

# Developing a Spring Framework MVC application step-by-step

**Thomas Risberg**

July, 2003

(Revised April, 2005)

This is a step-by-step account of how to develop a web application from scratch using the Spring Framework. It is divided into a number of parts. You can read them in sequence or skip one if you are already familiar with the concepts covered in it.

[Part 1 - Basic Application and Environment Setup](#)

[Part 2 – Developing and Configuring the Application](#)

[Part 3 – Adding Unit Tests and a Form to the Application](#)

[Part 4 – Implementing Database Persistence](#)

Copyright © 2003-2005, Thomas Risberg

# Developing a Spring Framework MVC application step-by-step

Part 1 – Basic Application and Environment Setup

Thomas Risberg  
July, 2003

(Revised April, 2005)

This is a step-by-step account of how to develop a web application from scratch using the Spring Framework.

Prerequisites:

- Java SDK (*I am currently using version 1.4.2*)
- Ant (*using version 1.6.2*)
- Apache Tomcat (*using version 5.0.28*)

You should also be reasonably comfortable using the above software.

I am not going to cover a lot of background information or theory in this document -- there are plenty of books available that covers this in depth. Instead we will dive right into developing the application.

## Step 1 – development directory

We are going to need a place to keep all the source and other files we will be creating, so I create a directory that I name 'springapp'. You can place this directory in your home folder or in some other location. I created mine in a 'projects' directory that I already had in my home directory so the full path to my directory is '/Users/trisberg/projects/springapp'. Inside this directory I create a 'src' directory to hold all Java source files. Then I create another directory that I name 'war'. This directory will hold everything that should go into the WAR file, that we would use to deploy our application. All source files other than Java source, like JSPs and configuration files, belongs in this directory.

## Step 2 – index.jsp

I will start by creating a JSP page named 'index.jsp' in the war directory. This 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, I create a web.xml in a WEB-INF directory that I create under the war directory.

springapp/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
</web-app>
```

### Step 3 – deploying the application to Tomcat

Next, I write an Ant build script that we are going to use throughout this document. There are tasks for building and deploying the application. A separate build script contains the app server specific tasks There are also tasks 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 4.1 use servlet.jar      -->
    <!--   for Tomcat 5.0 use servlet-api.jar  -->
    <!--   for Other app server - check the docs -->
    <fileset dir="${appserver.home}/common/lib">
      <include name="servlet*.jar"/>
    </fileset>
    <pathelement path="${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="${build.dir}" target="1.3" debug="true"
            deprecation="false" optimize="false" failonerror="true">
            <src path="${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="${deploy.path}" preservelastmodified="true">
            <fileset dir=".">
                <include name="*.war"/>
            </fileset>
        </copy>
    </target>
<!-- ===== -->
<!-- Tomcat tasks - remove these if you don't have Tomcat installed -->
<!-- ===== -->
    <taskdef name="install" classname="org.apache.catalina.ant.InstallTask">
        <classpath>
            <path location="${appserver.home}/server/lib/catalina-ant.jar"/>
        </classpath>
    </taskdef>
    <taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask">
        <classpath>
            <path location="${appserver.home}/server/lib/catalina-ant.jar"/>
        </classpath>
    </taskdef>
    <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
        <classpath>
            <path location="${appserver.home}/server/lib/catalina-ant.jar"/>
        </classpath>
    </taskdef>
    <taskdef name="start" classname="org.apache.catalina.ant.StartTask">
        <classpath>
            <path location="${appserver.home}/server/lib/catalina-ant.jar"/>
        </classpath>
    </taskdef>
    <taskdef name="stop" classname="org.apache.catalina.ant.StopTask">
        <classpath>
            <path location="${appserver.home}/server/lib/catalina-ant.jar"/>
        </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>

```

This script now contains all the targets that we are going to need to make our development efforts easier. I am 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 and put it at 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=${user.home}/jakarta-tomcat-5.0.28
deploy.path=${appserver.home}/webapps
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=tomcat

```

*If you are on a system where you are not the owner of the Tomcat install, 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, and also give you full rights to deploy to this newly created directory. On Linux I run the command **chmod a+rx springapp** to give everybody full rights to this directory.*

*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.*

## Spring MVC step-by-step Part 1

Now I run Ant to make sure that everything is working OK. You should have your current directory set to the 'springapp' directory.

```
[trisberg@localhost springapp]$ ant
Buildfile: build.xml

usage:

    [echo] springapp build file
    [echo] -----

    [echo] Available targets are:

    [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

BUILD SUCCESSFUL
Total time: 2 seconds
```

Last action here is to do the actual deployment. Just run Ant and specify 'deploy' or 'deploywar' as the target.

```
[trisberg@localhost springapp]$ 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/jakarta-tomcat-
5.0.28/webapps/springapp
BUILD SUCCESSFUL
Total time: 2 seconds
```

### Step 4 – Test the application

Let's just quickly start Tomcat and make sure that we can access the application. Use the 'list' task from our build file to see if Tomcat has picked up the new application.

```
[trisberg@localhost springapp]$ ant list
Buildfile: build.xml

list:
    [list] OK - Listed applications for virtual host localhost
    [list] /admin:running:0:/Users/trisberg/jakarta-tomcat-
5.0.28/server/webapps/admin
    [list] /webdav:running:0:/Users/trisberg/jakarta-tomcat-5.0.28/webapps/webdav
    [list] /servlets-examples:running:0:/Users/trisberg/jakarta-tomcat-
5.0.28/webapps/servlets-examples
    [list] /springapp:running:0:/Users/trisberg/jakarta-tomcat-
5.0.28/webapps/springapp
    [list] /jsp-examples:running:0:/Users/trisberg/jakarta-tomcat-5.0.28/webapps/jsp-
examples
    [list] /balancer:running:0:balancer
    [list] /tomcat-docs:running:0:/Users/trisberg/jakarta-tomcat-
5.0.28/webapps/tomcat-docs
    [list] /:running:0:/Users/trisberg/jakarta-tomcat-5.0.28/webapps/ROOT
    [list] /manager:running:0:/Users/trisberg/jakarta-tomcat-
5.0.28/server/webapps/manager

