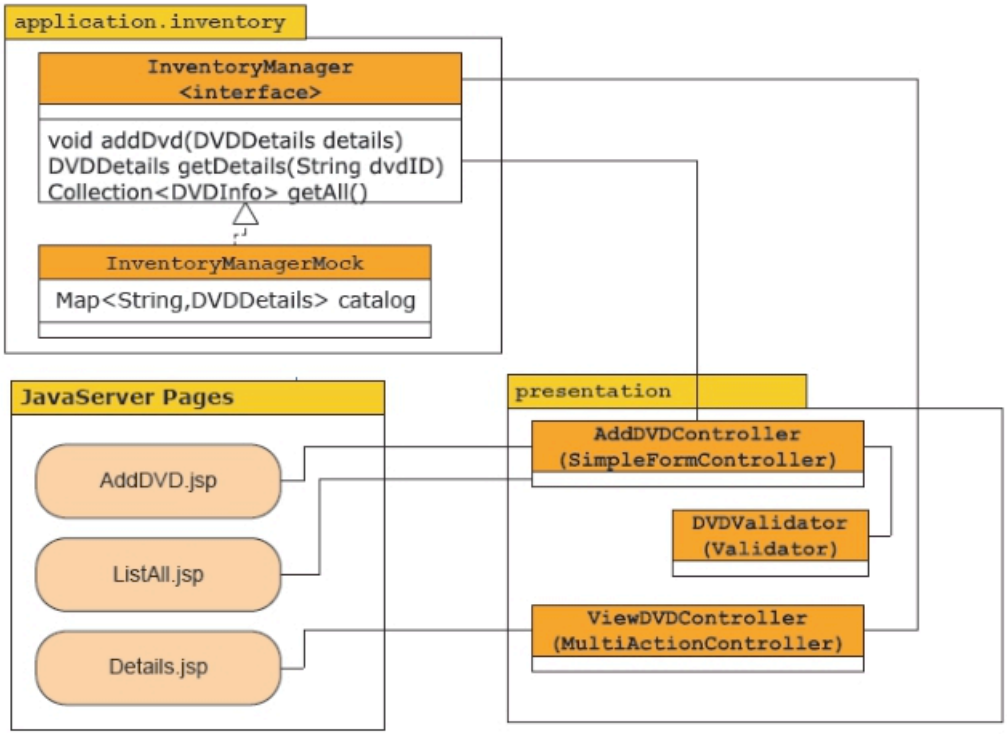
During this lab, you will be producing three views:

To view all DVDs

To see the details of a single DVD

Add a new DVD

Below you'll find a diagram depicting the key classes and JavaServer Pages used during this lab.



The business tier is exposed as an interface (InventoryManager) and is mocked by InventoryManagerMock. This implementation keeps an internal map of DVDs (using a catalog field). This catalog field is populated with some initial data using the Web Application Context file (see */WEB-INF/spring-servlet.xml*). This has all been done for you in advance. (You might want to take a look at this implementation, so you understand what is going on).

Observe that DVDs with id 100 and 101 already exist! This is handy when testing your application later.

Step 2: Set up the Web Application

The first step will be to setup the web application so that it is configured for Spring MVC. This task will involve three files:

The existing */WEB-INF/web.xml*

The Web Application context file (already exists and is named */WEB-INF/spring-servlet.xml*),

Open the *web.xml* and add the declaration for the DispatcherServlet (org.springframework.web.servlet package). You have to use the correct name. Normally you would be able to specify your own name, but in this exercise the name has been decided for you (due to the name of another file (Which file is it dependant on? Any Ideas??)). If you can't find this name, ask your instructor to give you some tips.

Next open the *spring-servlet.xml*. You will need to declare `a view resolver` bean in this file (ignore the other comments, these are for later tasks):

A ViewResolver

Add the ViewResolver of type org.springframework.web.servlet.view.InternalResourceViewResolver. This resolver requires a property prefix. Set this property to **/ view s/ inventory** which corresponds to the directory you imported in the **W ebContent** directory. This resolver requires a property suffix. Set this property to **.jsp** which will be added to the end of the view names that are returned from our controller methods.

That completes this task. Next you will focus on enabling the viewing of DVDs.

Step 3: Add functionality to list all the DVDs

During this task you will add support to list all DVDs. Spring 3 allows the traditional definition of controllers as beans, or the use of Annotations. Annotations are quickly becoming the preference.

First you need to configure Spring in order for your controller to be invoked. Open the *WEB-INF\spring-servlet.xml* and declare the Annotation tags (The context namespace has already been defined for you)

```
<mvc:annotation-driven />
<context:component-scan base-package="springmvc"/>
```

The ViewDVDController class has been provided. We need to add annotations to configure it as a controller. Add the @Controller annotation before the class declaration of ViewDVDController. Use @Autowired annotation above the declaration of the InventoryManager.

Next in the source code for the ViewDVDController add a viewAll method that displays all DVDs. This uses the InventoryManager to get all the DVDs and passes that on as a model to the view named `ListAll` (letting the ViewResolver take care of the rest). Remember the name of the model element as you will need this later. We will work with the name catalog for the model.

```
public ModelAndView viewAll() throws Exception {
   Collection<DVDInfo> all = manager.getAll();
   return new ModelAndView( "ListAll", "catalog", all);
}
```

The requests need to be routed to this new method. Add the `@RequestMapping(value="viewAll.view", method=RequestMethod.GET)` annotation immediately before the declaration of the viewAll method.
Next open the ListAll JSP. Iterate though the model element (catalog) using the forEach JSTL tag. For each dvdinfo add a link to *details.view* passing a request parameter dvdID with the value of the id, and display the title.

Step 4: Add functionality to display details of a single DVD

This task is very similar to the previous one. We will provide a little less guidance this time...

The request is made from the links you just added to the *ListAll.jsp* page. Add the appropriate method to the ViewDVDController class. It will use the InventoryManager to get all the details of DVD identified with the request parameter. Next it passes the result of that as a model to a named view (e.g., Detail). Again, make sure you remember the name of the model element, as you will need that when you create the JSP.

Do not forget the @ RequestMapping annotation before the method.

Open and complete the JSP (follow the comments in the code). You could deploy and test this part of the application (see steps in the Deploy and Test task below).

Step 5: Add the Add DVD Form

This task will add support for a Form (to add a DVD). You will be creating, declaring and mapping a new Controller (the AddDVDController).

Now you will focus on the controller implementation. Open the AddDVDController and navigate to the onSubmit method. Use the manager to add the DVD and return to the named view Detail. Add the @RequestMapping annotation to the onSubmit method.

Set the model so that the view can render the newly added DVD. The addDvd on the InventoryManager throws an InvalidDvdIdException when the supplied id is already in use. Catch this InvalidDvdIdException and return the user to the AddDVD view. Complete the JSP view (*AddDVD.jsp*). First declare a form using the correct action and method attribute values. For each property on the DVD (id, title, actors and releaseYear) add a row with a label and an input field bound (using the spring:bind tag) to the appropriate command property.

Step 6: Deploy and Test

Deploy and test the application.

After you have exercised this implementation to your satisfaction, we suggest that you undeploy the project from your server to minimize its impact on future labs.

Exercise 12. USING SPRING SECURITY

Overview	
	To learn how to add security to Spring Web pro
Objective	1. Create a base project with no security. 2. Secure web pages. 3. Work with multiple roles 4. Add logout capabilities 5. Add a custom login page 6. Use JSP Tags to conditionally include content based on role 7. Secure Services 8. Secure Individual Objects
Builds on Previous Labs	Uses the SpringMVC Solution as the starting point
Time to Complete	90 minutes

Overview of Steps

1. Create a base project w ith no security.
2. Secure w eb pages.
3. W ork w ith multiple roles
4. Add logout capabilities
5. Add a custom login page
6. Use J SP Tags to conditionally include content based on role
7. Secure Services
8. Secure I ndividual Objects

In this exercise, you will have a working application at the end of each step. Each step, will add one security feature. This project will work with any server including Tomcat, JBoss and others.

Step 1: Inspect the Classes

Prior to starting on this exercise, if you have done already done so, please work through the tutorial on setting up your IDE for this course. Take your time working through that tutorial, making sure you understand what you are being asked to do because you will be working with these tools and utilities throughout the remainder of the course.

Now that you have completed that foundational setup, let's move on.

Create a new Dynamic Web Application Project called **SpringSecurity.**

Visibility of classes at COMPI LE time

For this lab, add the **SPRI NG** user library to the build path.

Visibility of classes at RUN time

We add jars to the WEB-INF/lib folder by defining a Deployment Assembly. Right click on the **SpringSecurity** web project and choose Properties. Select Deployment Assembly.

In the Deployment Assembly section use the add button to add items from the Java Build Path into this assembly. Add the **SPRI NG** user library for deployment and then select **Apply** and **OK.**

Placement of resources in proper location

Import the source code (under **~ / StudentW ork/ Spring-Security/ W eb/ src**) into the project's **src** folder. Make sure to be in the Java perspective while importing these files.

Import the web resources

(under **~ / StudentW ork/ Spring-Security/ W eb/ W ebContent**) into the project's **W ebContent** folder. This will place the servable resources (HTML and JSP files) into the WAR file's base folder. It will place (after you allow files to be overwritten) various configuration files into the WEB-INF folder (such as web.xml, jboss-web.xml, dispatcher-servlet.xml, etc).

This lab will have the same essential features as the previous lab. Most of the changes will be to configuration files to add security features. There will be a minimal amount of Java and JSP coding.

The views that this application has are:

The initial web page.

To view all DVDs

To see the details of a single DVD

Add a new DVD

This application should already be working – as it is based on the solution to the Spring MVC lab.

Make sure that the application actually works and that you can access the various web pages before you go on.

? Step 2: Add Basic Security to the application.

In this step, you will add basic security to this application. The user will be required to login before accessing any of the functions. Only the initial page (index.html) won't require security to login to it.

Spring Security is a separate download from Spring. The instructor will let you know which directory it is in. In all likelihood, you will have to unzip the file **{ course-directory} / StudentW ork/ Tools/ spring-security-3.0.5.zip** (the version number might be different) **.** Call the directory where you unzip this file to: **{ SPRI NG-SECURI TY }** .

