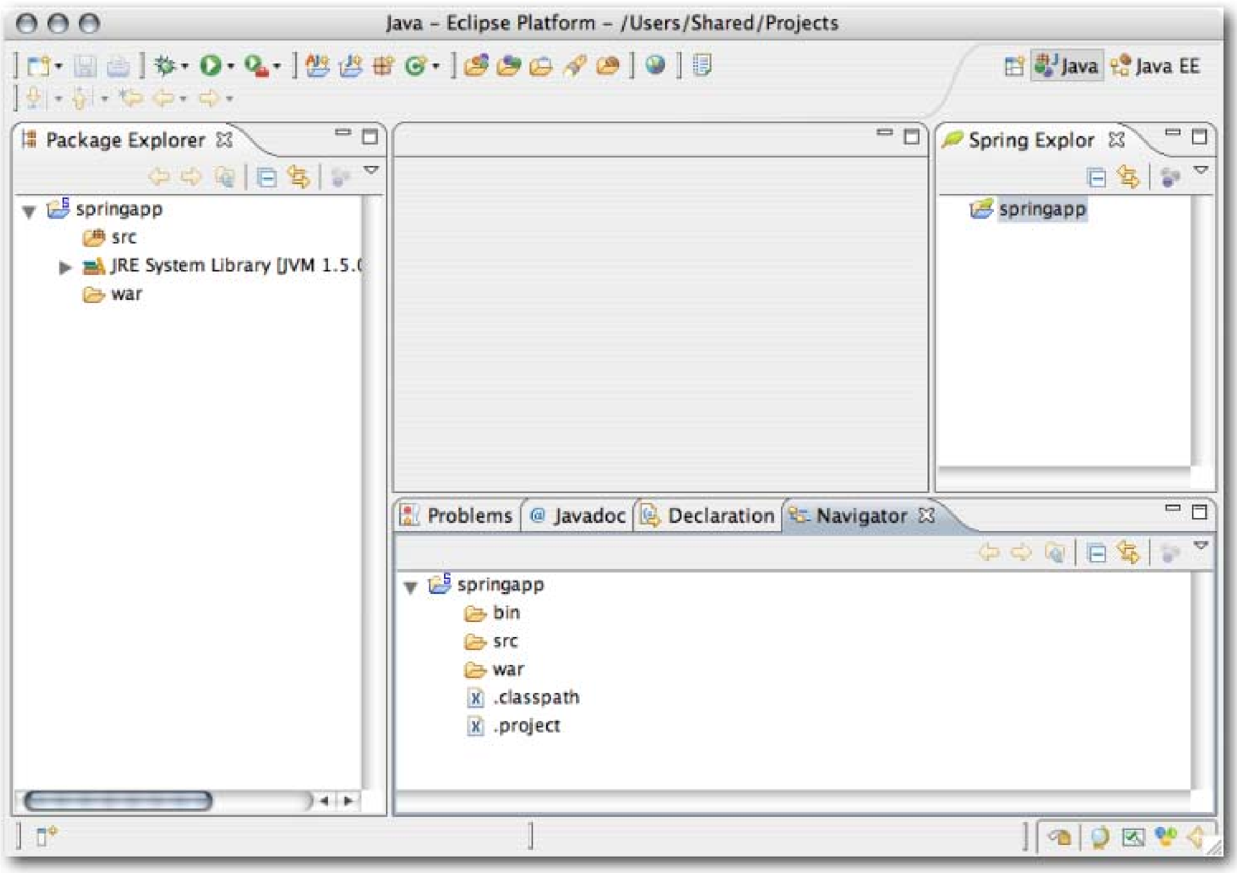
Chapter 1. Basic Application and Environment Setup

1.1. Create the project directory structure

We are going to need a place to keep all the source and other files we will be creating, so let's create a directory named **'springapp'**. The decision as to where you create this directory is totally up to you; we created ours in a **'Projects'** directory that we already had in our **'home'** directory so the complete path to our project directory is now **'$HOME/Projects/springapp'**. Inside this directory we create a sub-directory named **'src'** to hold all the Java source files that we are going to create. Then we create another sub-directory that we name **'war'**. This directory will hold everything that should go into the WAR file that we will use to package and deploy our application. All source files other than Java source, like JSPs and configuration files, belong in the **'war'** directory.