BUILD SUCCESSFUL
Total time: 1 second
```

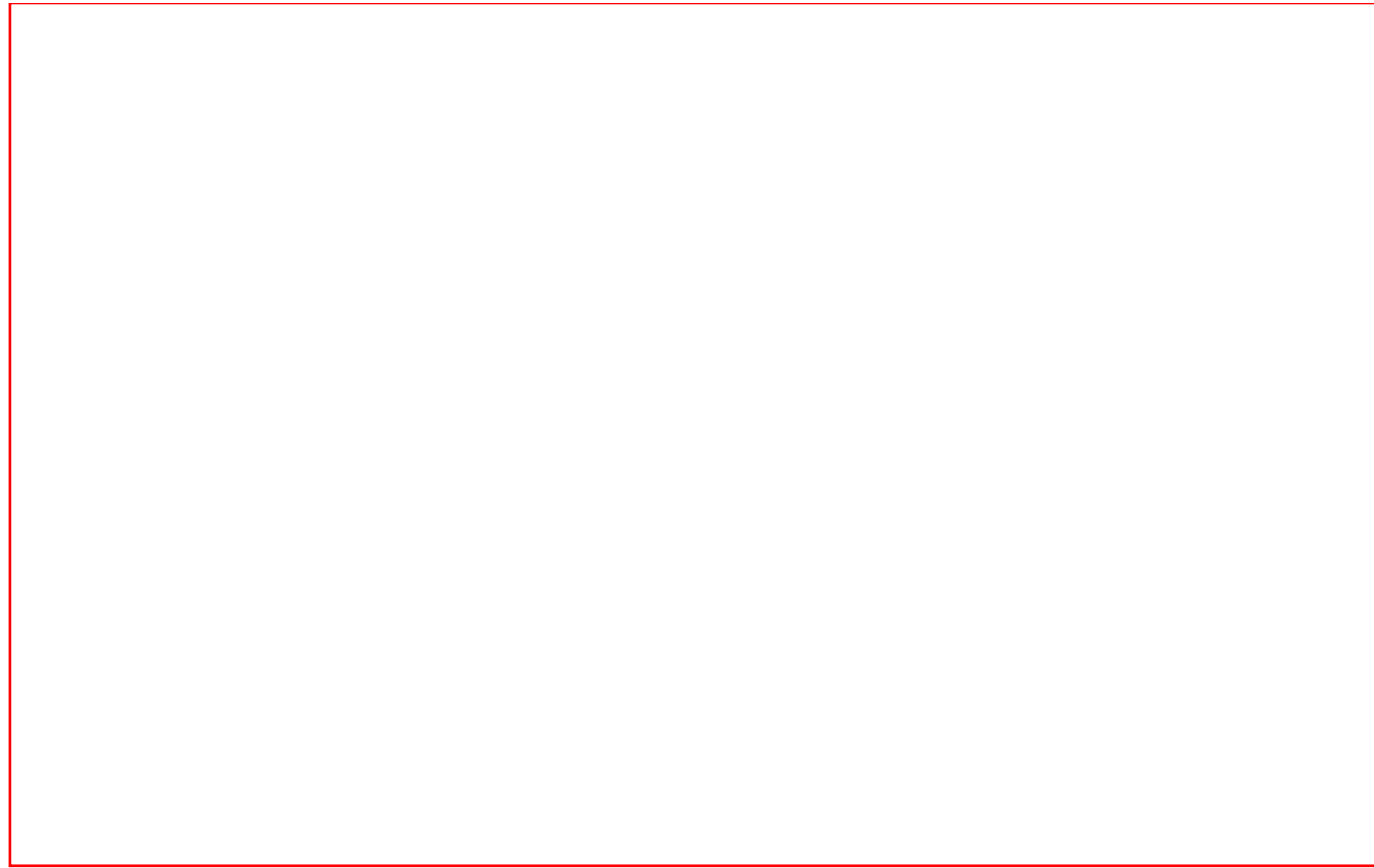
If it is not listed, use the 'install' task to get the application installed in Tomcat.

```
[trisberg@localhost springapp]$ ant install
Buildfile: build.xml

install:
    [install] OK - Installed application at context path /springapp

BUILD SUCCESSFUL
Total time: 2 seconds
```

Now open a browser and browse to <http://localhost:8080/springapp/index.jsp>.



#### Step 5 – Download Spring distribution

If you have not already downloaded the Spring Framework Release file, now is the time to do so. I am currently using 'spring-framework-1.2-with-dependencies.zip' that can be downloaded from [www.springframework.org/download.html](http://www.springframework.org/download.html). I unzipped this file in my home directory. 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.**

#### Step 6 – Modify web.xml in WEB-INF directory

Go to the 'springapp/war/ WEB-INF' directory. Modify the minimal 'web.xml' file that we created earlier. Now we will modify it to suit our needs. We define a DispatcherServlet that is going to control where all our request are routed based on information we will enter at a later point. It also has a standard servlet-mapping entry that maps to the url patterns that we will be using. I have decided to let any url with an '.htm' extension



be routed to the 'springapp' dispatcher.

#### springapp/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
  <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 (you can copy an example of this file from the Spring distributions sample/skeletons/webapp-minimal directory). This is the file where definitions used by the DispatcherServlet should be entered. It is named based on the servlet-name from web.xml with '-servlet' appended. This is a standard naming convention used in the Spring Framework. Now, add a bean entry named springappController and make the class SpringappController. This defines the controller that our application will be using. We also need to add a url mapping so the DispatcherServlet knows which controller should be invoked for different url:s.

#### springapp/war/WEB-INF/springapp-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!--
  - Application context definition for "springapp" DispatcherServlet.
-->
<beans>
  <bean id="springappController" class="SpringappController"/>
  <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/hello.htm">springappController</prop>
      </props>
    </property>
  </bean>
</beans>
```

**Step 7 – Copy jars to WEB-INF/lib**

First create a 'lib' directory in the 'war/WEB-INF' directory. Then, from the Spring distribution, copy spring.jar (spring-framework-1.2/dist/spring.jar) to the new war/WEB-INF/lib directory. Also copy commons-logging jars to the war/WEB-INF/lib directory (spring-framework-1.2/lib/jakarta-commons/commons-logging.jar). We are also going to need a log4j jar. Copy log4j-1.2.9.jar to the war/WEB-INF/lib directory (spring-framework-1.2/lib/log4j/log4j-1.2.9.jar). These jars will be deployed to the server and they are also used during the build process.