This step will involve editing three files, plus adding all the JAR files that you will need:

1. The existing */WEB-INF/web.xml – you will add security filters to it.*
2. ***W EB-I NF/ applicationContext-security.xml*** - A Spring Beans configuration file for this project. You will add a Spring description of the core security that this project requires.
3. ***W EB-I NF/ users.properties*** – A file that will contain a list of user accounts. In a real project, of course, a database or other authentication mechanism will contain the account details.

? List of steps

Substep A: Add the following jar files to both the build path and Deployment Assembly: (The version numbers might be different.)

{ SPRI NG-SECURI TY } / dist/ spring-security-core-3.0.5.jar
{ SPRI NG-SECURI TY } / dist/ spring-security-w eb-3.0.5.jar
{ SPRI NG-SECURI TY } / dist/ spring-security-config-3.0.5.jar
{ SPRI NG-SECURI TY } / dist/ spring-security-taglibs-3.0.5.jar

Substep B: Add the following code to the *web.xml* file, right after the **< display-name> … < / display-name>** element. This will add the needed filters to your web application to intercept all calls to web pages and make sure that the viewer only sees them if they are authorized to.

```
< context-param>
      < param-name> contextConfigLocation< /param-name>
      < param-value>
            /WEB-INF/applicationContext-security.xml
      < /param-value>
< /context-param>

< filter>
      < filter-name> springSecurityFilterChain< /filter-name>
      < filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
      < /filter-class>
< /filter>

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

< !--
      - Loads the root application context of this web app at startup.
      - The application context is then available via
      - WebApplicationContextUtils.getWebApplicationContext(servletContext).
-->
< listener>
      < listener-class>
            org.springframework.web.context.ContextLoaderListener
      < /listener-class>
< /listener>

< !--
      - Publishes events for session creation and destruction through the application
      - context. Optional unless concurrent session control is being used.
-->
< listener>
      < listener-class>
            org.springframework.security.web.session.HttpSessionEventPublisher
      < /listener-class>
< /listener>
```

Substep C: Edit the file: ***W EB-I NF/ applicationContext-security.xml.*** Notice that it's default namespace is set to *http://www.springframework.org/schema/security* . *This means that you won't have to add a namespace qualifier to each of the elements below – as they will default to coming from the Spring Security XML schema. Add the following to this file, between the root < beans:beans> element and its closing < /beans:beans> tag.*

```
< http auto-config= "false">
        < intercept-url pattern= "/inventory/*" access= "ROLE_ USER" />
        < intercept-url pattern= "/**"
                access= "IS_ AUTHENTICATED_ ANONYMOUSLY" />
        < form-login />
        < anonymous />
        < remember-me />
        < logout />
        < remember-me />
< /http>

< authentication-manager alias= "authManager">
        < authentication-provider>
                < user-service id= "userDetailsService"
                        properties= "WEB-INF/users.properties" />
        < /authentication-provider>
< /authentication-manager>
```

This file says that we are adding http security. The main restriction is that all URLs in this project that start with /inventory are restricted to only people who are logged in and have the ROLE_ USER permissions.

The list of accounts will be in the *users.properties* file.

Substep D: Create the file: ***W EB-I NF/ users.properties.*** Give it the following content.

```
robert=silver,ROLE_USER,enabled
```

This means that you are creating one account. The user name is "robert". The password is "silver". When the person logs in they will have the privileges of **ROLE_ USER.**

Step E: Restart the application and try it. The index.html page will show. Select " View all DVDs ". The application will generate a login page.

First, login with the incorrect credentials (for example, a user name of "michael" and a password of "gold"). The system will not let you access the web page. Now, login in as robert with a password of silver. The application will work.

We will add logout capabilities later. For now, you can logout by going to the URL: http://localhost:8080/SpringSecurity/j_spring_security_logout . (assuming that you are running the server at port 8080).

Congratulations. You have now added basic security to your application. Notice, that very little work was required. Also notice that you did not have to make any changes to your web pages or to code any Java code.

? Step 3: Create multiple roles/authorizations. Have different pages require differing permissions.

In this step, you will create multiple accounts and multiple roles. We will setup security, so that any body with the permissions of ROLE_ USER will be able to view the existing DVDs. However, you will need the permissions of ROLE_ MANAGER to add a new DVD.

To help do this easily, you will split the JSP files of the application to two directories. You will place *Detail.jsp* and *ListAll.jsp* in a directory called *customers* . You will place *AddDVD.jsp* into a directory called *managers* .

To do this, you will modify the following files.

A: **W EB-I NF/ users.properties** – to add new accounts and roles.
B: ***W EB-I NF/ applicationContext-security.xml*** – You will restrict some pages to only Managers.
C: Create two subdirectories for the JSP files and reposition the JSP files in those directories.
D: **index.html** – to change the URLs.
E: **W EB-I NF/ dispatcher-servlet.xml** – to change the URLs that Spring will respond to.
F: **src/ inventory.properties** – to change the location where Spring MVC will look for the JSP web pages.

Substep A : Add the following line to WEB-INF/users.properties.

```
michael=gold,ROLE_USER,ROLE_MANAGER,enabled
```

Substep B: Modify the **applicationContext-security.xml** file.

Replace the line:

```
<intercept-url pattern="/inventory/*" access="ROLE_USER" />
```

with

```
<intercept-url pattern="/customers/*"
                   access="ROLE_USER, ROLE_MANAGER" />
<intercept-url pattern="/managers/*" access="ROLE_MANAGER" />
```

This allows both USER and MANAGER authorization to access the *customers* pages, while allowing only MANAGERs to access the *managers* pages.

Substep C: Create two subdirectories for the JSP files.

i. Create two subdirectories *WEB-INF/views/customers* and *WEB-INF/views/managers* .
ii. Place *WEB-INF/views/inventory/ListAll.jsp* and *WEB-INF/views/inventory/Detail.jsp* into the *customers* directory.

iii. Place *WEB-INF/views/inventory/AddDVD.jsp* into the *managers* directory.
iv. Delete the *WEB-INF/views/inventory* directory.

Substep D: Modify index.html – so that the two links go to *customers/viewAll.view* and *managers/AddDVD.form* respectively.

Substep E: Modify WEB-INF/dispatcher-servlet.xml.

In the bean named *handlerMapping* , in the property named *mappings* , there are two properties. Modify the keys of these two properties to be the correct names of the two URLs as shown below.

```
<prop key="/managers/AddDVD.form">addDVDController</prop>
<prop key="/customers/*.view">viewDVDController</prop>
```

Substep F: Modify the **src/ inventory.properties** file. Modify the URLs to be the correct directory for each of the 3 jsp files. For example, the second line of the file should be modified to:

```
DVDlist.url=/WEB-INF/views/customers/ListAll.jsp
```

Modify the other 2 url references to go to *customers* and *managers* respectively.

Substep G: Test the application. Try adding DVDs as Robert. This system will give you a 403 error. Logout and try adding DVDs as Michael. This time, you will have sufficient permissions and you should be able to add new DVDs as well as view them.

? Step 4: Add a way to logout of the application.

In this step, you will add a *logout* button to each of the three JSP web pages.

Add the following code to the 3 JSP files in the ***W EB-I NF/ view s/ customers*** and ***W EB-I NF/ view s/ managers*** directories. Add this code right before the < */body*> tag.

```
<hr />
<form action='<c:url value="/j_spring_security_logout"/>'
            method='get'>
      <input type='submit' name='logout' value='Logout' />
</form>
```

This will add a Logout button to each of the 3 pages.

Also, make sure that each of these JSP files have the line:

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

Test the application and make sure that logging out works correctly. After logging out, if you try to invoke any of the functions of the application you will have to login again.

Step 5: Add a custom login page.

Spring Security generated a login page as needed. In this step, you will create your own custom login web page.

Substep A : Create a file **login.jsp** in the WebContent directory. Run the application and when the Spring generated login page comes, do "view source" from the browser and cut and paste the content into login.jsp. This will serve as starting point code for your **login.jsp** file.

Edit login.jsp to be the following code. Feel free to change or embellish the look and feel. The only critical part is the names of the form elements and where the form posts to. You will also want error messages to display as shown.

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

<html> <head>  <title>CUSTOM SPRING SECURITY LOGIN</title> </head>

  <body onload="document.f.j_username.focus();">
  <h1>CUSTOM SPRING SECURITY LOGIN</h1>
   <P>Valid users:  <P>
   <P>username <b>michael</b>, password <b>gold</b>
   <P>username <b>robert</b>, password <b>silver</b>
   <p>

  <%-- this form-login-page form is also used as the
        form-error-page to ask for a login again.          --%>
 <c:if test="${ not empty param.login_error}">
     <font color="red">
       Your login attempt was not successful, try again.<br/><br/>
       Reason: <c:out value="${SPRING_SECURITY_LAST_EXCEPTION.message}"/>.
     </font>
 </c:if>

 <form name="f"
           action="<c:url value='j_spring_security_check'/>" method="POST">
     <table>
       <tr><td>User:</td><td>
           <input type='text' name='j_username'
                 value='<c:if test="${ not empty param.login_error}">
           <c:out value="${SPRING_SECURITY_LAST_USERNAME}"/></c:if>'/>
       </td></tr>

       <tr><td>Password:</td><td>
           <input type='password' name='j_password'></td></tr>
       <tr><td>
           <input type="checkbox" name="_spring_security_remember_me"></td>
           <td>Don't ask for my password for two weeks</td></tr>

       <tr><td colspan='2'>
           <input name="submit" type="submit" value="LOGIN"></td></tr>
     </table>
 </form>
</body>
</html>
```

Substep B: Modify the **applicationContext-security.xml** file.
Replace the line < form-login /> with

```
<form-login login-page="/login.jsp"
      authentication-failure-url="/login.jsp?login_error=1"/>
```

Substep C: Test the application to make sure that your custom login page is shown.

? Step 6: Use JSP Security tags to show some content to only some users.

In this step, we will add a link in the **ListAll.jsp** (and **Detail.jsp**) file to the *AddDVD.jsp* file. However, we want this link to show up only if you have ROLE_ MANAGER authority. If you only have ROLE_ USER authority, then the link should not show up.

Substep A: Add the following jar file to **W EB-I NF/ lib**

{ SPRI NG-SECURI TY } / dist/ spring-security-taglibs-2.0.3.jar

Substep B: Add the following line near the beginning of the JSP file. You can add it right after the other taglib directive in the file.

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

Modify the ***view s/ customers/ ListAll.jsp*** file, by adding the following right before the < /body> element.