Find below a screen shot of what your project directory structure must look like after following the above instructions. *(The screen shot shows the project directory structure inside the Eclipse IDE: you do not need to use the Eclipse IDE to complete this tutorial successfully, but using Eclipse will make it much easier to follow along.)*



The project directory structure

1.2. Create 'index.jsp'

Since we are creating a web application, let's start by creating a very simple JSP page named **'index.jsp'** in the **'war'** directory. The **'index.jsp'** is the entry point for our application.

'springapp/war/index.jsp':

```
<html>
  <head><title>Example :: Spring Application</title></head>
  <body>
    <h1>Example - Spring Application</h1>
    <p>This is my test.</p>
  </body>
</html>
```

Just to have a complete web application, let's create a **'WEB-INF'** directory inside the **'war'** directory and place a **'web.xml'** file in this new directory.

'springapp/war/WEB-INF/web.xml':

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

  <welcome-file-list>
    <welcome-file>
      index.jsp
    </welcome-file>
  </welcome-file-list>

</web-app>
```

1.3. Deploy the application to Tomcat

Let's now write the Ant build script that we are going to use throughout the tutorial. This Ant build script will contain targets for compiling, building and deploying the application. A separate build script will be used for application server specific targets, such as targets for controlling the application under Tomcat.

'springapp/build.xml':

```
<?xml version="1.0"?>

<project name="springapp" basedir="." default="usage">
    <property file="build.properties"/>

    <property name="src.dir" value="src"/>
    <property name="web.dir" value="war"/>
    <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
    <property name="name" value="springapp"/>

    <path id="master-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <!-- We need the servlet API classes: -->
        <!--  * for Tomcat 5/6 use servlet-api.jar -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="servlet*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
    </path>

    <target name="usage">
        <echo message=""/>
        <echo message="${name} build file"/>
        <echo message="-----------------------------------"/>
        <echo message=""/>
        <echo message="Available targets are:"/>
        <echo message=""/>
        <echo message="build     --> Build the application"/>
        <echo message="deploy    --> Deploy application as directory"/>
        <echo message="deploywar --> Deploy application as a WAR file"/>
        <echo message="install   --> Install application in Tomcat"/>
        <echo message="reload    --> Reload application in Tomcat"/>
        <echo message="start     --> Start Tomcat application"/>
        <echo message="stop      --> Stop Tomcat application"/>
        <echo message="list      --> List Tomcat applications"/>
        <echo message=""/>
    </target>

    <target name="build" description="Compile main source tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
               deprecation="false" optimize="false" failonerror="true">
            <src path="${src.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <target name="deploy" depends="build" description="Deploy application">
        <copy todir="${deploy.path}/${name}" preservelastmodified="true">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </copy>
    </target>

    <target name="deploywar" depends="build" description="Deploy application as a WAR file">
        <war destfile="${name}.war"
             webxml="${web.dir}/WEB-INF/web.xml">
            <fileset dir="${web.dir}">
                <include name="**/*.*"/>
            </fileset>
        </war>
        <copy todir="${deploy.path}" preservelastmodified="true">
            <fileset dir=".">
                <include name="*.war"/>
            </fileset>
        </copy>
    </target>

<!-- ============================================================== -->
<!-- Tomcat tasks - remove these if you don't have Tomcat installed -->
<!-- ============================================================== -->

    <path id="catalina-ant-classpath">
        <!-- We need the Catalina jars for Tomcat -->
        <!--  * for other app servers - check the docs -->
        <fileset dir="${appserver.lib}">
            <include name="catalina-ant.jar"/>
        </fileset>
    </path>

    <taskdef name="install" classname="org.apache.catalina.ant.InstallTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="start" classname="org.apache.catalina.ant.StartTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>
    <taskdef name="stop" classname="org.apache.catalina.ant.StopTask">
        <classpath refid="catalina-ant-classpath"/>
    </taskdef>

    <target name="install" description="Install application in Tomcat">
        <install url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"
                 path="/${name}"
                 war="${name}"/>
    </target>

    <target name="reload" description="Reload application in Tomcat">
        <reload url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"
                 path="/${name}"/>
    </target>