**Step 8 – Create your Controller**

Create your Controller – I named mine SpringappController.java and placed it in the springapp/src directory.

```
springapp/src/SpringappController.java

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 java.io.IOException;

public class SpringappController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        return new ModelAndView("");
    }
}
```

This is as basic a Controller as you can use. We will be expanding this later on, and we will also later on extend some provided abstract base implementations. The Controller “handles” the request and returns a ModelAndView. We have not yet defined any Views, so right now there is nothing to do.

**Step 9 – Build the Application**

Run the 'build' task of the build.xml. Hopefully the code compiles OK.

```
[trisberg@localhost springapp]$ ant build
Buildfile: build.xml

build:
    [javac] Compiling 1 source file to /Users/trisberg/projects/springapp/war/WEB-
INF/classes

BUILD SUCCESSFUL
Total time: 2 seconds
```

**Step 10 – Copy and modify log4j.properties**

The Spring Framework uses log4j for logging so we have to create a configuration file for log4j. Copy the log4j.properties from the sample Petclinic application (spring-framework-1.2/samples/petclinic/war/WEB-INF/log4j.properties) to the war/WEB-INF/classes directory (this directory should have been created in the previous step). Now uncomment or modify the log4j.rootCategory property and change the name and location

of the logfile that will be written. I decided to have it written to the same directory as all other Tomcat logs.

springapp/war/WEB-INF/classes/log4j.properties

```
# For JBoss: Avoid to setup Log4J outside
$JBoss_HOME/server/default/deploy/log4j.xml!
# For all other servers: Comment out the Log4J listener in web.xml to activate Log4J.
log4j.rootLogger=INFO, stdout, logfile
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <m>%n
log4j.appender.logfile=org.apache.log4j.RollingFileAppender
log4j.appender.logfile.File=/Users/trisberg/jakarta-tomcat-5.0.28/logs/springapp.log
log4j.appender.logfile.MaxFileSize=512KB
# Keep three backup files.
log4j.appender.logfile.MaxBackupIndex=3
# Pattern to output: date priority [category] - message
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
```

## Step 11 – Deploy Application

Run the 'deploy' task and then the 'stop' and 'start' tasks of the build.xml. This will force a reload of the application. We have to check the Tomcat logs for any deployment errors – there could be typos in the above xml files or there could be missing classes or jar files. This is an example of what it should look like. (/Users/trisberg/jakarta-tomcat-5.0.28/logs/springapp.log)

```
2005-04-24 14:58:18,112 INFO [org.springframework.web.servlet.DispatcherServlet] -
Initializing servlet 'springapp'
2005-04-24 14:58:18,261 INFO [org.springframework.web.servlet.DispatcherServlet] -
FrameworkServlet 'springapp': initialization started
2005-04-24 14:58:18,373 INFO
[org.springframework.beans.factory.xml.XmlBeanDefinitionReader] - Loading XML bean
definitions from ServletContext resource [/WEB-INF/springapp-servlet.xml]
2005-04-24 14:58:18,498 INFO
[org.springframework.web.context.support.XmlWebApplicationContext] - Bean factory for
application context [WebApplicationContext for namespace 'springapp-servlet']:
org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
[springappController,urlMapping]; root of BeanFactory hierarchy
2005-04-24 14:58:18,505 INFO
[org.springframework.web.context.support.XmlWebApplicationContext] - 2 beans defined
in application context [WebApplicationContext for namespace 'springapp-servlet']
2005-04-24 14:58:18,523 INFO [org.springframework.core.CollectionFactory] - JDK 1.4+
collections available
2005-04-24 14:58:18,524 INFO [org.springframework.core.CollectionFactory] - Commons
Collections 3.x available
2005-04-24 14:58:18,537 INFO
[org.springframework.web.context.support.XmlWebApplicationContext] - Unable to locate
MessageSource with name 'messageSource': using default
[org.springframework.context.support.DelegatingMessageSource@8dacb]
2005-04-24 14:58:18,539 INFO
[org.springframework.web.context.support.XmlWebApplicationContext] - Unable to locate
ApplicationEventMulticaster with name 'applicationEventMulticaster': using default
[org.springframework.context.event.SimpleApplicationEventMulticaster@5674a4]
2005-04-24 14:58:18,549 INFO
```

## Spring MVC step-by-step Part 1

```
[org.springframework.ui.context.support.UiApplicationContextUtils] - No ThemeSource
found for [WebApplicationContext for namespace 'springapp-servlet']: using
ResourceBundleThemeSource
2005-04-24 14:58:18,556 INFO
[org.springframework.beans.factory.support.DefaultListableBeanFactory] - Pre-
instantiating singletons in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
[springappController,urlMapping]; root of BeanFactory hierarchy]
2005-04-24 14:58:18,557 INFO
[org.springframework.beans.factory.support.DefaultListableBeanFactory] - Creating
shared instance of singleton bean 'springappController'
2005-04-24 14:58:18,603 INFO
[org.springframework.beans.factory.support.DefaultListableBeanFactory] - Creating
shared instance of singleton bean 'urlMapping'
2005-04-24 14:58:18,667 INFO [org.springframework.web.servlet.DispatcherServlet] -
Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for servlet
'springapp'
2005-04-24 14:58:18,668 INFO [org.springframework.web.servlet.DispatcherServlet] -
Unable to locate MultipartResolver with name 'multipartResolver': no multipart
request handling provided
2005-04-24 14:58:18,670 INFO [org.springframework.web.servlet.DispatcherServlet] -
Unable to locate LocaleResolver with name 'localeResolver': using default
[org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@318309]
2005-04-24 14:58:18,675 INFO [org.springframework.web.servlet.DispatcherServlet] -
Unable to locate ThemeResolver with name 'themeResolver': using default
[org.springframework.web.servlet.theme.FixedThemeResolver@c11e94]
2005-04-24 14:58:18,681 INFO [org.springframework.web.servlet.DispatcherServlet] - No
HandlerAdapters found in servlet 'springapp': using default
2005-04-24 14:58:18,700 INFO [org.springframework.web.servlet.DispatcherServlet] - No
ViewResolvers found in servlet 'springapp': using default
2005-04-24 14:58:18,700 INFO [org.springframework.web.servlet.DispatcherServlet] -
FrameworkServlet 'springapp': initialization completed in 439 ms
2005-04-24 14:58:18,704 INFO [org.springframework.web.servlet.DispatcherServlet] -
Servlet 'springapp' configured successfully
```