```
<security:authorize ifAnyGranted="ROLE_MANAGER">
      <hr />
      <a href='<c:url value="/managers/AddDVD.form"/>'>
            Add a DVD</a>
</security:authorize>
```

Make the same changes to **view s/ customers/ Detail.jsp.**

Substep C: Test the application. Login as michael/gold and go to the *View all DVDs* page. Notice that the page gives you a link to the *AddDVD* page. Click on that link and see that it takes you to the Add DVD page. Logout.

Log back in as robert/silver. Again, go to the *View all DVDs* page. Notice that the page does NOT give you a link to the *AddDVD* page – because you do not have ROLE_ MANAGER authority when you are logged in as robert.

Step 7: In this step, you will secure the services themselves.

For this application, there is no real need to do this, as there is no way to access the services other than through the already secured web pages. However, it is always a good idea to place security on the business layer itself – as a back stop in case there are security flaws elsewhere.

In addition, this approach can also be used in now web applications, such as Swing based applications.

In this step, you will modify the file **dispatcher-servlet.xml** . This file has the Spring declaration of all the business objects beans. All the steps below are modifications to **dispatcher-servlet.xml** .

Substep A: Change the starting < beans> element to also have the schema location and namespace for the Spring security features. Below, we have underlined the text that has to be added.

```
<beans xmlns=http://www.springframework.org/schema/beans

     xmlns:security=http://www.springframework.org/schema/security

     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

     xsi:schemaLocation="

http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.0.4.xsd

http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

Substep B: In this step, you will protect the service beans. You will do this by protecting the *InventoryManager* interface.

Notice, that for now, we have put an **access= ” ROLE_ SUPER”** . This is a non-existent role. We are using it to force the security to fail. This will demonstrate that security is actually being applied.

The credentials and authorities you use for the web-access are also available to the business tier. That is because the business-tier methods are being executed in the same thread as the web-code.

Add the following, as the first element within the < beans> element.

```
<global-method-security
      xmlns="http://www.springframework.org/schema/security">
      <protect-pointcut
            expression=
"execution(* springmvc.application.inventory.InventoryManager.addDvd(..))"
            access="ROLE_SUPER" />
</global-method-security>


<security:authentication-manager alias="authManager">
      <security:authentication-provider>
            <security:user-service id="userDetailsService"
                  properties="WEB-INF/users.properties" />
      </security:authentication-provider>
</security:authentication-manager>
```

The pointcut expression was explained in the AOP part of this course. It means that access to the *addDvd()* method in objects that implement the InventoryManager interface should only be granted if you are logged in and have ROLE_ SUPER authority. Currently, nobody has that authority.

Substep C: Add the following jar files to both the build path and Deployment Assembly:

com.springsource.org.aspectj.tools-1.6.6.RELEASE.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

Navigate to the **~ / StudentW ork/ Tools/ SpringDependencies** folder and add the needed JAR files to your build path.

Substep D: Restart the application, and try adding a DVD. Login as michael/gold. Notice that you will be shown the web page form to add a DVD. However, when you submit the form, you will be shown a 403 error page, because the security system did not allow you to invoke that *addDvd()* method.

Substep E: Change the access restriction from ROLE_ SUPER to ROLE_ MANAGER.

Substep F: Restart the application, and try adding a DVD. This time, you should be able to add a new DVD when you are logged in as Michael. You have now successfully provided protection to objects and their methods, using AspectJ pointcuts. Spring has created proxies for the inventoryManager bean, described later in the **dispatcher-servlet.xml** file.

Step 8: Secure some features based on the user who is logged in, rather than their role.

Until now, all the permissions that were granted were based on the role/authorities that the logged in user has. They were not based on their individual identity. In some situations, you would like permissions to be based on the individual identity. For example, two customers might be logged in with the same ROLE. You will want to make sure that they can not see each other's accounts even though they have the same ROLE.

In this step, you will modify the Java code for the inventory manager implementation. We will make the (admittedly contrived) restriction, that when adding a new DVD, that the list of Actors, must contain the name of the person who is logged in. For example, if you are logged in as Michael, then the text of the Actors, must contain the string "michael". Otherwise, an Exception will be thrown, which will result in a 403 page being shown.

Substep A: Modify the addDvd() method of the class
springmvc.application.inventory.impl.I nventoryManagerMock

The code to add is shown below. This code gets the credentials of the person who is logged in and makes sure that

```
public void addDvd(DVDDetails details) throws InvalidDvdIdException{

     /* NEW CODE */
     Object authority =
           SecurityContextHolder. getContext()
                 .getAuthentication().getPrincipal();

     String username;
     if (authority instanceof UserDetails) {
         username = ((UserDetails)authority).getUsername().toLowerCase();
         String actors = details.getActors().toLowerCase();
         if (!actors.contains(username))
           throw new AccessDeniedException
                 ("You are not one of the Actors in this DVD.");
     } else {
           throw new AccessDeniedException
                 ("You are not logged in.");
     }
     /* End of NEW CODE */

     String id = details.getId();
     if (catalog.containsKey(id))
           throw new InvalidDvdIdException(id + " is in use");
     catalog.put(id,details);
}
```

Substep B: Make sure that Eclipse has added the following import statements.

```
import org.springframework.security.access.AccessDeniedException;
```

```
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
```

Substep C: Redeploy the application. Add a new DVD. Login as michael. Try adding an Actors list that does not contain the string michael. Notice that you will not be able to. When you try to submit the new DVD, you will get a 403 error page with the message given by the exception.

Try again, this time giving an actors list that contains the String michael. Notice that you will be successful.

You have now applied a lot of security to your application. Notice that other than the last step, all the changes were to XML files and properties files. Of course, in real-applications, you may have to write custom Java code, for example to represent UserDetails with additional information that your enterprise stores for each user.

Still, most of the configuration for security can be done declaratively.

Exercise 13. USING JMS WITH SPRING

Overview	
order.	In this exercise you will create two JMS client a JMS (Queue) producer. The producer will send
Objective	Become familiar with the JMS support in Spring.
Builds on Previous Labs	Standalone Application – The Configuration of ActiveMQ is from the prior lab
Time to Complete	60 minutes

Overview of Steps

1. **Configure and start Apache ActiveMQ**
2. **Setup the ActiveMQ library**
3. **Inspect the Classes**
4. **Implement the Producer**
5. **Implement the Consumer**
6. **Complete *context.xml***
7. **Test your work**

Step 1: Configure and start Apache ActiveMQ

If you have already configured ActiveMQ, you can go to the next step.

ActiveMQ is an open-source JMS solution that is becoming very popular. The ActiveMQ distribution can be found at

~ / StudentW ork/ Tools/ apache-activemq-5.3.2

ActiveMQ can be configured with Queues and Topics through the editing of configuration files or by using a dynamic mechanism. Any JNDI name lookup for a destination beginning with either dynamicQueues or dynamicTopics will create the new destination dynamically.

In the ActiveMQ folder switch to the **bin** folder and run the **activemq.bat** script to start our JMS server.

Step 2: Setup the ActiveMQ jars library

If you have already configured the **ACTI VEMQ** library, you can go to the next step.

Create a new Library named **ACTI VEMQ** with the following jars that are included in the ActiveMQ Distribution:

active-mq-all-5.3.2.jar

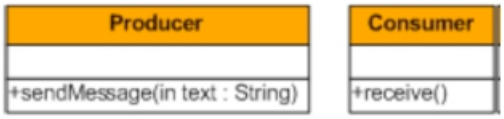
lib/ optional/ activemq-pool-5.3.2.jar

Step 2: Inspect the Classes

For this lab, we will use a Java project, although we will be using it as a JEE client, which involves bringing in a variety of extra JAR files.

Create a new Java project called **SpringJ MS** and add the **SPRI NG and ACTI VEMQ** user library to the build path

Import source code and resources from **~ / StudentW ork/ Labs/ Spring-J MS** into the project. At this time, examine the two starter classes.



Step 3: Implement the Producer

Populate the class that implements the springjms.producer.Producer interface.

The JMS ConnectionFactory and the Destination to which the message is to be sent will be provided to this class using Dependency Injection. You must provide the instance variables for these two properties and their appropriate accessor methods.

Within the sendMessage method, use an instance of JmsTemplate to send the given text to the destination.

Step 4: Implement the Consumer

Populate the class that implements the springjms.consumer.Consumer interface.

The JMS ConnectionFactory and the Destination from which the messages will be provided to this class using Dependency Injection. You must provide the instance variables for these two properties and their appropriate accessor methods.

Within the receive method, use an instance of JmsTemplate to receive messages from the given destination. Before you start receiving messages, keep in mind that by default, the thread will block until a message arrives. You might want to set a certain timeout before you start receiving messages.

Once an object has been received, print a message to the console containing the text message that was sent.

Step 5: Complete *context.xml*

You will have to provide all of the configuration for both the producer and the consumer.

- Obtain the ConnectionFactory from ActiveMQ.
- Obtain the Destination from JNDI (The JNDI name is: dynamicQueues/AMP_ Q)
- Define the Producer, make sure the bean is called producer (This bean name is used by the pre-built test application)
- Define the Consumer, make sure the bean is called consumer (This bean name is used by the pre-built test application)

You might notice that the definition of the Consumer is very similar to the definition of the Producer, to avoid typos, you might want to define a `JmsTemplate` bean (abstract !!!) and use that definition as parent definition for both the `producer` and `consumer` bean.

We will be using ActiveMQ Dynamic Queues like in the last lab.