    <target name="start" description="Start Tomcat application">
        <start url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"
                 path="/${name}"/>
    </target>

    <target name="stop" description="Stop Tomcat application">
        <stop url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"
                 path="/${name}"/>
    </target>

    <target name="list" description="List Tomcat applications">
        <list url="${tomcat.manager.url}"
                 username="${tomcat.manager.username}"
                 password="${tomcat.manager.password}"/>
    </target>

<!-- End Tomcat tasks -->

</project>
```

If you are using a different web application server, then you can remove the Tomcat specific tasks at the end of the build script. You will have to rely on your server's hot deploy feature, or you will have to stop and start your application manually.

*If you are using an IDE, you may find a number of errors reported by the IDE in the **'build.xml'** such as the Tomcat targets. You can ignore these. The file listing above is correct.*

The above Ant build script now contains all the targets that we are going to need to make our development efforts easier. We are not going to cover this script in detail, since most if not all of it is pretty much standard Ant and Tomcat stuff. You can just copy the above build file text and paste it into a new file called **'build.xml'** in the root of your development directory tree. We also need a **'build.properties'** file that you should customize to match your server installation. This file belongs in the same directory as the **'build.xml'** file.

'springapp/build.properties':

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret
```

*If you are on a system where you are not the owner of the Tomcat installation, then the Tomcat owner must either grant you full access to the **'webapps'** directory or the owner must create a new directory named **'springapp'** in the **'webapps'** directory of the Tomcat installation directory, and also give you full rights to deploy to this newly created directory. On Linux, run the command **'chmod a+rwx springapp'** to give everybody full rights to this directory.*

*To create Tomcat user named 'tomcat' with 's3cret' as their password, go to the Tomcat users file **'appserver.home/conf/tomcat-users.xml'** and add the user entry.*

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="s3cret" roles="manager"/>
</tomcat-users>
```

Now we run Ant to make sure that everything is working okay. You must have your current directory set to the **'springapp'** directory.

Open up a command shell (or prompt) and execute **'ant'** .

```
$ ant
Buildfile: build.xml

usage:
     [echo]
     [echo] springapp build file
     [echo] -----------------------------------
     [echo]
     [echo] Available targets are:
     [echo]
     [echo] build     --> Build the application
     [echo] deploy    --> Deploy application as directory
     [echo] deploywar --> Deploy application as a WAR file
     [echo] install   --> Install application in Tomcat
     [echo] reload    --> Reload application in Tomcat
     [echo] start     --> Start Tomcat application
     [echo] stop      --> Stop Tomcat application
     [echo] list      --> List Tomcat applications
     [echo]

BUILD SUCCESSFUL
Total time: 2 seconds
```

The last thing we need to do here is to build and deploy the application. Just run Ant and specify `'deploy'` or `'deploywar'` as the target.

```
$ ant deploy
Buildfile: build.xml

build:
    [mkdir] Created dir: /Users/trisberg/Projects/springapp/war/WEB-INF/classes

deploy:
     [copy] Copying 2 files to /Users/trisberg/apache-tomcat-5.5.17/webapps/springapp