### Step 12 – Create a View

Now it is time to create our first view. I will use a JSP page that I decided to name hello.jsp. I'll put it in the war directory to begin with.

springapp/war/hello.jsp

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

Nothing fancy here, but it will do for now. Next we have to modify the SpringappController to forward to this view.

springapp/src/SpringappController.java

```
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 java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class SpringappController implements Controller {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

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

        logger.info("SpringappController - returning hello view");

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

While I was modifying this class, I also added a logger so we can verify that we actually got here. Changes are highlighted in **red**. The model that this class returns is actually resolved via a ViewResolver. Since we have not specified a specific one, we are going to get a default one that just forwards to a url matching the name of the view specified. We will modify this later on.

Now compile and deploy the application. After instructing Tomcat to stop and then start the application, everything should get reloaded.

Let's try it in a browser – enter the url <http://localhost:8080/springapp/hello.htm> and we should see the following:



We can also check the log – I'm only showing the last entries, but we can see that the controller did get invoked and that it forwarded to the hello view. (`/Users/trisberg/jakarta-tomcat-5.0.28/logs/springapp.log`)

```
2005-04-24 15:01:56,217 INFO [org.springframework.web.servlet.DispatcherServlet] -  
FrameworkServlet 'springapp': initialization completed in 372 ms  
2005-04-24 15:01:56,217 INFO [org.springframework.web.servlet.DispatcherServlet] -  
Servlet 'springapp' configured successfully  
2005-04-24 15:03:57,908 INFO [SpringappController] - SpringappController - returning  
hello view
```

## Summary

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

1. An introduction page **index.jsp** that does not do anything useful. It was just used to test our setup. We will later change this to actually provide a link into our application.
2. A DispatcherServlet with a corresponding **springapp-servlet.xml** configuration file.
3. A controller **springappController.java** with limited functionality – it just forwards a ModelAndView to the ViewResolver. Actually, we only have an empty model so far, but we will fix this later.

4. A view **hello.jsp** that again is extremely basic. But the whole setup works and we are now ready to add more functionality.

[Back Part 2 – Developing and Configuring the Application](#)

Copyright © 2003-2005, Thomas Risberg

# Developing a Spring Framework MVC application step-by-step

## Part 2 – Developing and Configuring the Application

**Thomas Risberg**

July, 2003

(Revised April, 2005)

This is Part 2 of a step-by-step account of how to develop a web application from scratch using the Spring Framework. In Part 1 (Steps 1 – 12) we configured the environment and set up a basic application that we will build upon.

This is what we have to start with.

1. An introduction page **index.jsp**.
2. A DispatcherServlet with a corresponding **springapp-servlet.xml** configuration file.
3. A controller **springappController.java**.
4. A view **hello.jsp**.

We will now improve on these parts to build a more useful application.

### Step 13 – Improve index.jsp

We will make use of JSP Standard Tag Library (JSTL) so I will start by copying the JSTL files we need to our WEB-INF/lib directory. Copy jstl.jar from the 'spring-framework-1.2/lib/j2ee' directory and standard.jar from the 'spring-framework-1.2/lib/jakarta-taglibs' directory to the springapp/war/WEB-INF/lib directory. I am also creating a “header” file that will be included in every JSP page that I'm going to write. This will make development easier and I will be sure that I have the same definitions in all JSPs. I am going to put all JSPs in a directory named jsp under the WEB-INF directory. This will ensure that only the controller has access to the views - it is not possible to get to these pages by entering them directly as a URL in the browser. This strategy might not work in all application servers and if this is the case with the one you are using, just move the jsp directory up a level. You would then use springapp/war/jsp as the directory instead of springapp/war/WEB-INF/jsp in all the code examples that will follow.

**springapp/war/WEB-INF/jsp/include.jsp**

```
<%@ page session="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
```



Now we can change index.jsp to use this include and since we are using JSTL we can use the `<c:redirect>` tag for redirecting to our Controller. This ties the index.jsp into our application framework.

**springapp/war/index.jsp**

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<!-- Redirected because we can't set the welcome page to a virtual URL. --%>
<c:redirect url="/hello.htm"/>
```

**Step 14 – Improve the view and the controller**

I am going to move the view hello.jsp to the WEB-INF/jsp directory. The same include that was added to index.jsp gets added to hello.jsp. I also add the current date and time as output that I will retrieve from the model, passed to the view, using the JSTL `<c:out>` tag.

**springapp/war/WEB-INF/jsp/hello.jsp**

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<html>
<head><title>Hello :: Spring Application</title></head>
<body>
<h1>Hello - Spring Application</h1>
<p>Greetings, it is now <c:out value="${now}"/>
</p>
</body>
</html>
```

For SpringappController.java there are a few changes we need to make. Change the view to WEB-INF/jsp/hello.jsp since we moved the file to this new location. Also add a string containing the current data and time as the model.

**springapp/src/SpringappController.java**

```
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 java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class SpringappController implements Controller {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        String now = (new java.util.Date()).toString();
        logger.info("returning hello view with " + now);

        return new ModelAndView("WEB-INF/jsp/hello.jsp", "now", now);
    }
}
```

Now we are ready to try this after we build and deploy this new code. We enter <http://localhost:8080/springapp> in a browser and that should pull up index.jsp, which should redirect to hello.htm, which in turn gets us to the controller that sends the data and time to the view.



### Step 15 – Decouple the view and the controller

Right now the controller specifies the full path of the view, which creates an unnecessary dependency between the controller and the view. Ideally we would like to map to the view using a logical name, allowing us to switch the view without having to change the controller. You can set this mapping in a properties file if you like using a `ResourceBundleViewResolver` and a `SimpleUrlHandlerMapping` class. If your mapping needs are simple it is easier to just set a prefix and a suffix on the `InternalResourceViewResolver`. The latter approach is the one that I will implement now, so I modify the `springapp-servlet.xml` and include this `viewResolver` entry. I have elected to use a `JstlView` which will enable us to use JSTL in combination with message resource bundles and it will also support internationalization.