Step 6: Test your work

Run the consumer's TestConsumer as a Java application

Once the consumer has successfully started, run the producer's TestProducer as a Java application. As the two JMS processes start and stop messages should be sent and acknowledged.

TUTORIAL : WORKING WITH KEPLER (JEE VERSION) AND TOMCAT 7.0

Overview	
In this lab exercise, you will: ? Tomcat instance. ? on a Tomcat instance. ? server itself. By the end of the lab you should be able to: ? ?	Setup a Dynamic Web Project, create a resourc Setup a Dynamic Web Project, import some res Work with the Tomcat instance from outside of Setup a web project and deploy it to an instanc Verify that Tomcat is operating and determine i
Objectives	To learn how to use Kepler with Tomcat.
Builds on Previous Labs	None
Time to Complete	60 minutes

Task List

Preface

During this training, you will be developing several mini-projects, using Juno and Tomcat. For each exercise, we have created skeleton code, which you will be editing. In some cases, we also provide additional files, which already have been implemented for you. For example, helper classes and some welcome pages for the web applications may be provided for you.

Before each exercise, all of these files (skeleton code, helper classes, etc…) will have to be imported into Kepler. This tutorial describes how to work with various aspects of Kepler *(JEE Version)* as well as Tomcat.

The W orkshop directory

For the location of the Workshop directory, ask your trainer. During this tutorial, we will presume that the Workshop directory has been placed in C:\StudentWork

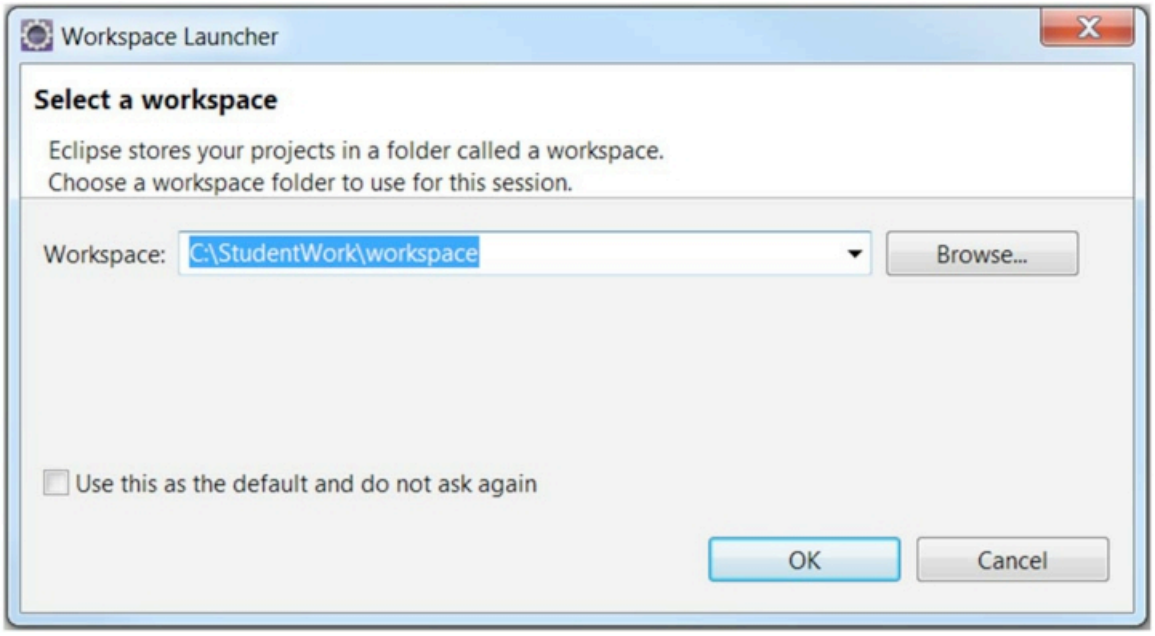
Starting Eclipse

1. Navigate to the home installation directory for Eclipse (the default is **C:/eclipse**) and double click on the Eclipse executable (**eclipse.exe**)

   A popup will appear in which you can specify the location of the workspace.

   Change the location of the workspace to:

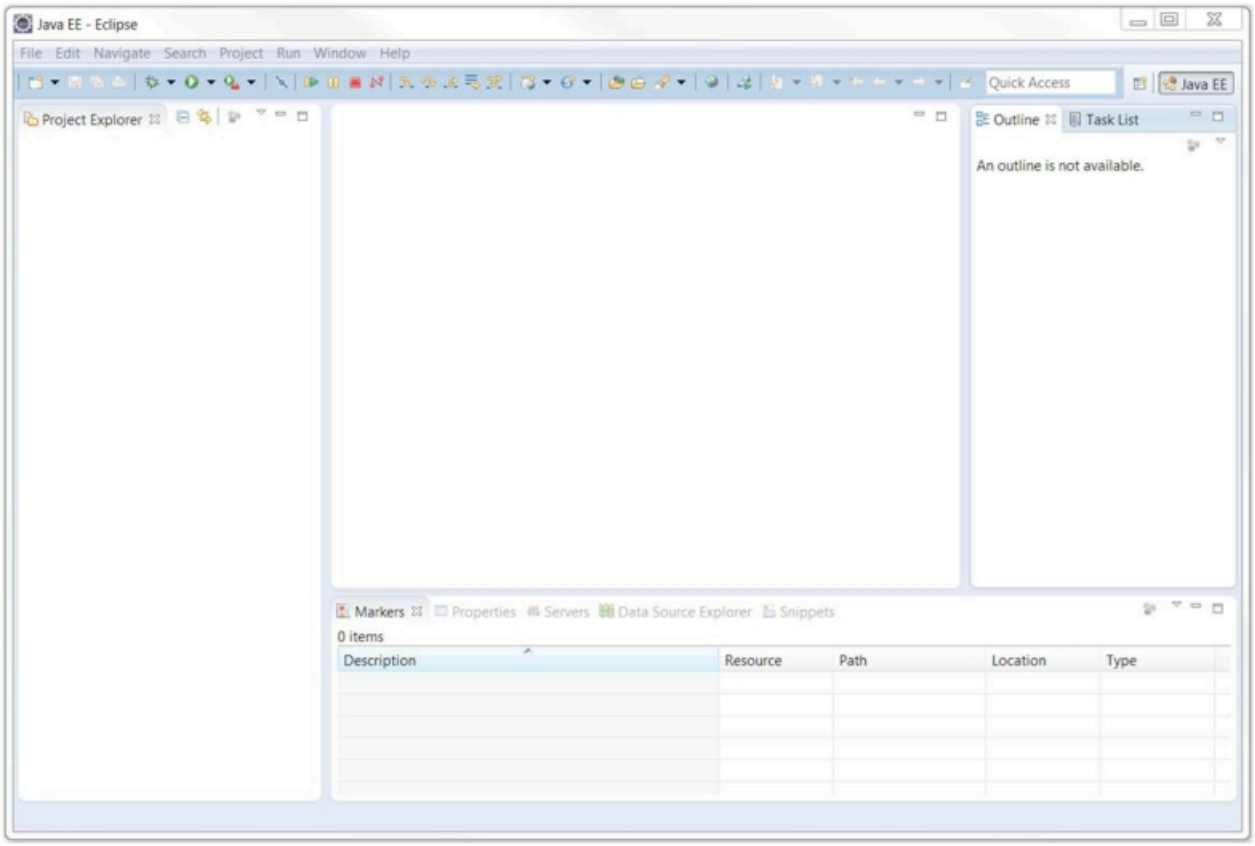
   C:\StudentWork\workspace



An empty workspace will now be created in the workspace directory.

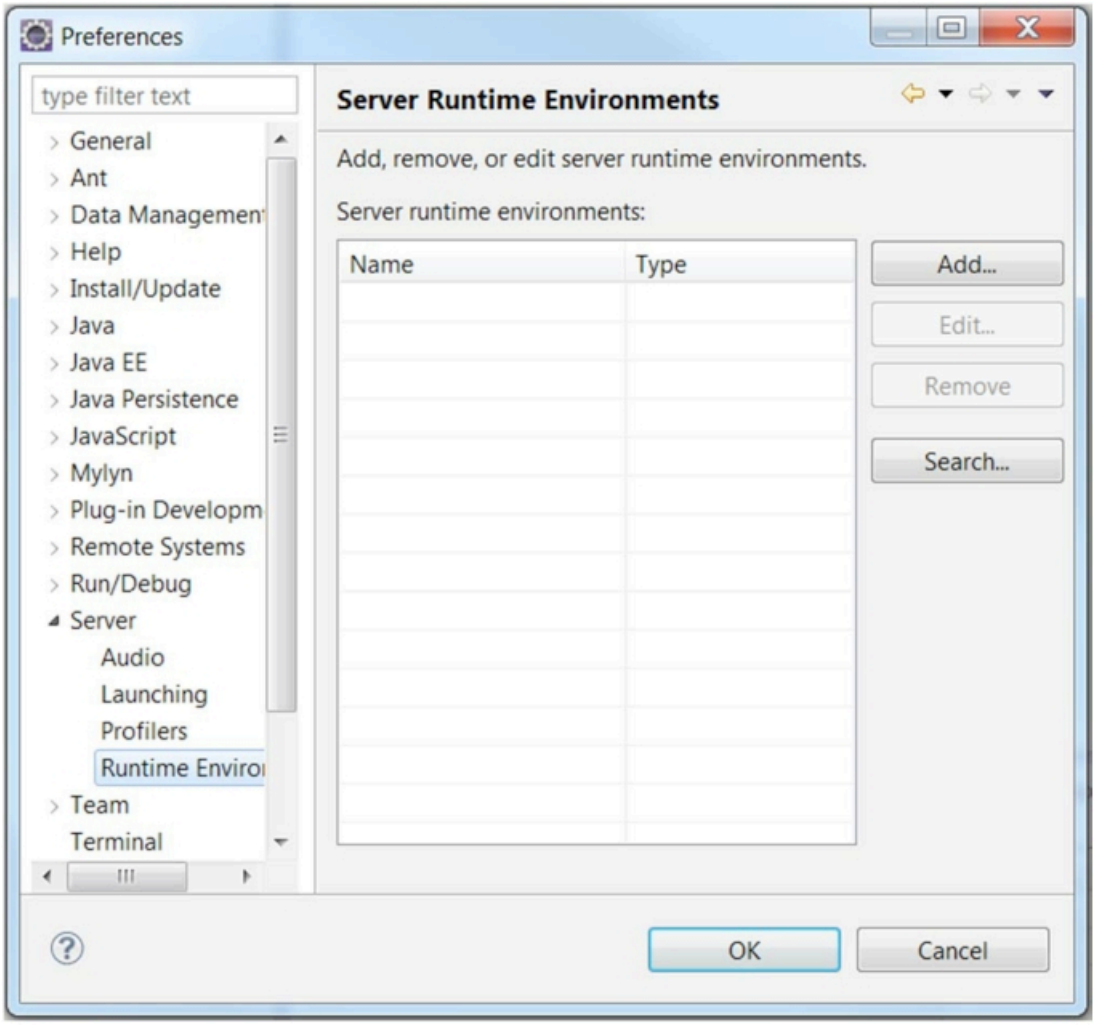
When the application is started, the Welcome page will be presented. Close the Welcome page.