BUILD SUCCESSFUL
Total time: 4 seconds
```

1.4. Check the application works

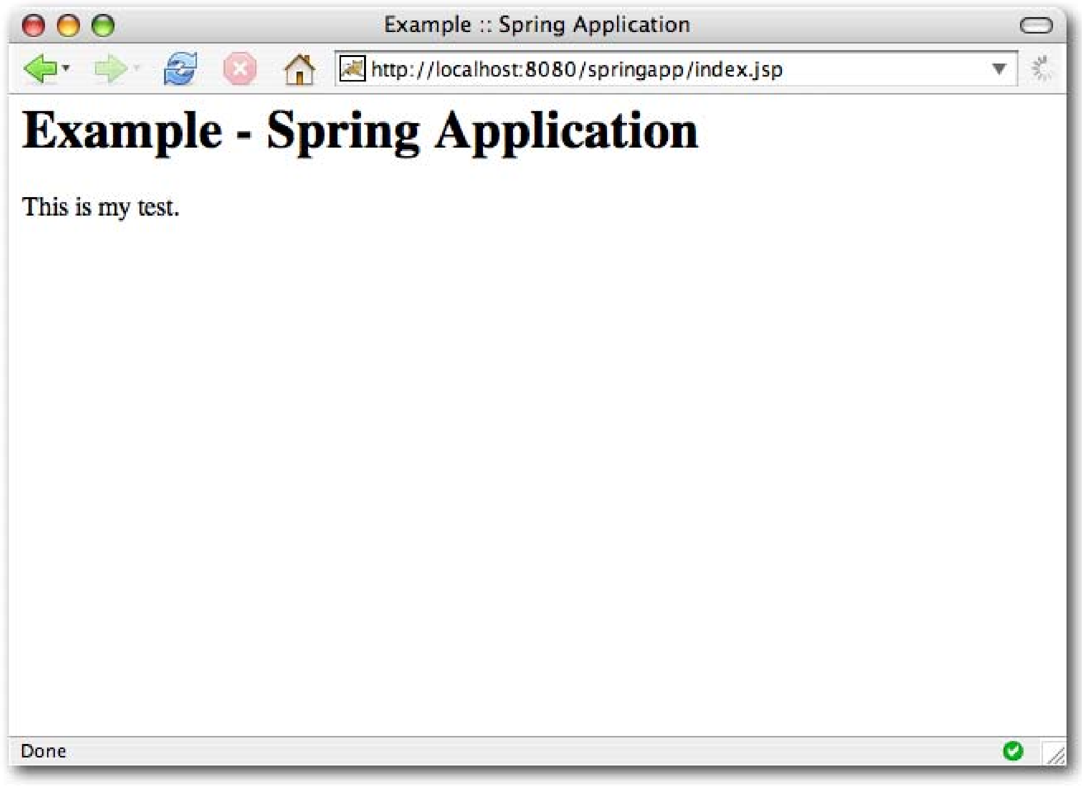
Let's just quickly start Tomcat by running **'${appserver.home}/bin/startup.bat'**. To make sure that we can access the application, run the `'list'` task from our build file to see if Tomcat has picked up the new application.

```
$ ant list
Buildfile: build.xml

list:
     [list] OK - Listed applications for virtual host localhost
     [list] /springapp:running:0:springapp
     [list] /manager:running:0:manager
     [list] /:running:0:ROOT
     [list] /docs:running:0:docs
     [list] /examples:running:0:examples
     [list] /host-manager:running:0:host-manager

BUILD SUCCESSFUL
Total time: 3 seconds
```

You can now open up a browser and navigate to the starting page of our application at the following URL: http://localhost:8080/springapp/index.jsp.


Example :: Spring Application
http://localhost:8080/springapp/index.jsp
Done

Example - Spring Application

This is my test.

The application's starting page

1.5. Download the Spring Framework

If you have not already downloaded the Spring Framework, now is the time to do so. We are currently using the 'Spring Framework 2.5' release that can be downloaded from http://www.springframework.org/download. Unzip this file somewhere as we are going to use several files from this download later on.

This completes the setup of the environment that is necessary, and now we can start actually developing our Spring Framework MVC application.

1.6. Modify 'web.xml' in the 'WEB-INF' directory

Go to the **'springapp/war/WEB-INF'** directory. Modify the minimal **'web.xml'** file that we created earlier. We will define a `DispatcherServlet` (also known as a `'Front Controller'` (Crupi et al)). It is going to control where all our requests are routed based on information we will enter at a later point. This servlet definition also has an attendant `<servlet-mapping/>` entry that maps to the URL patterns that we will be using. We have decided to let any URL with an `'.htm'` extension be routed to the `'springapp'` servlet (the `DispatcherServlet`).

'springapp/war/WEB-INF/web.xml':

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

  <servlet>
    <servlet-name>springapp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springapp</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
```

```
    <welcome-file>
      index.jsp
    </welcome-file>
  </welcome-file-list>

</web-app>
```