```
springapp/war/WEB-INF/springapp-servlet.xml
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!--
- Application context definition for "springapp" DispatcherServlet.
-->
<beans>
    <bean id="springappController" class="SpringappController"/>
    <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/hello.htm">springappController</prop>
            </props>
        </property>
    </bean>
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property
name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
        <property name="prefix"><value>/WEB-INF/jsp</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>
</beans>

```

So now I can remove the prefix and suffix from the view name in the controller.

#### springapp/src/SpringappController.java

```

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 java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class SpringappController implements Controller {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response)

```

```
        throws ServletException, IOException {
    String now = (new java.util.Date()).toString();
    logger.info("returning hello view with " + now);
    return new ModelAndView("hello", "now", now);
}
}
```

Compile and deploy and the application should still work.

## Step 16 – Add some classes for business logic

So far our application is not very useful. I would like to add a little bit of business logic in form of a Product class and a class that will manage all the products. I name this management class ProductManager. In order to separate the web dependent logic from the business logic I will create two separate packages for the Java source – web and bus. If this was an application for a real company I would name the packages something like com.mycompany.web and com.mycompany.bus, but since this is just a demo application I will keep the package names real short. The Product class is implemented as a JavaBean – it has the default constructor (automatically provided if we don't specify any constructors) and getters and setters for the two instance variables description and price. I also make it Serializable, not necessary for our application, but could come in handy later on if we have to pass this class between different application layers.

**springapp/src/bus/Product.java**

```
package bus;
import java.io.Serializable;
public class Product implements Serializable {
    private String description;
    private Double price;
    public void setDescription(String s) {
        description = s;
    }
    public String getDescription() {
        return description;
    }
    public void setPrice(Double d) {
        price = d;
    }
    public Double getPrice() {
        return price;
    }
}
```

The ProductManager holds a List of Products, and again this this class is implemented as a JavaBean.

**springapp/src/bus/ProductManager.java**

```
package bus;
import java.io.Serializable;
import java.util.List;
public class ProductManager implements Serializable {
    private List products;
    public void setProducts(List p) {
        products = p;
    }
    public List getProducts() {
        return products;
    }
}
```

Next, I modify the SpringappController to hold a reference to this ProductManager class. As you can see, it is now in a separate package called web – remember to move the source to this new location. I also add code to have the controller pass some product information to the view. The getModelAndView now returns a Map with both the date and time and the product manager reference.

**springapp/src/web/SpringappController.java**

```
package 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 java.io.IOException;
import java.util.Map;
import java.util.HashMap;

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

import bus.Product;
import bus.ProductManager;

public class SpringappController implements Controller {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
```

```
private ProductManager prodMan;

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

    String now = (new java.util.Date()).toString();
    logger.info("returning hello view with " + now);

    Map myModel = new HashMap();
    myModel.put("now", now);
    myModel.put("products", getProductManager().getProducts());

    return new ModelAndView("hello", "model", myModel);
}

public void setProductManager(ProductManager pm) {
    prodMan = pm;
}

public ProductManager getProductManager() {
    return prodMan;
}
}
```

### Step 17 – Modify the view to display business data and add support for message bundle

Using the JSTL `<c:forEach>` tag, I add a section that displays product information. I have also replaced the title, heading and greeting text with a JSTL `<fmt:message>` tag that pulls the text to display from a provided 'message' source – I will show this source in a later step.

springapp/war/WEB-INF/jsp/hello.jsp

```

<%@ include file="/WEB-INF/jsp/include.jsp" %>
<html>
<head><title><fmt:message key="title"/></title></head>
<body>
<h1><fmt:message key="heading"/></h1>
<p><fmt:message key="greeting"/> <c:out value="${model.now}"/>
</p>
<h3>Products</h3>
<c:forEach items="${model.products}" var="prod">
  <c:out value="${prod.description}"/> <i>${c:out value="${prod.price}"/></i><br><br>
</c:forEach>
</body>
</html>

```

## Step 18 – Add some test data to automatically populate some business objects

I am not going to add any code to load the business objects from a database just yet. Instead, we can “wire up” a couple of instances using Spring's bean and application context support. I will simply put the data I need as a couple of bean entries in springapp-servlet.xml. I will also add the messageSource entry that will pull in the messages resource bundle ('messages.properties') that I will create in the next step.

### springapp/war/WEB-INF/springapp-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!--
  - Application context definition for "springapp" DispatcherServlet.
  -->
<beans>
  <bean id="springappController" class="web.SpringappController">
    <property name="productManager">
      <ref bean="prodMan"/>
    </property>
  </bean>

  <bean id="prodMan" class="bus.ProductManager">
    <property name="products">
      <list>
        <ref bean="product1"/>
        <ref bean="product2"/>
      </list>
    </property>
  </bean>

```



```
        <ref bean="product3"/>
    </list>
</property>
</bean>

<bean id="product1" class="bus.Product">
    <property name="description"><value>Lamp</value></property>
    <property name="price"><value>5.75</value></property>
</bean>

<bean id="product2" class="bus.Product">
    <property name="description"><value>Table</value></property>
    <property name="price"><value>75.25</value></property>
</bean>

<bean id="product3" class="bus.Product">
    <property name="description"><value>Chair</value></property>
    <property name="price"><value>22.79</value></property>
</bean>

<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>messages</value></property>
</bean>

<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/hello.htm">springappController</prop>
        </props>
    </property>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property
name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
</beans>
```

## Step 19 – Add the message bundle and a 'clean' target to build.xml

I create a 'messages.properties' file in the war/WEB-INF/classes directory. This properties bundle so far has three entries matching the keys specified in the <fmt:message> tags that we added to the hello.jsp.

**springapp/war/WEB-INF/classes/messages.properties**

```
title=SpringApp
heading=Hello :: SpringApp
greeting=Greetings, it is now
```

Since we moved some source files around, it makes sense to add a 'clean' and an 'undeploy' target to the build scripts. I add the following entries to the build.xml file.

```
<target name="clean" description="Clean output directories">
  <delete>
    <fileset dir="${build.dir}">
      <include name="**/*.class"/>
    </fileset>
  </delete>
</target>

<target name="undeploy" description="Un-Deploy application">
  <delete>
    <fileset dir="${deploy.path}/${name}">
      <include name="**/*.*/>
    </fileset>
  </delete>
</target>
```

Now stop the Tomcat server, run the clean, undeploy and deploy targets. This should remove all old class files, re-build the application and deploy it. Start up Tomcat again and you should see the following:



[Back](#) [Part 3 – Adding Unit Tests and a Form to the Application](#)

Copyright © 2003-2005, Thomas Risberg

## Developing a Spring Framework MVC application step-by-step

### Part 3 – Adding Unit Tests and a Form to the Application

Thomas Risberg

July, 2003

(Revised April, 2005)

This is Part 3 of a step-by-step account of how to develop a web application from scratch using the Spring Framework. In Part 1 (Steps 1 – 19) we configured the environment and set up a basic application that we will build upon. Part 2 (Steps 13-19) improved the application in several ways. We are now going to add some unit tests to the application.