You will see the workbench, pictured below:

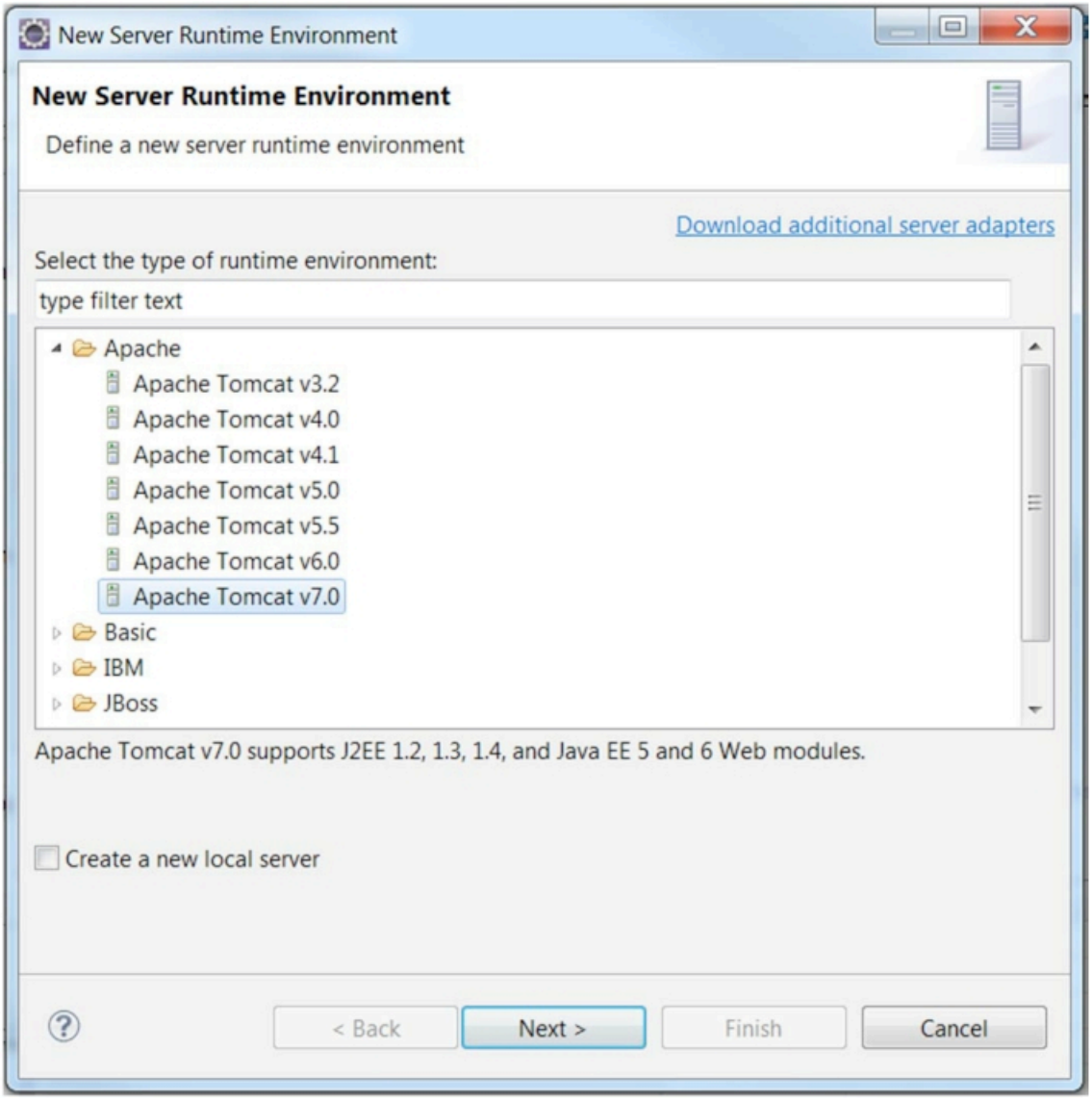


Configuring Kepler (JEE Version) for Tomcat 7.0

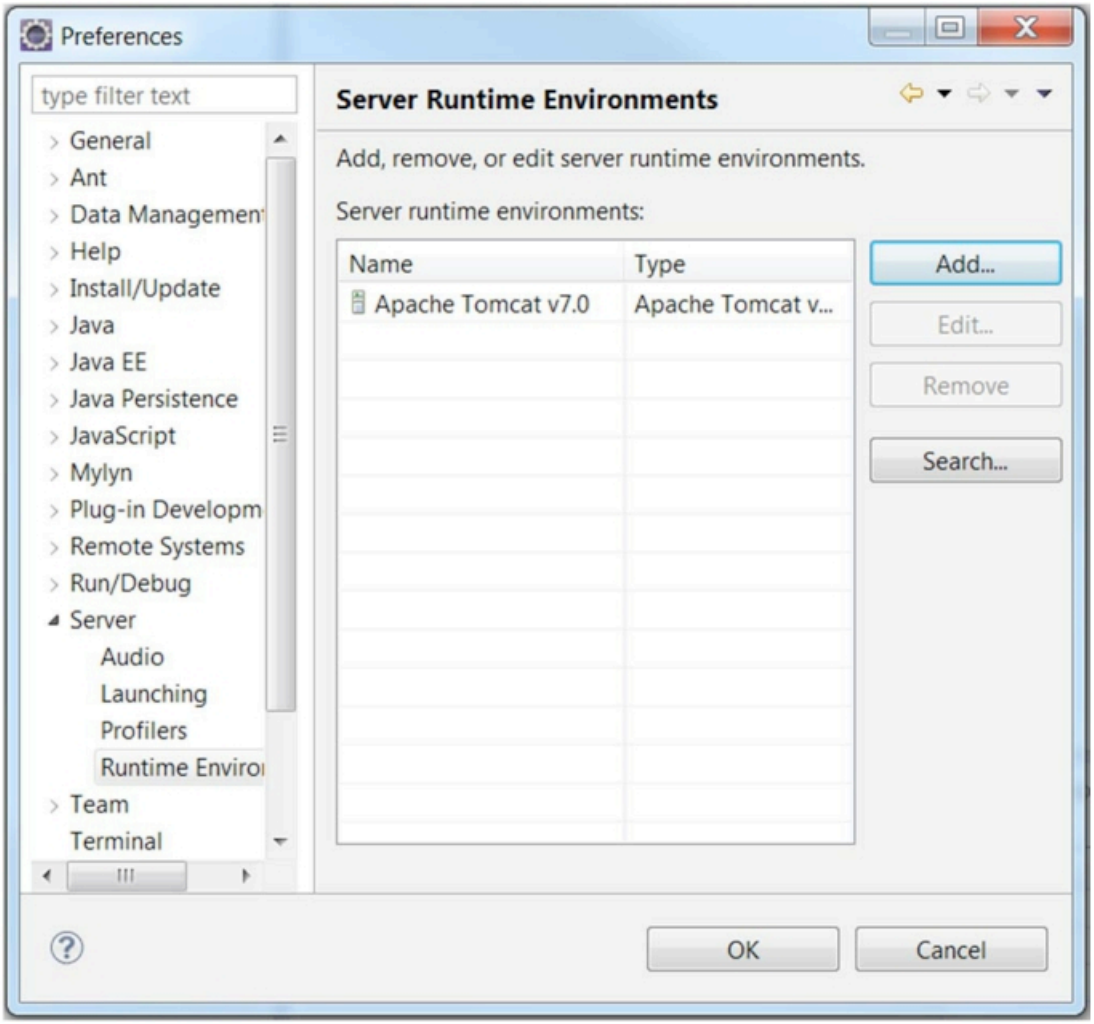
1. Eclipse needs to be configured to support whatever web server is being used in this class. The web server is already installed for your use. The following instructions are specific to **Tomcat 7.0** .
2. The app server should already be installed on your local machine.
3. Start Eclipse if it is not already running.
4. From the main menu select **Window** → **Preferences** . In the tree view to the left select **Servers** → **Server Runtime Environments** .



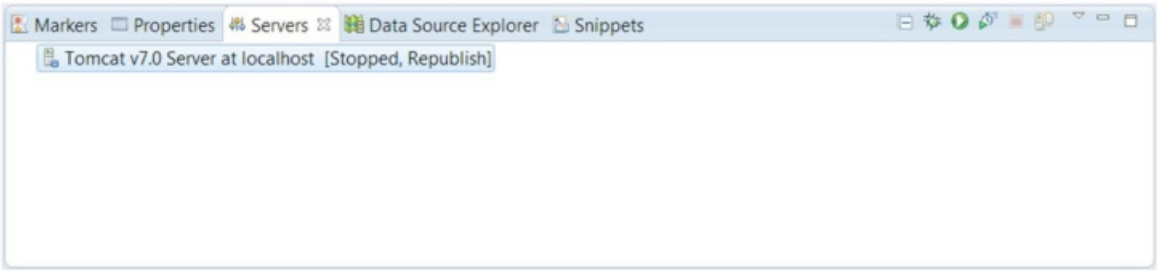
5. There should be no entries available. Click **Add** .
6. When the New Server Runtime window opens select **Apache → Apache Tomcat 7.0** .