Next, create a file called **'springapp-servlet.xml'** in the **'springapp/war/WEB-INF'** directory. This file contains the bean definitions (plain old Java objects) used by the `DispatcherServlet`. It is the `WebApplicationContext` where all web-related components go. The name of this file is determined by the value of the `<servlet-name/>` element from the **'web.xml'**, with `'-servlet'` appended to it (hence **'springapp-servlet.xml'**). This is the standard naming convention used with Spring's Web MVC framework. Now, add a bean entry named `'/hello.htm'` and specify the class as `springapp.web.HelloController`. This defines the controller that our application will be using to service a request with the corresponding URL mapping of `'/hello.htm'`. The Spring Web MVC framework uses an implementation class of the interface called `HandlerMapping` to define the mapping between a request URL and the object that is going to handle that request (the handler). Unlike the `DispatcherServlet`, the `HelloController` is responsible for handling a request for a particular page of the website and is also known as a `'Page Controller'` (Fowler). The default `HandlerMapping` that the `DispatcherServlet` uses is the `BeanNameUrlHandlerMapping`; this class will use the bean name to map to the URL in the request so that the `DispatcherServlet` knows which controller must be invoked for handling different URLs.

'springapp/war/WEB-INF/springapp-servlet.xml':

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <!-- the application context definition for the springapp DispatcherServlet -->

  <bean name="/hello.htm" class="springapp.web.HelloController"/>

</beans>
```

1.7. Copy libraries to 'WEB-INF/lib'

First create a **'lib'** directory in the **'war/WEB-INF'** directory. Then, from the Spring distribution, copy **spring.jar** (from **spring-framework-2.5/dist**) and **spring-webmvc.jar** (from **spring-framework-2.5/dist/modules**) to the new **'war/WEB-INF/lib'** directory. Also, copy **commons-logging.jar** (from **spring-framework-2.5/lib/jakarta-commons**) to the **'war/WEB-INF/lib'** directory. These jars will be deployed to the server and they are also used during the build process.

1.8. Create the Controller

Create your `Controller` class – we are naming it `HelloController`, and it is defined in the `'springapp.web'` package. First we create the package directories and then we create the **'HelloController.java'** file and place it in the **'src/springapp/web'** directory.

'springapp/src/springapp/web/HelloController.java':

```
package springapp.web;

import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.io.IOException;
```

```
public class HelloController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        logger.info("Returning hello view");

        return new ModelAndView("hello.jsp");
    }

}
```

This is a very basic `Controller` implementation. We will be expanding this later on as well as extending some of the base controller implementations provided by Spring. In Spring Web MVC, the `Controller` *handles* the request and returns a `ModelAndView` - in this case, one named **'hello.jsp'** which is also the name of the JSP file we will create next. The model that this class returns is actually resolved via a `ViewResolver`. Since we have not explicitly defined a `ViewResolver`, we are going to be given a default one by Spring that simply forwards to a URL matching the name of the view specified. We will modify this later on. We have also specified a logger so we can verify that we actually got into the handler. Using Tomcat, these log messages should show up in the **'catalina.out'** log file which can be found in the **'${appserver.home}/log'** directory of your Tomcat installation.

*If you are using an IDE, you will want to configure your project's build path by adding the jars from the **'lib'** directory. You will also want to add **servlet-api.jar** from your servlet container's **'lib'** directory ('`${appserver.lib}`'). Adding these to your build path should successfully resolve all the import statements in the **'HelloController.java'** file.*

1.9. Write a test for the `Controller`

Testing is a vital part of software development. It is also a core practice in Agile development. We have found that the best time to write tests is during development, not after, so even though our controller doesn't contain complex logic, we're going to write a test. This will allow us to make changes to it in the future with confidence. Let's create a new directory under **'springapp'** called **'test'**. This is where all our tests will go in a package structure that will mirror the source tree in **'springapp/src'**.

Create a test class called **'HelloControllerTests'** and make it extend JUnit's test class `TestCase`. It is a unit test that verifies the view name returned by `handleRequest()` matches the name of the view we expect: **'hello.jsp'**.

'springapp/test/springapp/web/HelloControllerTests.java':