#### Step 20 – Add unit test for the SpringappController

Before we create any unit tests, we want to prepare Ant and our build script to be able to handle this. Ant has a built in JUnit target, but we need to add junit.jar to Ant's lib directory. I used the one that came with the Spring distribution spring-framework-1.2/lib/junit/junit.jar. Just copy this file to the lib directory in your Ant installation. I also added the following target to our build script.

```
<target name="junit" depends="build" description="Run JUnit 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="**/Test*.*/>
      </fileset>
    </batchtest>

  </junit>

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

```

    </fail>
</target>

```

Now I add a new sub-directory in the src directory that I name tests. This directory will, as you might have guessed, contain all the unit tests.

After all this, we are ready to start writing the first unit test. The SpringappController depends on both the HttpServletRequest, HttpServletResponse and our application context. Since the controller does not use the request or the response, we can simply pass in null for these objects. If that was not the case, we could create some mock objects using EasyMock that we would pass in during our test. The application context can be loaded outside of a web server environment using a class that will load an application context. There are several available, and for the current task the FileSystemXmlApplicationContext works fine.

#### springapp/src/tests/TestSpringappController.java

```

package tests;
import java.util.Map;
import java.util.List;
import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import junit.framework.TestCase;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import org.springframework.web.servlet.ModelAndView;
import web.SpringappController;
import bus.ProductManager;
import bus.Product;
public class TestSpringappController extends TestCase {
    private ApplicationContext ac;
    public void setUp() throws IOException {
        ac = new FileSystemXmlApplicationContext("src/tests/WEB-INF/springapp-
servlet.xml");
    }
    public void testHandleRequest() throws ServletException, IOException {
        SpringappController sc = (SpringappController)
ac.getBean("springappController");
        ModelAndView mav = sc.handleRequest((HttpServletRequest) null,
(HttpServletResponse) null);
        Map m = mav.getModel();
        List pl = (List) ((Map) m.get("model")).get("products");
        Product p1 = (Product) pl.get(0);
        assertEquals("Lamp", p1.getDescription());
        Product p2 = (Product) pl.get(1);
        assertEquals("Table", p2.getDescription());
        Product p3 = (Product) pl.get(2);
        assertEquals("Chair", p3.getDescription());
    }
}

```

```

    }
}

```

The only test is a call to `handleRequest`, and we check the products that are returned in the model. In the `setUp` method, we load the application context that I have copied into a `WEB-INF` directory in the `src/tests` directory. I create a copy just so this file will work during tests with a small set of beans necessary for running the tests. So, copy `springapp/war/WEB-INF/springapp-servlet.xml` to `springapp/src/tests/WEB-INF` directory. You can then remove the “`messageSource`”, “`urlMapping`” and “`viewResolver`” bean entries since they are not needed for this test.

#### springapp/src/tests/WEB-INF/springapp-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!--
  - Application context definition for "springapp" DispatcherServlet.
-->
<beans>
  <bean id="springappController" class="web.SpringappController">
    <property name="productManager">
      <ref bean="prodMan"/>
    </property>
  </bean>

  <bean id="prodMan" class="bus.ProductManager">
    <property name="products">
      <list>
        <ref bean="product1"/>
        <ref bean="product2"/>
        <ref bean="product3"/>
      </list>
    </property>
  </bean>

  <bean id="product1" class="bus.Product">
    <property name="description"><value>Lamp</value></property>
    <property name="price"><value>5.75</value></property>
  </bean>

  <bean id="product2" class="bus.Product">
    <property name="description"><value>Table</value></property>
    <property name="price"><value>75.25</value></property>
  </bean>

  <bean id="product3" class="bus.Product">
    <property name="description"><value>Chair</value></property>
    <property name="price"><value>22.79</value></property>
  </bean>

```

```

    </bean>

</beans>

```

When you run this test, you should see a lot of log messages from the loading of the application context.

## Step 21 – Add unit test and new functionality for ProductManager

Next I add a test case for the ProductManager, and I also add a test for a new method to increase the prices that I am planning on adding to the ProductManager.

**springapp/src/tests/TestProductManager.java**

```

package tests;
import java.util.List;
import java.util.ArrayList;
import junit.framework.TestCase;
import bus.ProductManager;
import bus.Product;
public class TestProductManager extends TestCase {
    private ProductManager pm;
    public void setUp() {
        pm = new ProductManager();
        Product p = new Product();
        p.setDescription("Chair");
        p.setPrice(new Double("20.50"));
        ArrayList al = new ArrayList();
        al.add(p);
        p = new Product();
        p.setDescription("Table");
        p.setPrice(new Double("150.10"));
        al.add(p);
        pm.setProducts(al);
    }
    public void testGetProducts() {
        List l = pm.getProducts();
        Product p1 = (Product) l.get(0);
        assertEquals("Chair", p1.getDescription());
        Product p2 = (Product) l.get(1);
        assertEquals("Table", p2.getDescription());
    }
    public void testIncreasePrice() {
        pm.increasePrice(10);
        List l = pm.getProducts();

```

```

        Product p = (Product) l.get(0);
        assertEquals(new Double("22.55"), p.getPrice());
        p = (Product) l.get(1);
        assertEquals(new Double("165.11"), p.getPrice());
    }
}

```

For this test, there is no need to create an application context. I just create a couple of products in the setUp method and add them to the product manager. I add tests for both getProducts and increasePrice. The increasePrice method is a cross the board increase based on the percentage passed in to the method. I modify the ProductManager class to implement this new method.

