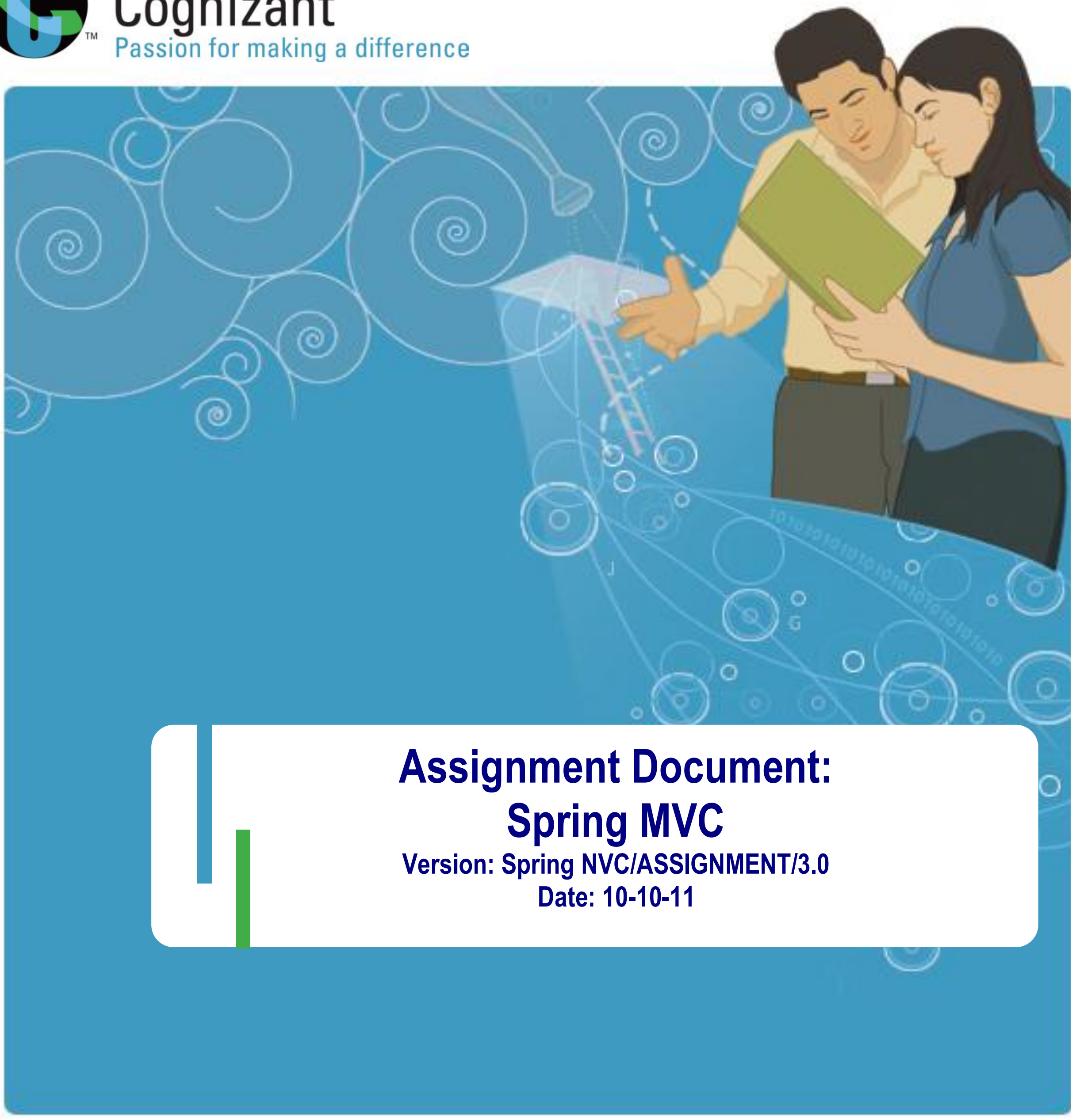


Assignment Document: Spring MVC

Version: Spring NVC/ASSIGNMENT/3.0
Date: 10-10-11

Cognizant
500 Glen Pointe Center West
Teaneck, NJ 07666
Ph: 201-801-0233
www.cognizant.com

Contents

Topic: Spring MVC

Hands-On Exercise 1: Setup for Spring and Create a Spring Application

Estimated Completion Time: 45 Minutes

Objectives: During this exercise we will prepare the Eclipse 3.7 WTP environment to work effectively with Spring and the various technologies that we will be integrating with it. We will also review how to configure and run a Spring program, how "Dependency Injection" works, and the basics of the Spring XML configuration file.

Instructions:

Start Eclipse and create a new workspace if on has not already been set up for you.

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.

1. 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.

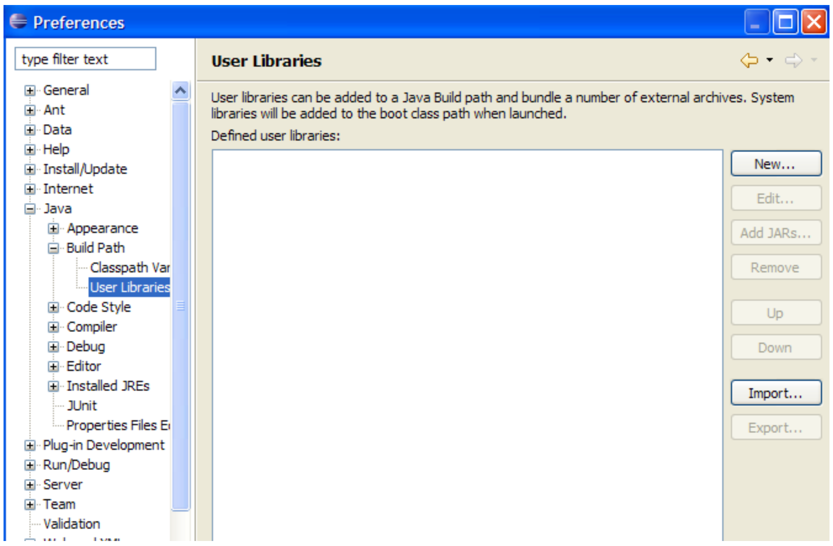
   Take a few minutes to examine the contents of this folder and sub-folders. In particular, examine ${SPRING_ROOT}/dist; this folder contains JAR files for the various Spring modules. In this course you will use many of these JAR files for both compiling and executing code.

   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.

2. 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.



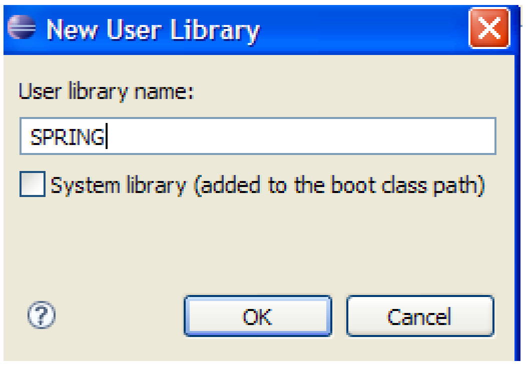
User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

Click the *New* button.

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



User library name: SPRING

System library (added to the boot class path) [ ]

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

The necessary Core jars are listed below. 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 will create a second user library called `SPRING-DEPENDENCIES` that will contain references to all of the supporting technologies. Using the steps outlined above, create a second user library using the name `SPRING-DEPENDENCIES`. As you did for the first library, add several JAR files. Navigate to the ~/StudentWork/Tools/SpringDependencies folder and work through adding each of these JAR files.

```
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
```

Under each JAR file are four indented lines. The first one “Source attachment” has `(None)` as the value. In the next steps you will associate the source with each JAR file. (This Step is not necessary for basic development. It does help with debugging, but it is a tedious process)

To associate the source code with a library JAR file, click on the line “Source attachment” and then click the *Edit* button. This will bring up a dialog box. Click on the *External File…* button and navigate to the `${SPRING_ROOT}/src` directory and select the JAR file that contains the source code for that module. Click the *OK* button.