```
package springapp.web;

import org.springframework.web.servlet.ModelAndView;

import springapp.web.HelloController;

import junit.framework.TestCase;

public class HelloControllerTests extends TestCase {

    public void testHandleRequestView() throws Exception{
        HelloController controller = new HelloController();
        ModelAndView modelAndView = controller.handleRequest(null, null);
        assertEquals("hello.jsp", modelAndView.getViewName());
    }
}
```

To run the test (and all the tests we're going to write), we need to add an Ant test task to our build script **'build.xml'**. First, we copy the **junit-3.8.2.jar** from **'spring-framework-2.5/lib/junit'** to **'war/WEB-INF/lib'**. Instead of creating a single task for compiling the tests and then running them, let's break them down into two distinct tasks: '`buildtests`' and '`tests`' which depends on '`buildtests`'.

If you are using an IDE, you may want to run your tests within your IDE. Configure your *project's build path by adding the* ***`junit-3.8.2.jar`*** *to it.*

`'springapp/build.xml'`:

```
    <property name="test.dir" value="test"/>

    <target name="buildtests" description="Compile test tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
            deprecation="false" optimize="false" failonerror="true">
            <src path="${test.dir}"/>
            <classpath refid="master-classpath"/>
        </javac>
    </target>

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="master-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ***********************************************************
            ***********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ***********************************************************
            ***********************************************************
        </fail>
    </target>
```

Now run the Ant `'tests'` task and the test should pass.

```
$ ant tests
Buildfile: build.xml

build:

buildtests:
    [javac] Compiling 1 source file to /Users/Shared/Projects/springapp/war/WEB-INF/classes

tests:
    [junit] Running springapp.web.HelloWorldControllerTests
    [junit] Oct 30, 2007 11:31:43 PM springapp.web.HelloController handleRequest
    [junit] INFO: Returning hello view
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.03 sec
    [junit] Testsuite: springapp.web.HelloWorldControllerTests
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.03 sec

    [junit] ------------- Standard Error -----------------
    [junit] Oct 30, 2007 11:31:43 PM springapp.web.HelloController handleRequest
    [junit] INFO: Returning hello view
    [junit] ------------- ---------------- ---------------

BUILD SUCCESSFUL
Total time: 2 seconds
```

Another of the best practices of Agile development is *Continuous Integration*. It's a good idea to ensure your tests are run with every build (ideally as automated project builds) so that you know your application logic is behaving as expected as the code evolves.

1.10. Create the View

Now it is time to create our first view. As we mentioned earlier, we are forwarding to a JSP page named **'hello.jsp'**. To begin with, we'll put it in the **'war'** directory.

'springapp/war/hello.jsp':

```
<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
    <p>Greetings.</p>
  </body>
</html>
```

1.11. Compile and deploy the application

Run the 'deploy' Ant target (which invokes the 'build' target), and then run the 'reload' task of the **'build.xml'** file. This will force a build and reload of the application in Tomcat. We have to check the Ant output and the Tomcat logs for any possible deployment errors – such as typos in the above files or missing classes or jar files.

Here is a sample output from the Ant build:

```
$ ant deploy reload
Buildfile: build.xml

build:
    [mkdir] Created dir: /Users/trisberg/Projects/springapp/war/WEB-INF/classes
    [javac] Compiling 1 source file to /Users/trisberg/Projects/springapp/war/WEB-INF/classes

deploy:
     [copy] Copying 7 files to /Users/trisberg/apache-tomcat-5.5.17/webapps/springapp