**springapp/src/bus/ProductManager.java**

```

package bus;
import java.io.Serializable;
import java.util.ListIterator;
import java.util.List;

public class ProductManager implements Serializable {

    private List products;

    public void setProducts(List p) {
        products = p;
    }

    public List getProducts() {
        return products;
    }

    public void increasePrice(int pct) {
        ListIterator li = products.listIterator();
        while (li.hasNext()) {
            Product p = (Product) li.next();
            double newPrice = p.getPrice().doubleValue() * (100 + pct)/100;
            p.setPrice(new Double(newPrice));
        }
    }
}

```

Next I build and run the tests. As you can see, this test is just like any regular test – the business classes don't depend on any of the servlet classes so these classes are very easy to test.



## Step 22 – Adding a form

To provide an interface in the web application, I add a form that will allow the user to enter a percentage value. This form uses a tag library named “spring” that is provided with the Spring Framework. We have to copy this file from the Spring distribution `spring-framework-1.2/dist/spring.tld` to the `springapp/war/WEB-INF` directory. Now we must also add a `<taglib>` entry to `web.xml`.

### springapp/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
  <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>

  <taglib>
    <taglib-uri>/spring</taglib-uri>
    <taglib-location>/WEB-INF/spring.tld</taglib-location>
  </taglib>

</web-app>
```

We also have to declare this taglib in a page directive in the jsp file. We declare a form the normal way with a `<form>` tag and an `<input>` text field and a submit button.

### springapp/war/WEB-INF/jsp/priceincrease.jsp

```

<%@ include file="/WEB-INF/jsp/include.jsp" %>
<%@ taglib prefix="spring" uri="/spring" %>
<html>
<head><title><fmt:message key="title"/></title></head>
<body>
<h1><fmt:message key="priceincrease.heading"/></h1>
<form method="post">
  <table width="95%" bgcolor="f8f8ff" border="0" cellspacing="0" cellpadding="5">
    <tr>
      <td alignment="right" width="20%">Increase (%):</td>
      <spring:bind path="priceIncrease.percentage">
        <td width="20%">
          <input type="text" name="percentage" value="<c:out
value="\${status.value}"/>" />
        </td>
        <td width="60%">
          <font color="red"><c:out value="\${status.errorMessage}"/></font>
        </td>
      </spring:bind>
    </tr>
  </table>
  <br>
  <spring:hasBindErrors name="priceIncrease">
    <b>Please fix all errors!</b>
  </spring:hasBindErrors>
  <br><br>
  <input type="submit" alignment="center" value="Execute">
</form>
<a href="<c:url value="hello.htm"/>">Home</a>
</body>
</html>

```

The `<spring:bind>` tag is used to bind an `<input>` form element to a command object `PriceIncrease.java`, that is used together with the form. This command object is later passed in to the validator and if it passes validation it is passed on to the controller. The `\${status.errorMessage}` and `\${status.value}` are special variables declared by the framework that can be used to display error messages and the current value of the field.

**springapp/src/bus/PriceIncrease.java**

```

package bus;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class PriceIncrease {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    private int percentage;
    public void setPercentage(int i) {
        percentage = i;
        logger.info("Percentage set to " + i);
    }
    public int getPercentage() {
        return percentage;
    }
}

```

This is a very simple JavaBean class, and in our case there is a single property with a getter and setter. The validator class gets control after the user presses submit. The values entered in the form will be set on the command object by the framework. The method validate is called and the command object and an object to hold any errors are passed in.

#### springapp/src/bus/PriceIncreaseValidator.java

```

package bus;
import java.io.Serializable;
import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class PriceIncreaseValidator implements Validator {
    private int DEFAULT_MIN_PERCENTAGE = 0;
    private int DEFAULT_MAX_PERCENTAGE = 50;
    private int minPercentage = DEFAULT_MIN_PERCENTAGE;
    private int maxPercentage = DEFAULT_MAX_PERCENTAGE;
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    public boolean supports(Class clazz) {
        return clazz.equals(PriceIncrease.class);
    }
    public void validate(Object obj, Errors errors) {
        PriceIncrease pi = (PriceIncrease) obj;
        if (pi == null) {
            errors.rejectValue("percentage", "error.not-specified", null, "Value
required.");
        }
        else {
            logger.info("Validating with " + pi + ": " + pi.getPercentage());

```

```

        if (pi.getPercentage() > maxPercentage) {
            errors.rejectValue("percentage", "error.too-high",
                new Object[] {new Integer(maxPercentage)}, "Value too high.");
        }
        if (pi.getPercentage() <= minPercentage) {
            errors.rejectValue("percentage", "error.too-low",
                new Object[] {new Integer(minPercentage)}, "Value too low.");
        }
    }
}

public void setMinPercentage(int i) {
    minPercentage = i;
}

public int getMinPercentage() {
    return minPercentage;
}

public void setMaxPercentage(int i) {
    maxPercentage = i;
}

public int getMaxPercentage() {
    return maxPercentage;
}
}

```

Now we need to add an entry in the springapp-servlet.xml file to define the new form and controller. We define properties for command object and validator. We also specify two views, one that is used for the form and one that we will go to after successful form processing. The latter which is called the success view can be of two types. It can be a regular view reference that is forwarded to one of our JSP pages. One disadvantage with this approach is, that if the user refreshes the page, the form data is submitted again, and you would end up with a double priceincrease. An alternative way is to use a redirect, where a response is sent back to the users browser instructing it to redirect to a new url. The url we use in this case can't be one of our JSP pages, since they are hidden from direct access. It has to be a url that is externally reachable. I have choosen to use 'hello.htm' as my redirect url. This url maps to the 'hello.jsp' page, so this should work nicely.

#### springapp/war/WEB-INF/springapp-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--
    - Application context definition for "springapp" DispatcherServlet.
-->
<beans>
    <!-- Controller for the initial "Hello" page -->
    <bean id="springappController" class="web.SpringappController">
        <property name="productManager">
            <ref bean="prodMan"/>
        </property>
    </bean>