3. 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. One of the Spring view-related options will be the first of the JEE applications that we will be working with.

4. Create a HelloWorld Project

Setup a new Java project in Eclipse called **HelloWorld.** 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 **HelloWorld** project's entry and select **Properties**. A dialog box will appear. Select **Java Build Path** in the left pane. Select the **Libraries** tab and then **Add Library -> User Library -> Next -> SPRING** and **SPRING-DEPENDENCIES -> Finish -> OK**. This brings the Spring library classes and the companion technologies into play during editing and compilation.

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

5. Create the Messager Interface and its two subclasses.

Right click on the **com.springclass** package and select **New -> Interface**. 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.

6. 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.

7. 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.

8. 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

   ~\StudentWork\Labs\Spring-HelloWorld\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" />
```

9. 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.

As an added note, you can always increase the amount of logging that Spring is performing. This is helpful not only in understanding what is going on behind the scenes, but can provide invaluable hints as to what is going wrong. The entry to make is:

log4j.logger.org.springframework=DEBUG, stdout

10. Run the program and inspect the output

Run the **HelloWorldMain** program. You can execute **HelloWorldMain** by right-clicking on it and selecting **Run As -> Java Application.**

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

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.

11. 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

12. Run the program and inspect the output

    Run the **HelloWorldMain** program. You can execute **HelloWorldMain** by right-clicking on it and selecting **Run As -> Java 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.

Hands-On Exercise 2: Using Spring MVC

Estimated Completion Time: 90 Minutes

Objectives: To learn how to build a web application using Spring MVC, including forms and data acquisition. In terms of the application we will be working with, we will be supporting an application that displays information about DVDs and provides the functionality to add more DVDs.

Note: If you are not familiar with Eclipse or setting up and working with Tomcat in Eclipse, please work through the tutorial titled Working with Eclipse Indigo (JEE Version) and Tomcat 7. This tutorial will lead you through setting up and running Tomcat, building a web application and deploying it to Tomcat, and importing and running an existing web application.

Instructions:

1. Inspect the Classes

   Create a new Dynamic Web Project called **SpringMVC.** Set the Target Runtime to ApacheTomcat v7.0 (You should have setup Tomcat earlier in the tomcat setup lab)

   Visibility of classes at compile time

   For this lab, add the **SPRING and SPRING-DEPENDENCIES** user libraries 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 Helios 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)

   SPRING-DEPENDENCIES (user library)

   Jstl-impl.jar

   Jstl.jar

   Jsf-impl.jar

   Jsf-api.jar

   Jta.jar

   Javax.jms.jar

   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 **~/StudentWork/Spring-MVC/Web/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 **~/StudentWork/Spring-MVC/Web/WebContent**) into the project's **WebContent** 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).

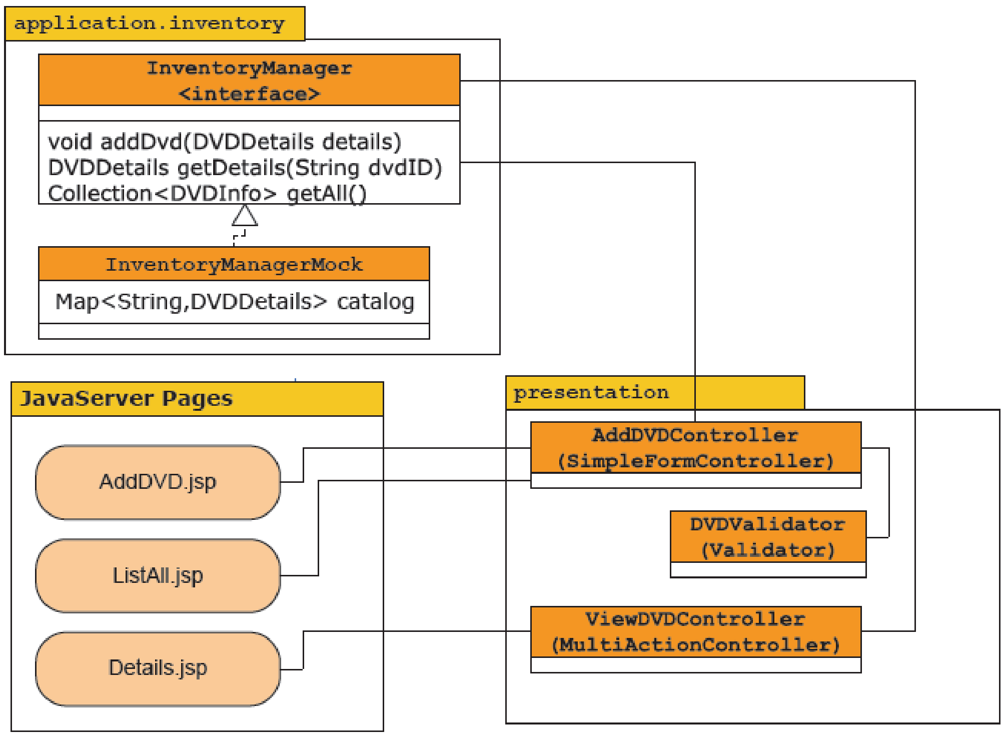
During this lab, you will be producing three views:

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.


application.inventory
InventoryManager <interface>
void addDvd(DVDDetails details)
DVDDetails getDetails(String dvdID)
Collection<DVDInfo> getAll()
InventoryManagerMock
Map<String,DVDDetails> catalog
JavaServer Pages
AddDVD.jsp
ListAll.jsp
Details.jsp
presentation
AddDVDController (SimpleFormController)
DVDValidator (Validator)
ViewDVDController (MultiActionController)

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. Take a look at this implementation, so you understand what is going on once things get going.

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

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 **/views/inventory** which corresponds to the directory you imported in the **WebContent** 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.

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.

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).

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.

6. Deploy and Test

Deploy and test the application. Add in the expanded logging and watch what happens when while you are starting up and then interacting with the application.

Hands-On Exercise 3: Using Spring MVC-Supported Validation
Estimated Completion Time: 60 Minutes

Objectives: During this exercise we extend the previously built Spring MVC application to include validation using the reference implementation for the JSR 303 Bean Validation specification.

Instructions:

1. Setup project to use the Hibernate Validator

   JSR 303 – Bean Validation defines an API for JavaBean validation. The Hibernate Validator 4.1 is the reference implementation for the specification. We will be setting the Spring MVC project up to use the validator to validate the incoming DVD data during the Add DVD process.

   Visibility of classes at compile time

   Add the jars within the ~/StudentWork/tools/Hibernate Validator folder to the Build path of the project:

   hibernate-validator-4.2.0.Final.jar

   hibernate-validator-annotation-processor-4.2.0.Final.jar

   validation-api-1.0.0.GA.jar

   Note that all three of these JARs are required at both compile and runtime. The validation-api-1.0.0.GA.jar is from the JSR 303 specification and is what we will be programming to.

   Visibility of classes at runtime time

   In Helios 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:

   hibernate-validator-4.2.0.Final.jar

   hibernate-validator-annotation-processor-4.2.0.Final.jar

   validation-api-1.0.0.GA.jar

2. Implement a Validator for the DVDDetails

   In this step, we will implement the `org.springframework.validation.Validator` interface in the new class called DVDDetailsValidator. The Validator interface requires the implementation of two methods:

   - The supports method that is used by the framework to check where an object is in an appropriate class to be validated by this validator.

- The validate method which is passed in the object to be validated and the `org.springframework.validation.Errors` data structure for accepting errors. These can be processed by the framework if it is configured to do so.

Recall that the DVDDetails class has several String values. The validation criteria that we will be implementing includes the following:

id: Must not be empty and must be parsable into an integer

title: Must not be empty

releaseYear: Must not be empty, must be parsable into an integer, and must be between the values of minYear and maxYear.

Create the DVDDetailsValidator class in the same package as DVDDetails implementing the Validator interface.

Use the following code as a guide in performing the validation.