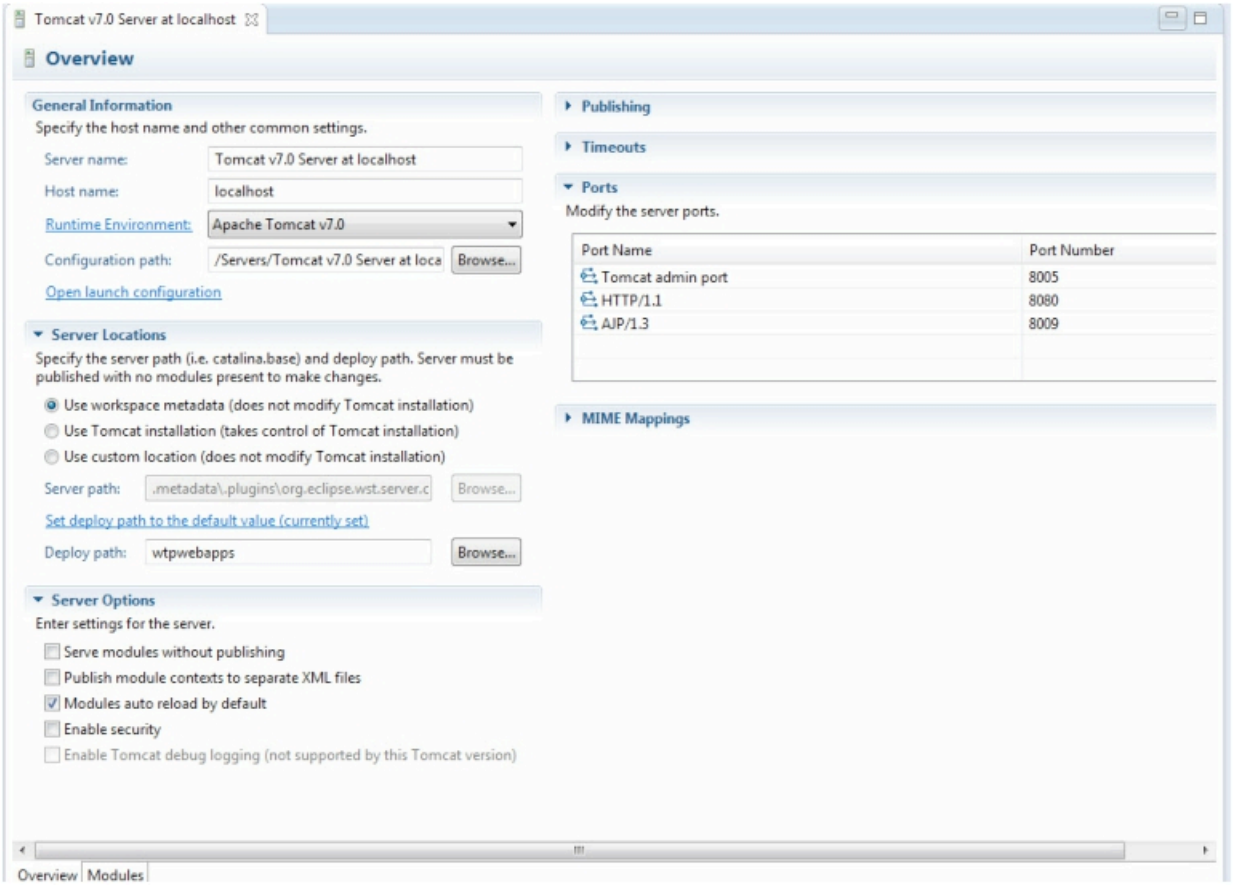
7. Click **Next** .
8. In the **JRE** drop down you must select a **full version of Java** (i.e., a JDK, as opposed to merely a JRE) as the web server needs the compiler classes to compile any JSPs into servlets. Select a **full** JDK from the drop down. If one is not available then click on button next to the drop down labeled **Installed JREs…** and create an entry to a full JDK. It is good practice to make this JDK the Workbench default.
9. Click **Browse** and navigate to the installation directory for Tomcat.
10. Click **Finish**. The Server Installed Runtimes will now have an entry for the Tomcat server. Click **OK** .



11. If you run into any problems after completing the above, ask your instructor for help.
12. You will now go through the process of setting up a server instance, configuring it, and then running it to verify that everything is configured correctly.
13. Ensure that you are in the **JEE Perspective** . Press **Ctrl+N** to open the **New** dialog. Select **Server** → **Server** and click **Next** . Select the **Tomcat v7.0 Server** entry and **Finish.**
14. Open the **Server** view and you will see the entry for the new server that you have created. Note that it is currently stopped. Most operations that you want to perform with the server can be accessed by right-clicking on the server entry.



15. Double-click on the server entry. This will open the **Server Overview** and show you various configuration entries for this server instance.



16. Select **Use Tomcat installation (takes control of Tomcat installation)** and save the configuration by closing this overview tab (click on the "x" of this tab) and confirming the changes if prompted by a dialog.
17. We recommend running Tomcat in this configuration for a couple of reasons. First, this version of Eclipse seems to do a better job of coordinating, publishing, and adding/removing projects when the server is set up in this fashion. Secondarily, publishing the projects out to the Tomcat installation makes the operation much more explicit and provides an opportunity to see the published applications in the file structure under the Tomcat installation. Third, any configuration changes you make via this facility (e.g. port numbers, etc.) will remain effective even when Tomcat is run stand-alone (i.e., without control from Eclipse).
18. Right-click on the server entry and select **Start** . The server should start, with status messages appearing in the **Console** to reflect a successful startup.

19. To stop the server, you can click on the red square in the console or return to the 'Servers' view and right-click on the server entry and select **Stop** . The server status should change to **Stopped** .

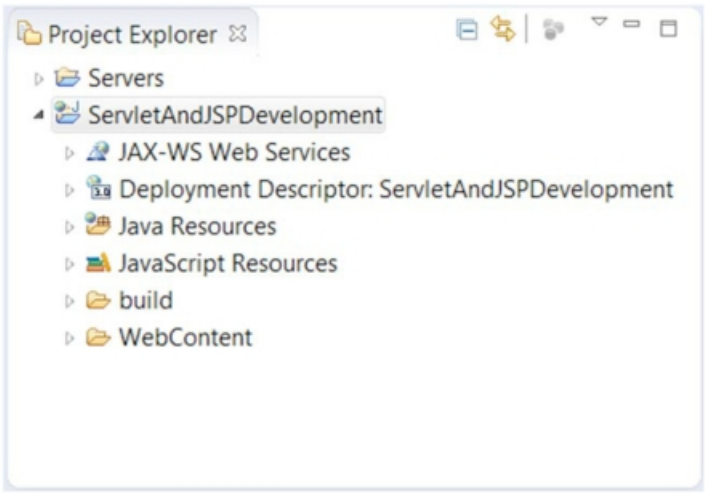
Creating and running a Web Project

1. Press **Ctrl+N** to open the **New** dialog. Select **Web** → **Dynamic Web Project** and click **Next** .
2. Enter a project name, for example, of **ServletAndJSPDevelopment** . Your project name will vary based on the lab you are implementing. Check the lab instructions for the actual name.

   Ensure that the Target Runtime **pulldow n has** Tomcat 7.0 **selected.**

3. Continuing from the New Dynamic Web Project dialog, ensure that **Configuration** is set to the default and **EAR membership** is **unchecked** . Click **Finish** . If an Open Associated Perspective dialog box appears, accept the perspective change by clicking on **Yes** .

   Your project will be listed in the **Project Explorer**.



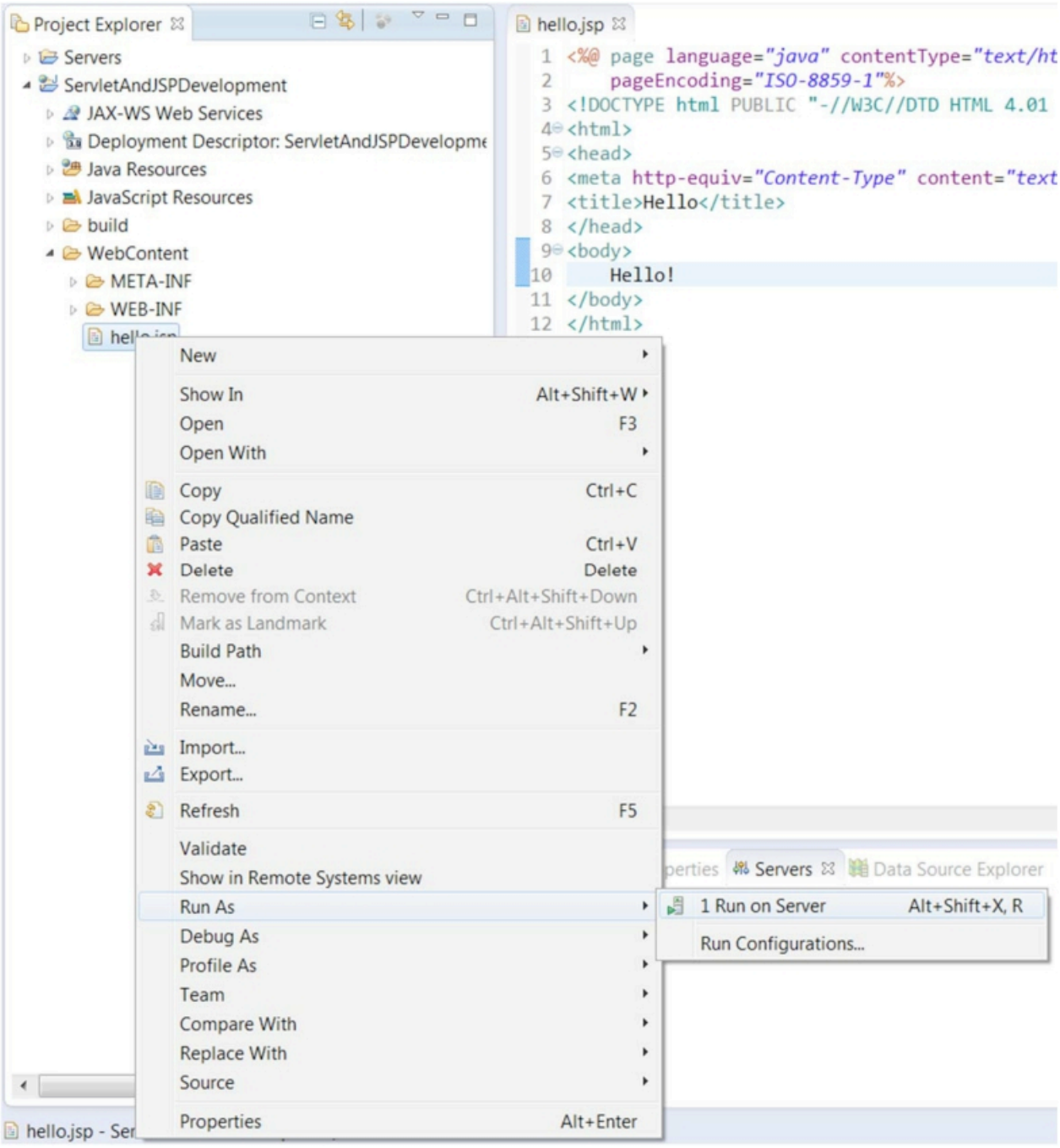
4. Next, we'll create a simple **JSP** file to test the server configuration. In the Project Explorer, select the "node" of the newly created project and then press **Ctrl+N**. Select **Web** → **JSP File** . Click **Next** .
5. Select the parent folder **ServletAndJSPDevelopment/WebContent** . Enter a filename, such as **hello.jsp**. Click **Finish** .
6. When the **hello.jsp** file opens enter some text in the **<TITLE>** and **<BODY>** elements. You can enter a simple string, such as **Hello** .

```
<HTML>
<HEAD>
      <TITLE>Hello</TITLE>
</HEAD>
<BODY>
      Hello!
</BODY>

</HTML>
```