BUILD SUCCESSFUL
Total time: 3 seconds
$ ant reload
Buildfile: build.xml

reload:
   [reload] OK - Reloaded application at context path /springapp

BUILD SUCCESSFUL
Total time: 2 seconds
```

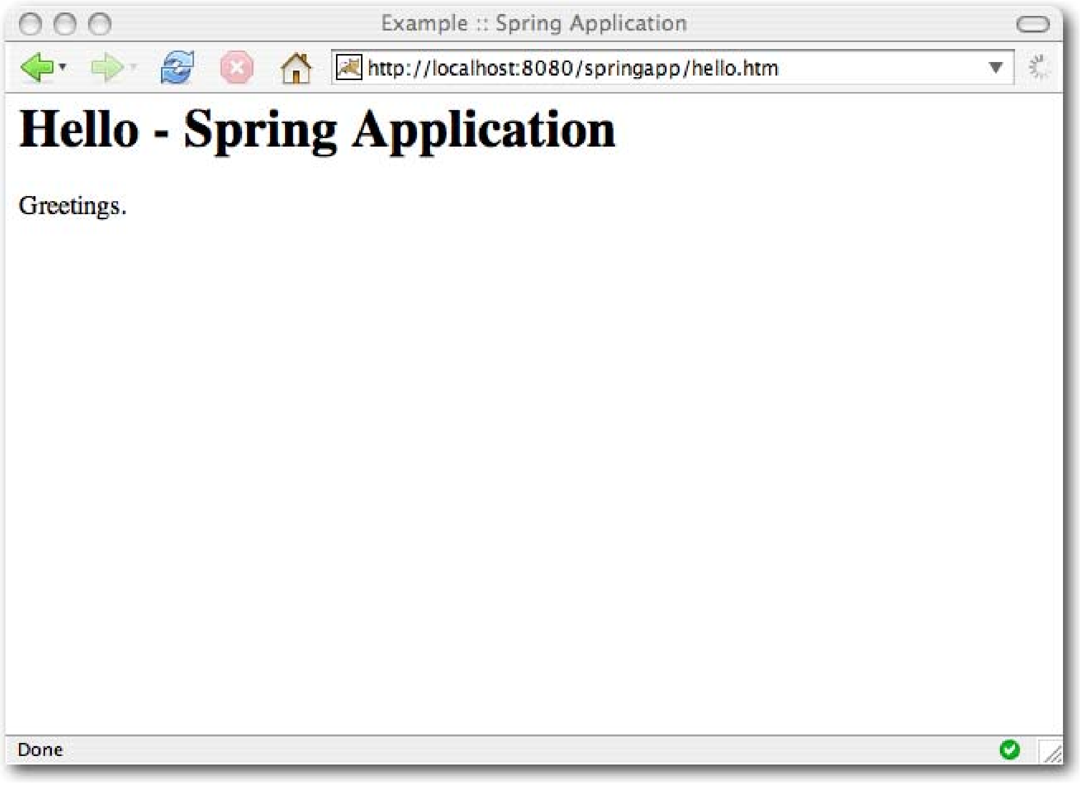
And here is an excerpt from the Tomcat **'catalina.out'** log file.