```
public class DVDDetailsValidator implements Validator {


  private int minYear = 1900;
  private int maxYear = 2013;

  public boolean supports(Class clazz) {
    return clazz.equals(DVDDetails.class);
  }

  public void validate(Object obj, Errors errors) {

        DVDDetails details = (DVDDetails) obj;

        if (details == null) {
      errors.rejectValue("id",
                          "error.not-specified", null, "Value required.");
    } else {
        if (details.getId().isEmpty()) {
        details.setId("Need Value");
          errors.rejectValue("id", "no value");
    } else {
        if (!isIntNumber(details.getId())) {
        details.setId("Need integer");
          errors.rejectValue("id", "not integer");
    } else {
        if (details.getTitle().isEmpty()) {
        details.setTitle("Need Value");
          errors.rejectValue("title", "no value");
    } else {
        if (details.getReleaseYear().isEmpty()) {
        details.setReleaseYear("Need Value");
          errors.rejectValue("releaseYear", "no value");
    } else {
        if (!isIntNumber(details.getReleaseYear())) {
          details.setReleaseYear("Need integer");
          errors.rejectValue("releaseYear", "not integer");
    } else {
       int year = Integer.parseInt(details.getReleaseYear());
        if (year < minYear) {
        details.setReleaseYear("Greater than " + minYear);
          errors.rejectValue("releaseYear", "too long ago");
        } else {
        if (year > maxYear) {
        details.setReleaseYear("Less than " + maxYear);
          errors.rejectValue("releaseYear", "in future");
        }
        }}}}}}};
  }

  public void setMinYear(int i) {
    minYear = i;
  }


  public void setMaxYear(int i) {
    maxYear = i;
  }

  public boolean isIntNumber(String num){
  try{
  Integer.parseInt(num);
  } catch(NumberFormatException nfe) {
  return false;
  }
  return true;
  }

}
```

Note that we set the values of the errant DVDDetails instance to an appropriate error message. This is because the instance will be returned to the form that it was submitted from and redisplayed. This provides a convenient mechanism for displaying an error to the user in exactly the data location where the error occurred.

3. Set up Spring to Locate a JSR 303 Validator

Spring provides the needed mechanisms to locate and work with a JSR 303 Validator. The first step in making that happen is to make sure that the validator is in the class path at run time (which we have already ensured). Next, we need to set up a bean to locate the validator. Open the spring-servlet.xml file and make the following entry:

```
<bean id="validator"

      class=

"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

The Spring framework will automatically detect the validator in the classpath and trigger it to initialize.

4. Set up Controller to Automatically Invoke Validator

There are several things in the AddDVDController that need to be set up so the proper validator will be invoked at the proper time on the proper object. The first thing we will be doing is to configure a validator to be called. This can be done through the spring-servet.xml file to set a validator for use across all controllers.

However, we are going for a more localized approach by setting the validator up at the controller level. This is done by setting the validator inside the @InitBinder callback. Make the following entry into the AddDVDController class:

```
@InitBinder
  protected void initBinder(WebDataBinder binder) {
      binder.setValidator(new DVDDetailsValidator());
  }
```

Note that this will cause the controller to apply this validator whenever validation is called for no matter what the object is. You can specify the specific object to link this validator with by including the object's name as an argument for the @InitBinder annotation.

The next item that we have to add in is an indicator of where in the processing we would like to have the validation occur. This is accomplished by annotating the targeted parameter with the @Valid annotation. The @Valid annotation is actually not a Spring annotation but is from the JSR 303 specification.

When we use the @Valid annotation on a parameter, the associated validator will be invoked as part of the data binding process. We need to control our processing by checking the results of the binding to see if there were any errors. We will need to change the ModelAndView signature to have the BindingResult passed into our method so we can check it.

Add the @Valid annotation and the BindingResult parameter as shown below:

```
@RequestMapping(method = RequestMethod.POST)
protected ModelAndView onSubmit(
   @ModelAttribute("command") @Valid  DVDDetails d,
   BindingResult binder) {
```

Finally, we will check the BindingResults to see if there were any errors. If so, we are simply going to send the processing back to the AddDVD form with the DVDDetail instance that was validated. Recall that we populated this instance with an error message, so those should show up once the form returns. Add the following to the start of the ModelAndView method as shown below:

```
@RequestMapping(method = RequestMethod.POST)
protected ModelAndView onSubmit(
    @ModelAttribute("command") @Valid  DVDDetails d,
    BindingResult binder) {
```

```
 if (binder.hasErrors()) {
   return new ModelAndView("AddDVD", "command", d);
 }…
```

5. Deploy and Run the Application

Run through the application and test your validation mechanisms. If you run into any problems or any curiosity, we suggest setting up the logger so Spring is more verbose.

Tutorial: Working with Indigo (JEE Version) and Tomcat 7

Estimated Completion Time: 45 Minutes

Objective:

This tutorial illustrates how to setup the Eclipse environment for use in implementing the exercises. You will:

- Setup a Dynamic Web Project, create a resource, and run the web application on a Tomcat instance.
- Setup a Dynamic Web Project, import some resources, and run the web application on a Tomcat instance.

By the end of the lab you should be able to:

- Setup a web project and deploy it to an instance of Tomcat
- Verify that Tomcat is operating and determine its status through a variety of means

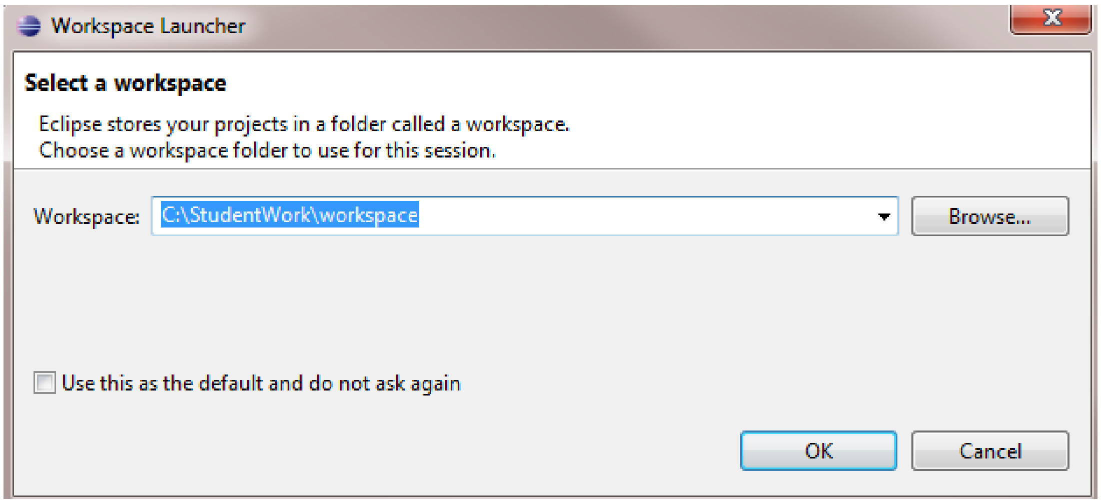
For the location of the Workshop directory, ask your trainer. During this tutorial, we will presume that the Workshop directory has been placed in **C:\StudentWork**

During this training, you will be developing several miniature-projects, using Indigo 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 Indigo. This tutorial describes how to work with various aspects of Indigo (JEE Version) as well as Tomcat.

Instructions:

From the Windows Desktop, double-click the Eclipse icon. If you do not have a shortcut icon on your desktop then add one that references the Eclipse executable. A popup will appear in which you can specify the location of the workspace.