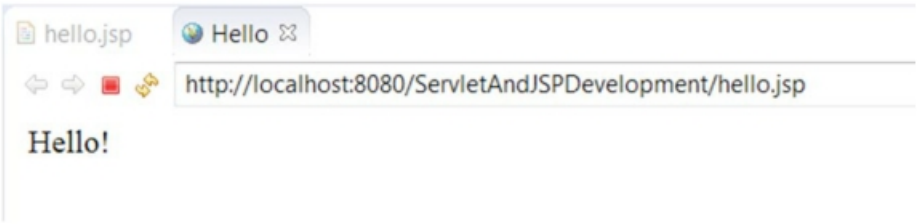
7. Save the file.

8. In order to check the above file a number of things have to happen:
9. The web server (application server) must be started.
10. The application must be deployed to the web server.
11. A web client (normally, a browser) must be started and it must send a request to the proper URL to cause the page to be executed and displayed.
12. All three of those tasks will be executed by selecting one popup menu item. Right-click on **hello.jsp** (you may have to open the **WebContent** folder to see it), and select **Run As → Run On Server**.



13. The **Run On Server** window will open. If you have already defined a **Tomcat 7.0** server for this project (essentially, a "deployment profile"), select **Choose an existing server**, make sure that the server you defined is selected, and select **Finish** . If you have not yet defined a server, select **Manually Define a New Server**. Then select **Tomcat v7.0** .
14. Click **Next** again to check that the project is in the **Configured Projects** list. If it is not then select it from the **Available Projects** list to the left and click **Add** to move it to the right.
15. Click **Finish**.

16. You will see build and deployment output quickly appearing in the **Console** view. First, an Ant task that publishes the web project to the server is run and then the app server is run. When it completes, a web browser window will open (by default, the built-in Eclipse browser will create a new tab in the IDE's file-editor area) and display the string **Hello!**. It may take some time to deploy the application and start up the Tomcat instance if it is not already running. Initially, the web browser may show an error message because the application is not yet deployed. After a few seconds, refresh the browser and the deployed application should be accessible.



17. If you run into any problems please ask your instructor for assistance.

Note: When you deployed this application to the Tomcat instance, you selected a web resources (hello.jsp) as the point from which it initiate this operation. Anytime that you are deploying an application to a Tomcat instance, do so from a web resource under the web project's Web Content folder.

Creating another Web Project.

1. On the workbench, select :

   File ->New -> Dynamic Web Project

2. Enter the project name and click 'Finish'

   The project name will be specified for each exercise (use the project name ' *Tutorial'* for this exercise).

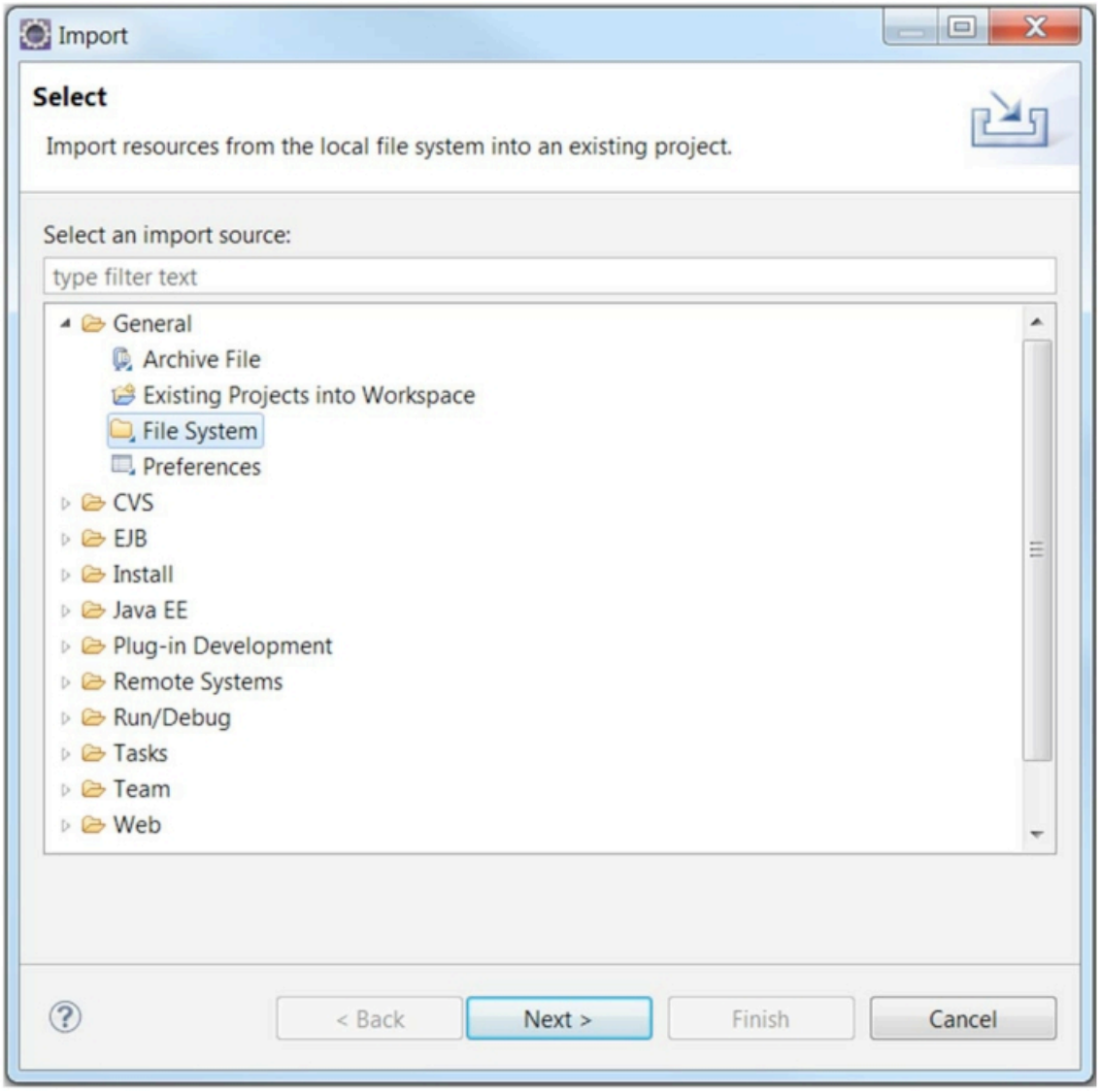
   Always target the appropriate runtime.

   Note: By default, Eclipse uses the project name as the context root for the associated web application. In order to change this default, select **Next** twice and change the context root entry at the location prior to selecting **Finish** . To change the context root at a later time, right-click on the web project entry in the **Project Explorer,** select **Properties,** select **Web Project Settings** , and change the context root entry at that location.

Importing the Java Code

1. In the **Project Explorer**, expand the tree for this project and right-click on the project name, select **Import** , and then **Import** again. This will open an **Import** dialog box.