```
Oct 30, 2007 11:43:09 PM org.springframework.web.servlet.FrameworkServlet initServletBean
INFO: FrameworkServlet 'springapp': initialization started
Oct 30, 2007 11:43:09 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.XmlWebApplicationContext@6576d5: display name
[WebApplicationContext for namespace 'springapp-servlet']; startup date [Tue Oct 30 23:43:09 GMT 2007];
...
...
Oct 30, 2007 11:43:09 PM org.springframework.web.servlet.FrameworkServlet initServletBean
INFO: FrameworkServlet 'springapp': initialization completed in 150 ms
```

1.12. Try out the application

Let's try this new version of the application.

Open a browser and browse to http://localhost:8080/springapp/hello.htm.


Example :: Spring Application
http://localhost:8080/springapp/hello.htm
Done

Hello - Spring Application

Greetings.

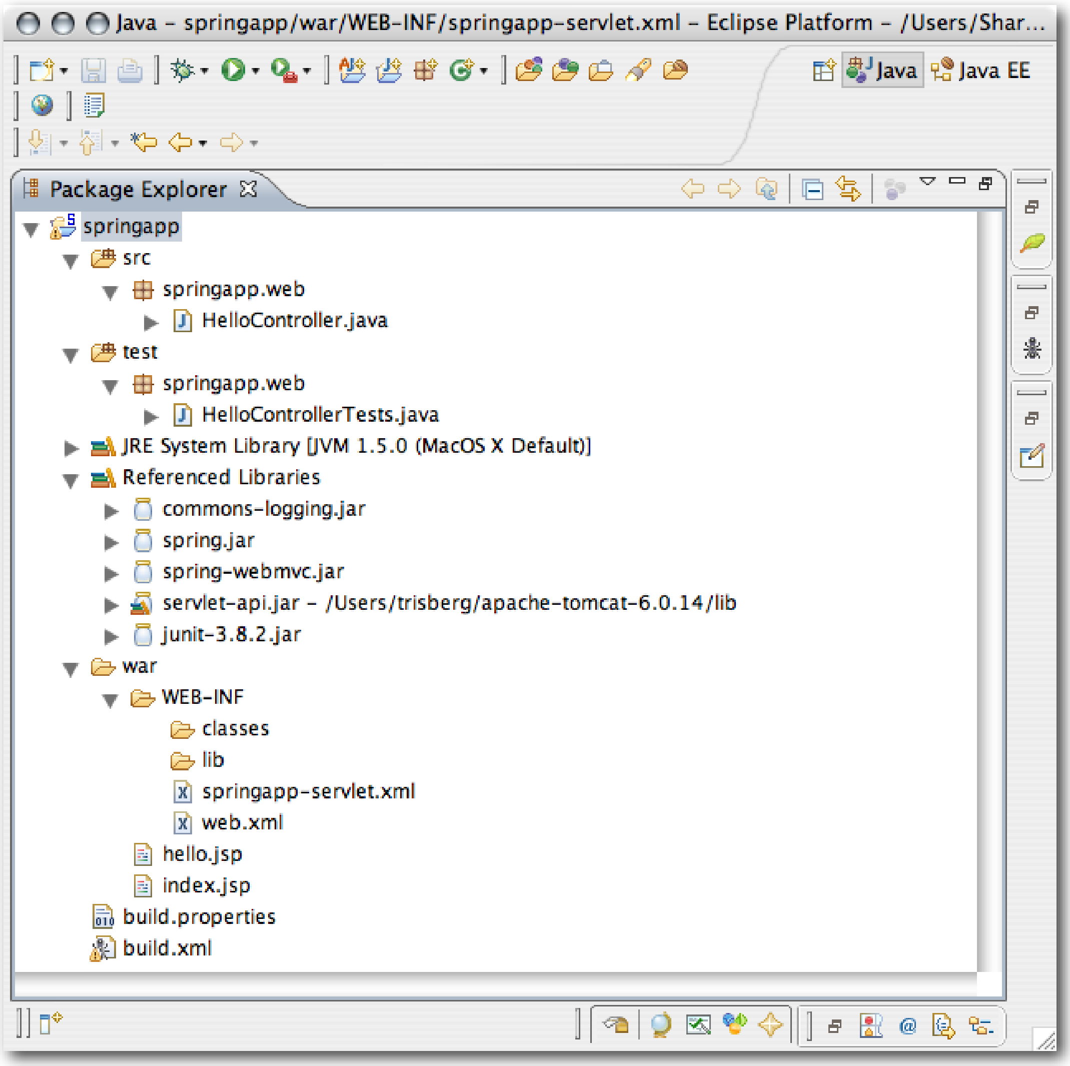
The updated application

1.13. Summary

Let's take quick look at the parts of our application that we have created so far.

- An introduction page, **'index.jsp'**, the welcome page of the application. It was used to test our setup was correct. We will later change this to actually provide a link into our application.
- A `DispatcherServlet` (front controller) with a corresponding **'springapp-servlet.xml'** configuration file.
- A page controller, `HelloController`, with limited functionality – it just returns a `ModelAndView`. We currently have an empty model and will be providing a full model later on.
- A unit test class for the page controller, `HelloControllerTests`, to verify the name of the view is the one we expect.
- A view, **'hello.jsp'**, that again is extremely basic. The good news is the whole setup works and we are now ready to add more functionality.

Find below a screen shot of what your project directory structure must look like after following the above instructions.


Java - springapp/war/WEB-INF/springapp-servlet.xml - Eclipse Platform - /Users/Shar…

```
springapp
  src
    springapp.web
      HelloController.java
  test
    springapp.web
      HelloControllerTests.java
  JRE System Library [JVM 1.5.0 (MacOS X Default)]
  Referenced Libraries
    commons-logging.jar
    spring.jar
    spring-webmvc.jar
    servlet-api.jar - /Users/trisberg/apache-tomcat-6.0.14/lib
    junit-3.8.2.jar
  war
    WEB-INF
      classes
      lib
      springapp-servlet.xml
      web.xml
    hello.jsp
    index.jsp
  build.properties
  build.xml
```

The project directory structure at the end of part 1

Prev | Home | Next
Overview | Sponsored by SpringSource | Chapter 2. Developing and Configuring the Views and the Controller