Change the location of the workspace to: **C:\StudentWork\workspace**


Workspace Launcher

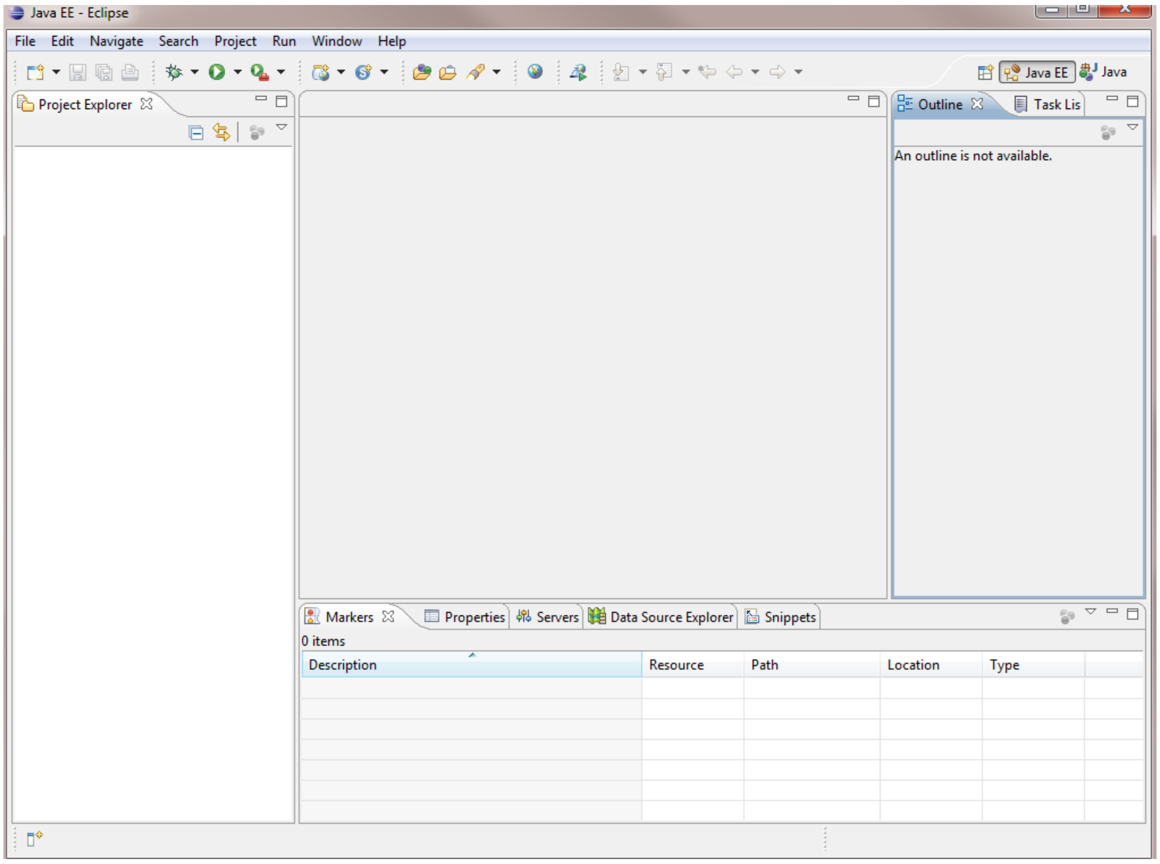
Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\StudentWork\workspace

An empty workspace will now be created in the **C:\StudentWork\workspace** directory. When the application is started, the Workbench will be presented.

Select the Go to WorkBench icon on the right side of the Eclipse window. You should see:



An outline is not available.

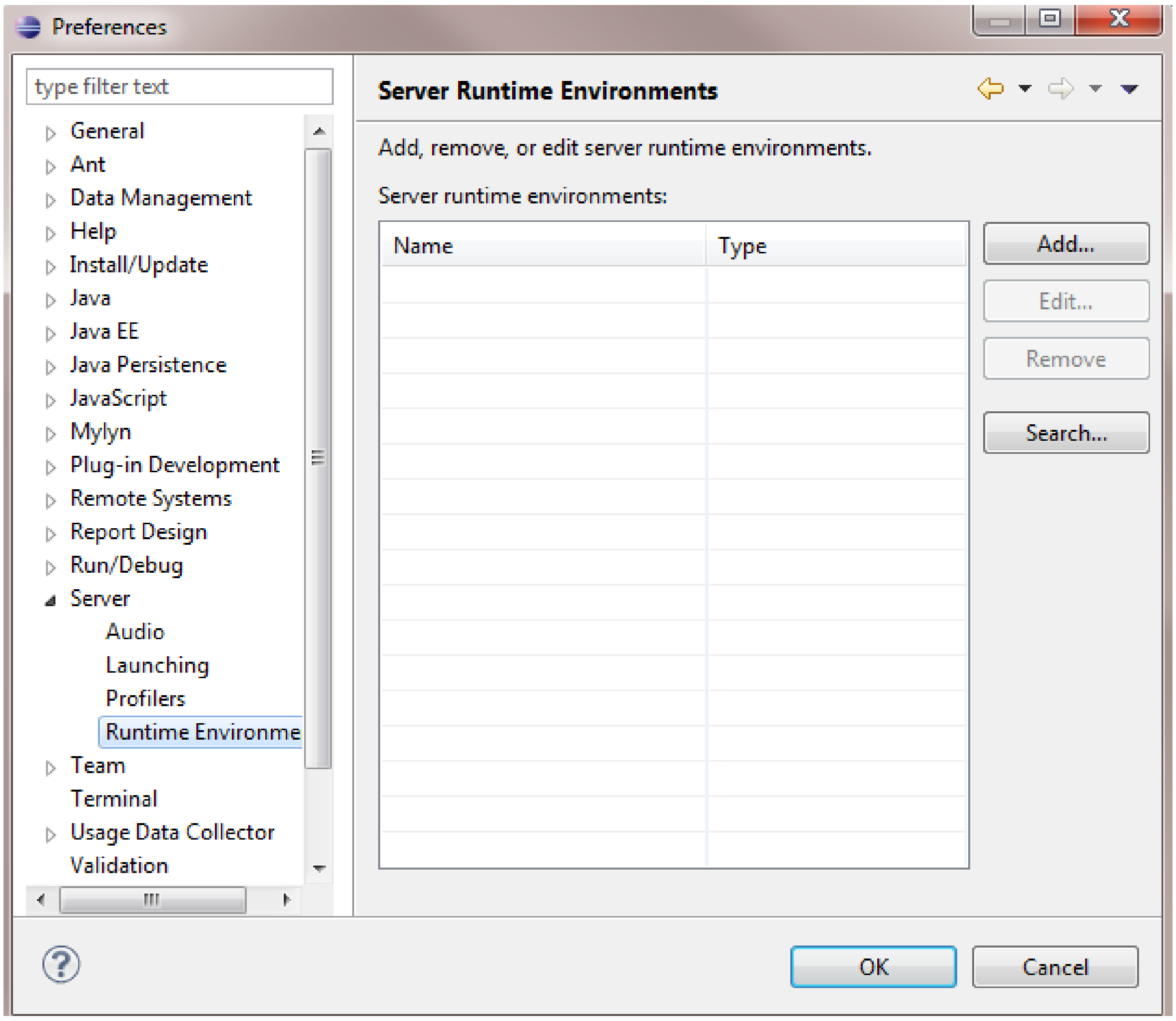
0 items

Description	Resource	Path	Location	Type

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. The server should already be installed on your local machine.

Start Eclipse if it is not already running.

From the main menu select **Window -> Preferences**. In the tree view to the left select **Servers -> Server Runtime Environments**.