2. Expand the **General** entry, select **File System** , and then **Next** .

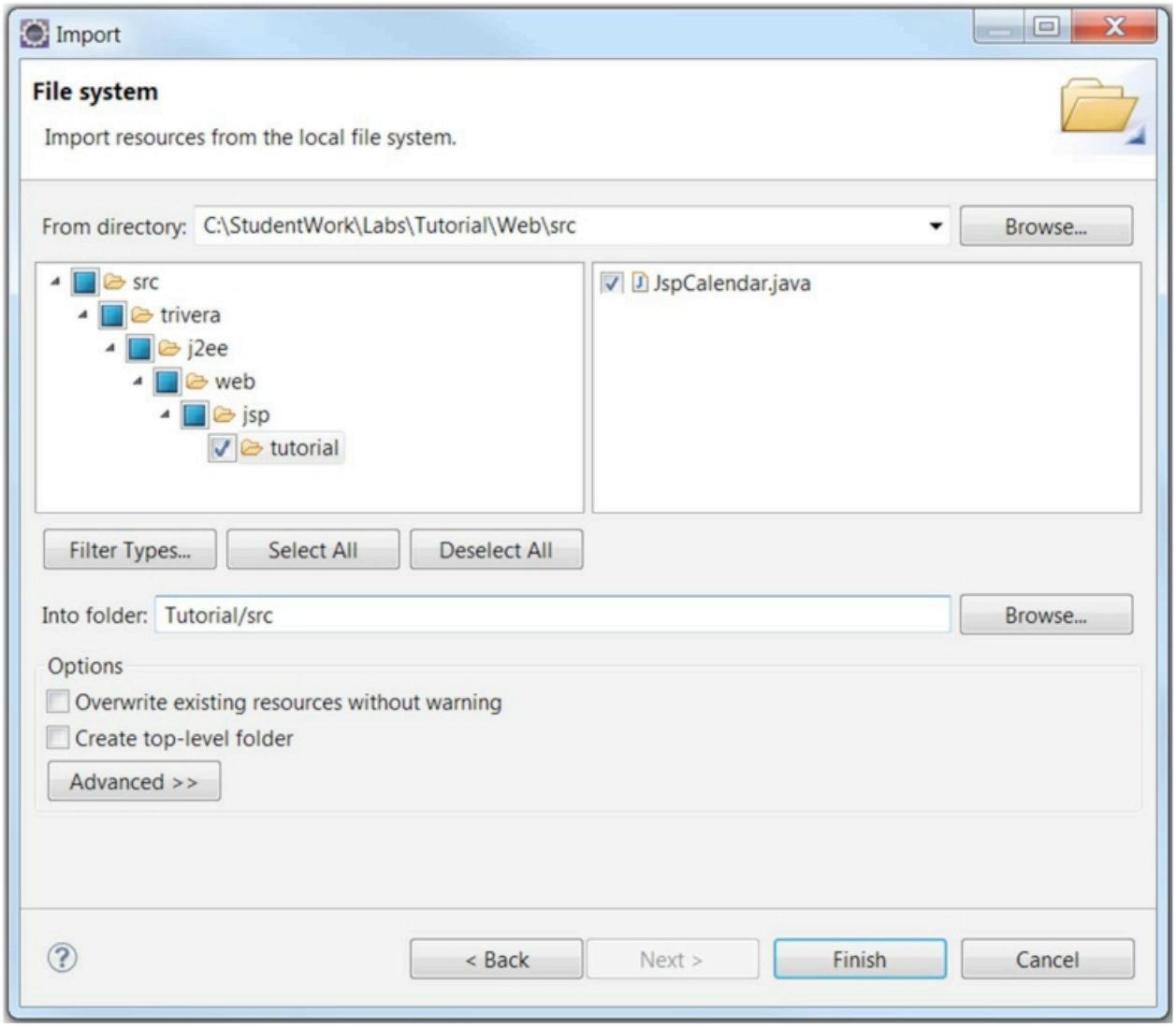


3. On the File System Import panel, browse to the directory in which the Java exercise code resides (e.g. *C:\StudentWork\Labs\Tutorial\Web* ***src***). Note that, while Eclipse doesn't remember the last location you browsed to, there is a drop down menu that can provide alternative starting points (where you have been in the past).

4. Select the necessary Java files as specified for each exercise. For this lab, select *kiddcorp.j2ee.web.jsp.tutorial.JspCalendar.java.*

   NOTE: Source code must be placed in the correct place in Eclipse's project structure or Eclipse will not treat the files as source code. In the case of a web project, the source code must be imported into the *"src"* subfolder.

   IMPORTANT: Select the **Browse** button next to the **Into folder** box and select the **src** folder in your current project.



Note the symmetry of the **" src"** contents being imported into the **" src"** project folder.

5. Select **Finish** to import the code .

Importing the Web Content

1. In the **Project Explorer**, right-click on **WebContent**

2. Click on **Import**

3. Select the **File System** import resource

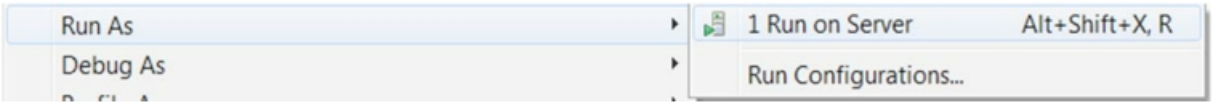
   a. On the **File System Import** panel, browse to the directory in which the Web content resides (e.g. *C:\StudentWork\Labs\Tutorial\Web\WebContent* for this lab).

Note the symmetry of the **"WebContent"** contents being imported into the **"WebContent"** project folder

b. Select the necessary files (in this case date.jsp) and click **Finish** . (The necessary files will be specified separately for each exercise) Typically, the entire contents of the WebContent folder should be imported into the WebContent folder in the workspace. For this exercise, you can select the WebContent folder, which will select *date.jsp* and the *WEB-INF* folder and its contents.

Deploy your Application

1. In the **Project Explorer**, right-click on a web resource such as **date.jsp**
2. Click on **Run As -> Run on Server**



Select **Choose an existing server** and **Tomcat 7.0 Server @ localhost**

Click **Finish**. It may take some time to deploy the application and start up the Tomcat instance if it is not already running. Initially, the web browser may show an error message because the application is not yet deployed. After a few seconds, refresh the browser and the deployed application should be accessible.

Examine Results

1. The application will be deployed on the server. If the deployment fails, the error will be displayed in the deployment panel.

2. Once the application has been deployed, Eclipse will start a browser, showing the welcome screen of your Web application. For this application, a simple display of the date and time should appear similar to this:

Day of month: is 6
Year: is 2014
Month: is February
Time: is 14:28:19
Date: is 2/6/2013
Day: is Thursday
Day of Year: is 37
Week of Year: is 6
era: is 1
DST Offset: is 0
Zone Offset: is -7

3. In addition, you can access your web application by starting your default web browser and point to :

- http://localhost:8080/Tutorial/date.jsp

4. If the server was already running when you deployed a new Web application, the server may need to be restarted, but if so, here is how:

   - On the **Servers** panel, right-click on the server entry and choose **Restart -> Start**

5. You must wait until the server status on the **Servers** Pane indicates **Started** , before you can test your application.

6. On the **Console** pane, the server will also indicate when it's capable of handling requests.

7. If you are successful in importing and deploying this web application, you will see today's date and time displayed in the browser.

Redeploy your Application

When you make changes to your application after it has been deployed, your application does not always have to be deployed again. The server may pick up updates automatically when the application is rebuilt. If the server does not seem to have picked up the changes, here is what to do:

1. Make sure all the edited files have been saved

2. In the **Server** view, right click on the targeted server entry, and select **Publish** to cause the application to be redeployed.

3. If that does not cause your changes to be deployed, rather than select Publish, select **Add and Remove Projects** , remove the project, and select **Finish** to get Tomcat to undeploy the application. Once that has completed, follow the same steps to add the project to the server and deploy it.

TUTORIAL : WORKING WITH TOMCAT

This tutorial covers how to perform the following operations in conjunction with Tomcat 7.0 and higher:

Start and stop the server

Access the Tomcat Status area

Deploying web applications to Tomcat

Starting and Stopping Tomcat 7.0 in "standalone" mode

1. Navigate to the Tomcat installation directory, which is, by default **C:\Program Files\Apache Software Foundation\Tomcat 7.x**, but in this course it is likely to be **C:\tools\apache-tomcat-7.0.50** , (where the "50" denotes the maintenance-release number for this version). Hereafter, this directory shall be referred to as **<Tomcat>** .
2. There are various subfolders, the three that are of most interest to us are the

   bin subfolder where the Tomcat executable is

   conf subfolder where Tomcat configuration is captured and maintained

   webapps subfolder where web applications are deployed to
3. Navigate to the **bin** subfolder and double click on the **startup.bat** file

   A DOS window should appear along with a series of startup messages

   One message should indicate where Coyote is listening (default is port 8080). You will use localhost and this port to access web resources on this Tomcat instance

   To verify that Tomcat is up and running, open a browser, enter the URL http://localhost:8080 , and check the result. You should see Tomcat's default page.
4. To stop Tomcat, navigate to the bin subfolder and double click on the **shutdown.bat** file.
5. You may find it convenient to make shortcuts these executables on your Desktop.

Accessing Tomcat Status Area

1. In order to enter the status and administrative areas of Tomcat's web page, you must be able to log in. Let's setup a username and password that we can use to access the administrative functions on Tomcat.
2. Navigate to the **<Tomcat>/conf** folder. Open the **tomcat-users.xml** file with a text editor. This file sets up users and roles for Tomcat as well as for web applications that Tomcat is running.
3. Between the XML tags **`<tomcat-users>`** and **`</tomcat-users>`** , Add entries in the file for a user and password that has a role of manager-gui. These should be:

<role rolename="manager-gui"/>

<user username="tomcat" password="s3cret" roles="manager-gui"/>

4. Take note of that username and password.

5. If you had to make an entry, you will need to “bounce” the server, i.e., stop Tomcat (if it is currently running), then re-start it. Open a browser, and access the Tomcat default page via **http://localhost:8080** .
6. In the upper right corner there is a link to **Server Status** . Select that link and enter the username and password that you made note of earlier.
7. You will now see the status area for Tomcat, showing what web applications are running, what their context roots are, and other information.

? Deploying Web Applications to Tomcat

It is very simple to deploy a packaged web application to Tomcat.

- ? Tomcat will recognize a newly deployed (or updated) web application while it is running and will bring that web application on line.
- ? Navigate to <Tomcat>/webapps and note the contents and subfolders.
- ? This is the folder where you want to copy your WAR files so that Tomcat can process that WAR file and bring it on line.
- ? When you copy a WAR file into this folder (if Tomcat is running), watch the status messages in the Tomcat DOS window. You will see Tomcat recognize the new WAR file and automatically deploy it (which includes extracting its compressed contents to a web application folder of the same name, and setup a web application context root of that name within Tomcat).
- ? Once the new web application has been deployed, you should be able to access it via whatever you set up the context root to be.