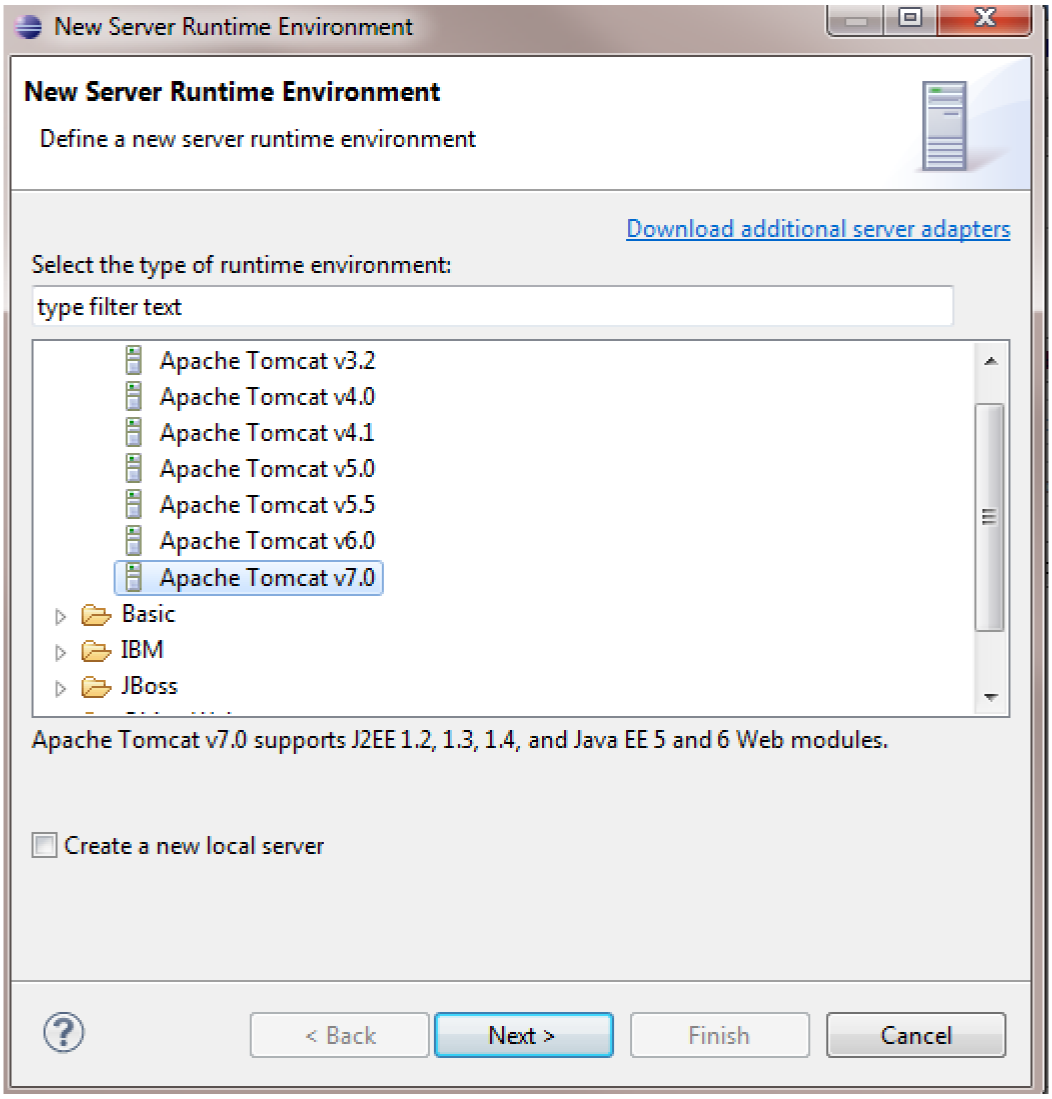
Add, remove, or edit server runtime environments.

Server runtime environments:

Name	Type

There should be no entries available. Click **Add.**

When the **New Server Runtime** window opens select **Apache -> Apache Tomcat 7.0**.


New Server Runtime Environment

New Server Runtime Environment

Define a new server runtime environment

Download additional server adapters

Select the type of runtime environment:

type filter text

- Apache Tomcat v3.2
- Apache Tomcat v4.0
- Apache Tomcat v4.1
- Apache Tomcat v5.0
- Apache Tomcat v5.5
- Apache Tomcat v6.0
- Apache Tomcat v7.0
- Basic
- IBM
- JBoss

Apache Tomcat v7.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5 and 6 Web modules.

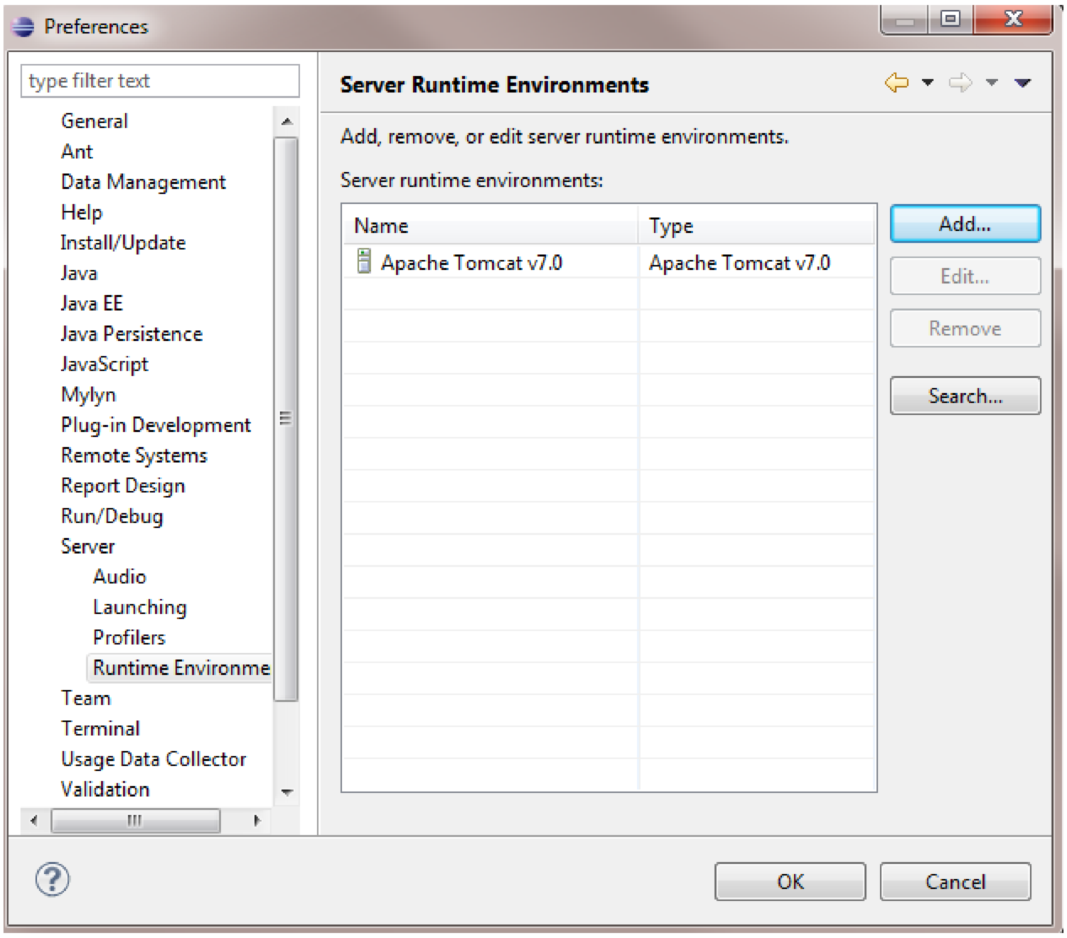
[ ] Create a new local server

Click **Next.**

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 the link just above the drop down labeled **Installed JRE preferences** and create an entry to a full JDK. It is good practice to make this JDK the Workbench default.

Click **Browse** and navigate to the installation directory for Tomcat.

Click **Finish**. The Server Installed Runtimes will now have an entry for the Tomcat server. Click **OK**.



Server Runtime Environments

Add, remove, or edit server runtime environments.

Server runtime environments:

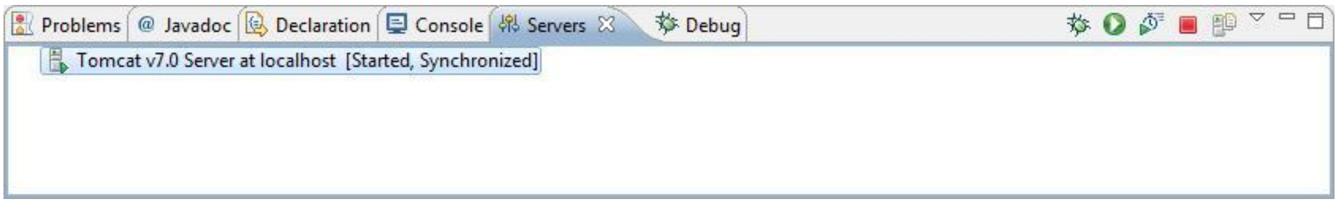
Name	Type
Apache Tomcat v7.0	Apache Tomcat v7.0

If you run into any problems after completing the above, ask your instructor for help.

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.

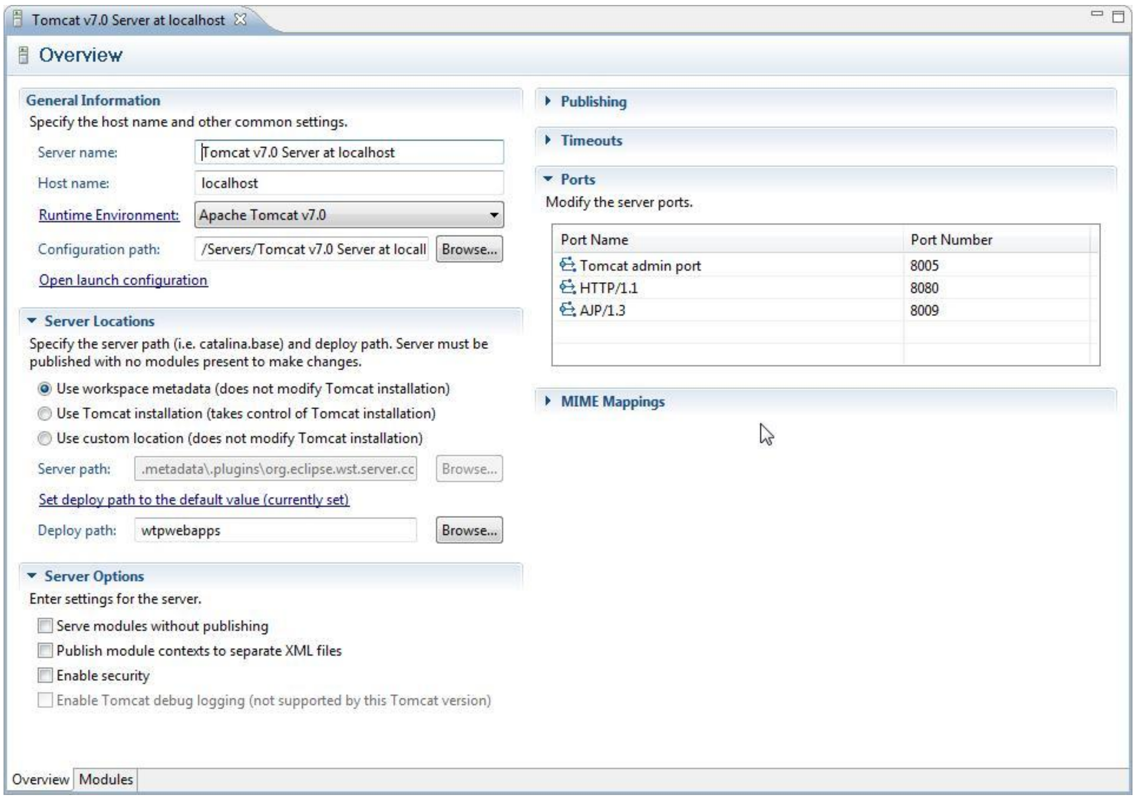
Ensure that you are in the **JEE Perspective**. Press **Ctrl+N** to open the **New** dialog. Select **Server -> Server** and click **Next**. Select the **Tomcat v7.0 Server** entry and **Finish**.

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.



Tomcat v7.0 Server at localhost [Started, Synchronized]

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



Tomcat v7.0 Server at localhost

Overview

General Information
Specify the host name and other common settings.

Server name:	Tomcat v7.0 Server at localhost
Host name:	localhost
Runtime Environment:	Apache Tomcat v7.0
Configuration path:	/Servers/Tomcat v7.0 Server at locall

Open launch configuration

Server Locations
Specify the server path (i.e. catalina.base) and deploy path. Server must be published with no modules present to make changes.

[✓ Use workspace metadata (does not modify Tomcat installation)]
Use Tomcat installation (takes control of Tomcat installation)
Use custom location (does not modify Tomcat installation)

Server path: .metadata\.plugins\org.eclipse.wst.server.cc
Set deploy path to the default value (currently set)
Deploy path: wtpwebapps

Server Options
Enter settings for the server.

☐ Serve modules without publishing
☐ Publish module contexts to separate XML files
☐ Enable security
☐ Enable Tomcat debug logging (not supported by this Tomcat version)

Publishing

Timeouts

Ports
Modify the server ports.

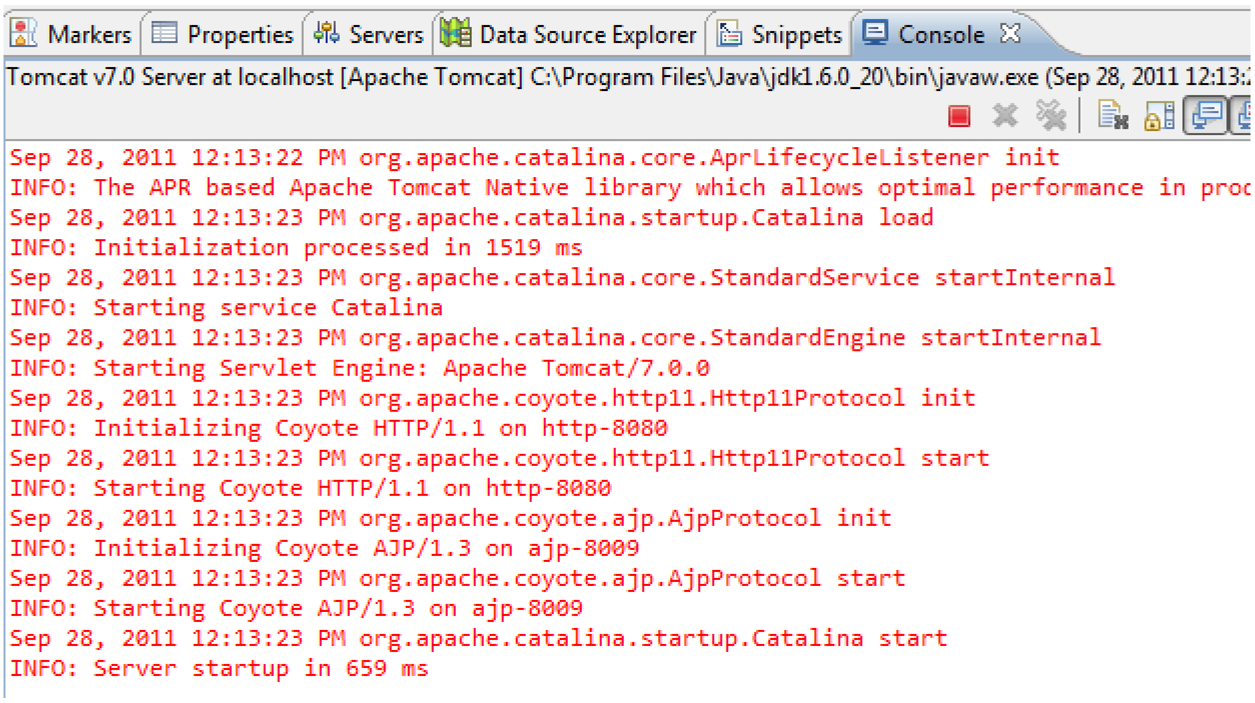
Port Name	Port Number
Tomcat admin port	8005
HTTP/1.1	8080
AJP/1.3	8009

MIME Mappings

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.

We recommend running Tomcat in this configuration for a couple of reasons. First, this version Eclipse seems to do a better job of coordination, 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).

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.



```
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.6.0_20\bin\javaw.exe (Sep 28, 2011 12:13:
Sep 28, 2011 12:13:22 PM org.apache.catalina.core.AprLifecycleListener init
INFO: The APR based Apache Tomcat Native library which allows optimal performance in proc
Sep 28, 2011 12:13:23 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1519 ms
Sep 28, 2011 12:13:23 PM org.apache.catalina.core.StandardService startInternal
INFO: Starting service Catalina
Sep 28, 2011 12:13:23 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.0
Sep 28, 2011 12:13:23 PM org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on http-8080
Sep 28, 2011 12:13:23 PM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Sep 28, 2011 12:13:23 PM org.apache.coyote.ajp.AjpProtocol init
INFO: Initializing Coyote AJP/1.3 on ajp-8009
Sep 28, 2011 12:13:23 PM org.apache.coyote.ajp.AjpProtocol start
INFO: Starting Coyote AJP/1.3 on ajp-8009
Sep 28, 2011 12:13:23 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 659 ms
```

Right-click on the server entry and select **Stop**. The server status should change to **Stopped**

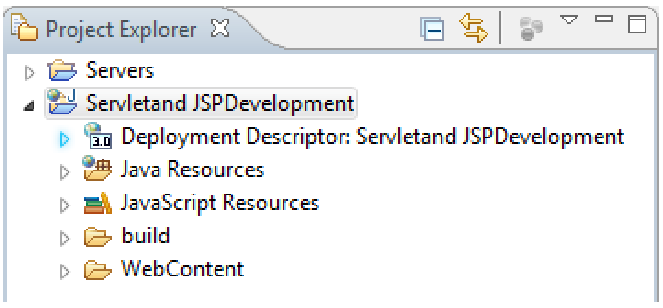
Next, we are going to create and deploy a **Web Project**. Press **Ctrl+N** to open the **New** dialog. Select **Web -> Dynamic Web Project** and click **Next.**

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** pulldown has **Tomcat 7.0** selected.

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**.



```
Project Explorer
▷ Servers
◢ Servletand JSPDevelopment
    ▷ Deployment Descriptor: Servletand JSPDevelopment
    ▷ Java Resources
    ▷ JavaScript Resources
    ▷ build
    ▷ WebContent
```

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. Click Next.

Select the parent folder ServletAndJSPDevelopment/WebContent. Enter a filename, such as hello.jsp. Click Finish.

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>
```

Save the file.

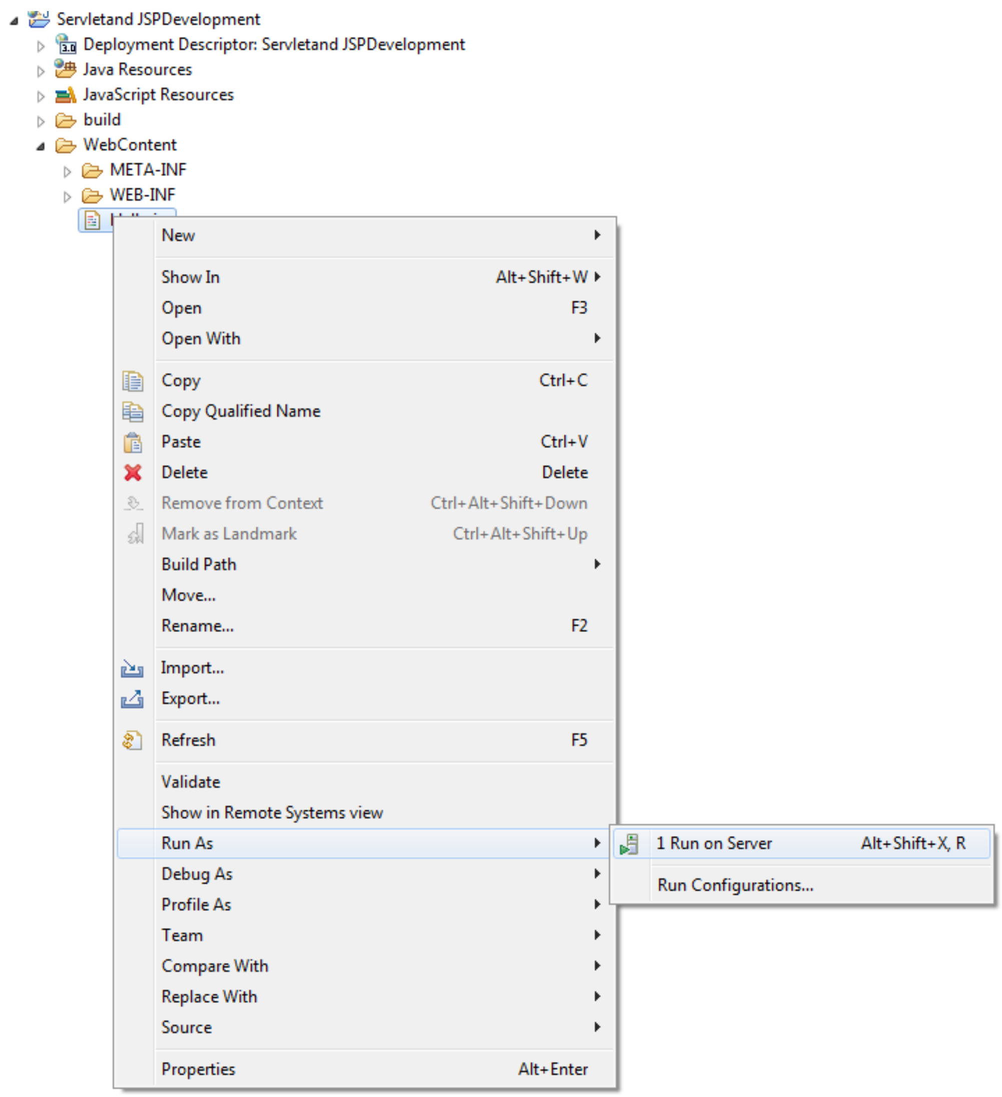
In order to check the above file a number of things have to happen:

The web server (application server) must be started.

The application must be deployed to the web server.

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.

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.

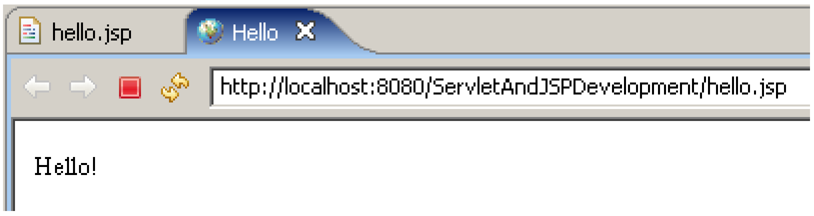


The **Run On Server** window will open. If you have already defined a **Tomcat 7.0** server for this project (essentially, a "deployment profile"), select **Choose an existing server**, make sure that the server you defined is selected, and select **Finish**. If you have not yet defined a server, select **Manually Define a New Server**. Then select **Tomcat v7.0**.

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.

Click **Finish.**

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.



Hello!

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.

Now we are going to create another Web Project and import resources from the StudentWork folder into the project. These need to be properly placed for Eclipse to recognize them and properly deploy them. The folder structure used for this tutorial reflects the folder structure of all the web application-based labs that you will be importing code for in this class.

On the workbench, select **File ->New -> Dynamic Web Project**

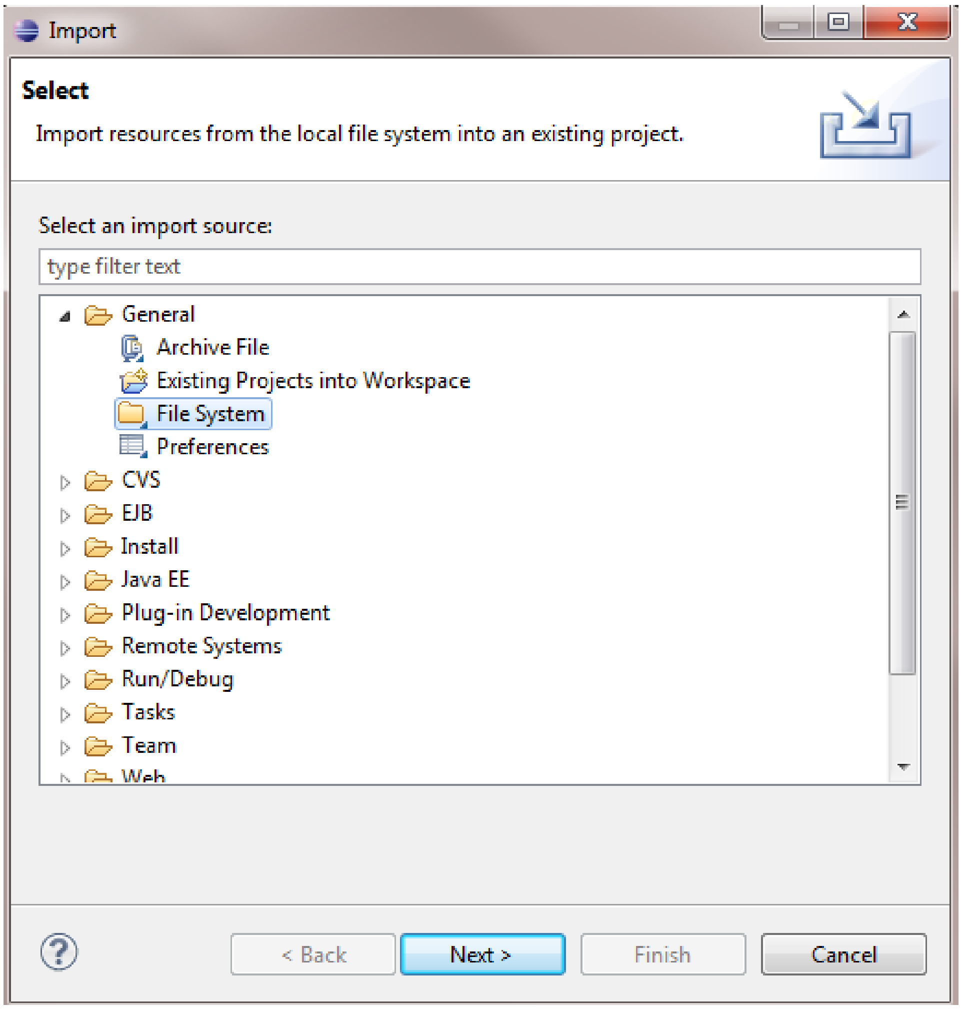
Enter the project name and click **'Finish'**. The project name will be specified for each exercise (use the project name **'Tutorial'** for this exercise).

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.

Now we will import the Java code 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.

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

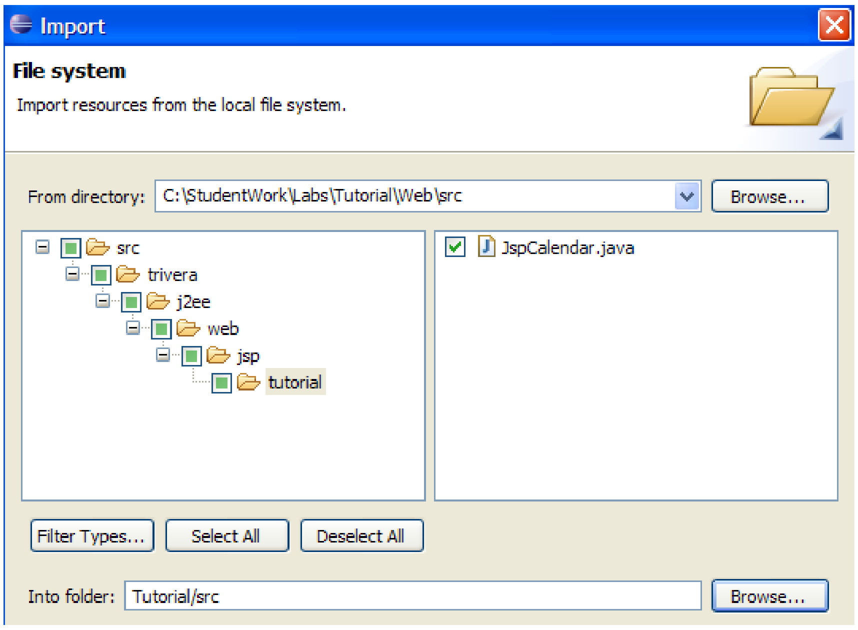


On the File System Import panel, browse to the directory in which the Java exercise code resides (e.g. **C:\StudentWork\Labs\Tutorial\Web\src**). Note that, while Eclipse doesn't remember the last location you browsed to, there is a drop down menu that can provide alternative starting points (where you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

Select the necessary Java files as specified for each exercise. For this lab, select **trivera.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.


Import
File system
Import resources from the local file system.
From directory: C:\StudentWork\Labs\Tutorial\Web\src
Browse...
src
trivera
j2ee
web
jsp
tutorial
JspCalendar.java
Filter Types...
Select All
Deselect All
Into folder: Tutorial/src
Browse...

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

Select **Finish** to import the code.

Importing the **Web Content** is the next task. In the **Project Explorer,** right-click on **WebContent**

Click on **Import**

Select the **File System** import resource

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).

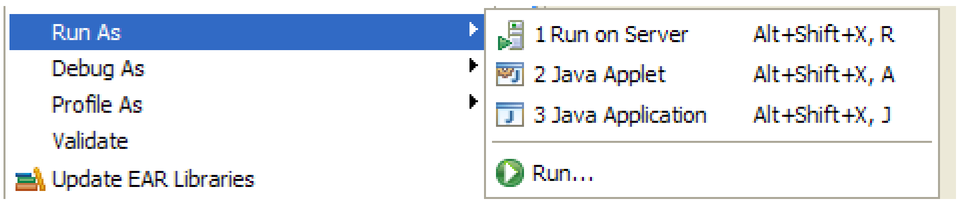
Note the symmetry of the **"WebContent"** contents being imported into the **"WebContent"** project folder

Select the necessary files 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.

Eclipse will ask if you would like to overwrite the existing **web.xml**. Click **Yes**

Finally, we are going to deploy your Application In the **Project Explorer**, right-click on a web resource such as **date.jsp**

Click on **Run As -> Run on Server**


Run As
Debug As
Profile As
Validate
Update EAR Libraries
1 Run on Server Alt+Shift+X, R
2 Java Applet Alt+Shift+X, A
3 Java Application Alt+Shift+X, J
Run...

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.

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

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:

a. Day of month: is 28
b. Year: is 2011
c. Month: is September
d. Time: is 13:43:11
e. Date: is 9/28/2011
f. Day: is Wednesday
g. Day of Year: is 271
h. Week of Year: is 40
i. era: is 1
j. DST Offset: is 1
k. Zone Offset: is -7

In addition, you can access your web application by starting your default web browser and point to :
http://localhost:8080/<ProjectName>

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**

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

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

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

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:

Make sure all the edited files have been saved

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

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.