```

```

</bean>

<!-- Validator and Form Controller for the "Price Increase" page -->
<bean id="priceIncreaseValidator" class="bus.PriceIncreaseValidator"/>
<bean id="priceIncreaseForm" class="web.PriceIncreaseFormController">
    <property name="sessionForm"><value>true</value></property>
    <property name="commandName"><value>priceIncrease</value></property>
    <property name="commandClass"><value>bus.PriceIncrease</value></property>
    <property name="validator"><ref bean="priceIncreaseValidator"/></property>
    <property name="formView"><value>priceincrease</value></property>
    <property name="successView"><value>hello.htm</value></property>
    <property name="productManager">
        <ref bean="prodMan"/>
    </property>
</bean>

<bean id="prodMan" class="bus.ProductManager">
    <property name="products">
        <list>
            <ref bean="product1"/>
            <ref bean="product2"/>
            <ref bean="product3"/>
        </list>
    </property>
</bean>

<bean id="product1" class="bus.Product">
    <property name="description"><value>Lamp</value></property>
    <property name="price"><value>5.75</value></property>
</bean>

<bean id="product2" class="bus.Product">
    <property name="description"><value>Table</value></property>
    <property name="price"><value>75.25</value></property>
</bean>

<bean id="product3" class="bus.Product">
    <property name="description"><value>Chair</value></property>
    <property name="price"><value>22.79</value></property>
</bean>

<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>messages</value></property>
</bean>

<bean id="urlMapping"

```

```

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/hello.htm">springappController</prop>
            <prop key="/priceincrease.htm">priceIncreaseForm</prop>
        </props>
    </property>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
</beans>

```

Next, let's take a look at the controller for this form. The onSubmit method gets control and does some logging before it calls the increasePrice method on the ProductManager object. It then returns a ModelAndView passing in a new instance of a RedirectView created using the url for the successView.

#### springapp/src/web/PriceIncreaseFormController.java

```

package web;
import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.view.RedirectView;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;
import java.util.HashMap;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import bus.Product;
import bus.ProductManager;
import bus.PriceIncrease;
public class PriceIncreaseFormController extends SimpleFormController {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    private ProductManager prodMan;
    public ModelAndView onSubmit(Object command)
        throws ServletException {

```

```

        int increase = ((PriceIncrease) command).getPercentage();
        logger.info("Increasing prices by " + increase + "%.");
        prodMan.increasePrice(increase);
        String now = (new java.util.Date()).toString();
        logger.info("returning from PriceIncreaseForm view to " + getSuccessView());
        return new ModelAndView(new RedirectView(getSuccessView()));
    }
    protected Object formBackingObject(HttpServletRequest request) throws
ServletException {
        PriceIncrease priceIncrease = new PriceIncrease();
        priceIncrease.setPercentage(20);
        return priceIncrease;
    }
    public void setProductManager(ProductManager pm) {
        prodMan = pm;
    }
    public ProductManager getProductManager() {
        return prodMan;
    }
}

```

We are also adding some messages to the messages.properties resource file.

**springapp/war/WEB-INF/classes/messages.properties**

```

title=SpringApp
heading=Hello :: SpringApp
greeting=Greetings, it is now
priceincrease.heading=Price Increase :: SpringApp
error.not-specified=Percentage not specified!!!
error.too-low=You have to specify a percentage higher than {0}!
error.too-high=Don't be greedy - you can't raise prices by more than {0}%!
required=Entry required.
typeMismatch=Invalid data.
typeMismatch.percentage=That is not a number!!!

```

Finally, we have to provide a link to the priceincrease page from the hello.jsp.

**springapp/war/WEB-INF/jsp/hello.jsp**

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<html>
<head><title><fmt:message key="title"/></title></head>
<body>
<h1><fmt:message key="heading"/></h1>
<p><fmt:message key="greeting"/> <c:out value="${model.now}"/>
</p>
<h3>Products</h3>
<c:forEach items="${model.products}" var="prod">
  <c:out value="${prod.description}"/> <i>${<c:out value="${prod.price}"/></i><br><br>
</c:forEach>
<br>
<a href="<c:url value="priceincrease.htm"/>">Increase Prices</a>
<br>
</body>
</html>
```

Compile and deploy all this and after reloading the application we can test it. This is what the form looks like with errors displayed.





[Back](#) [Part 4 – Implementing Database Persistence](#)

Copyright © 2003-2005, Thomas Risberg

# Developing a Spring Framework MVC application step-by-step

## Part 4 – Implementing Database Persistence

**Thomas Risberg**

August, 2003

(Revised April, 2005)

This is Part 4 of a step-by-step account of how to develop a web application from scratch using the Spring Framework. In Part 1 (Steps 1 – 12) we configured the environment and set up a basic application that we will build upon. Part 2 (Steps 13-19) improved the application in several ways. Part 3 (Steps 20-22) Added some unit tests to the application and we also added a Form for performing a price increase. In Part 4 it is time to deal with database persistence. We saw in the earlier parts how we loaded some business objects using bean definitions in a configuration file. It is obvious that this would never work in real life – whenever we re-start the server we are back to the original prices. We need to add code to actually persist these changes to a database.

### Step 23 – Add Ant tasks to create and load test data

Before we can start developing the persistence code, we should create the database tables that we need for our development and testing. We also need a database. I am planning on using HSQL, which is a good open source database written in Java. This database is distributed with Spring, so we can just copy the jar file to the web apps lib directory. Copy `spring-framework-1.2/lib/hsqldb/hsqldb.jar` to `springapp/war/WEB-INF/lib/`. I am planning on using HSQL in a standalone mode to begin with. That way we don't have to worry about starting up a separate database server every time we use it. The URL we specify for connecting to HSQL controls the mode that the database is running in. To be able to run some Ant tasks for the database we have to add some database properties to the build properties file.

#### springapp/build.properties

```
# Ant properties for building the springapp
appserver.home=${user.home}/jakarta-tomcat-5.0.28
deploy.path=${appserver.home}/webapps
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=tomcat
db.driver=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:db/test
db.user=sa
db.pw=
```

Next I add the targets I need to the build script. There are targets to create and delete tables and to load and delete test data.

```
<target name="createTables">
    <echo message="CREATE TABLES USING: ${db.driver} ${db.url}"/>
    <sql driver="${db.driver}"
        url="${db.url}"
        userid="${db.user}"
        password="${db.pw}"
        onerror="continue">
        <classpath refid="master-classpath"/>

        CREATE TABLE products (
            id INTEGER NOT NULL PRIMARY KEY,
            description varchar(255),
            price decimal(15,2)
        );
        CREATE INDEX products_description ON products(description);

    </sql>
</target>

<target name="dropTables">
    <echo message="DROP TABLES USING: ${db.driver} ${db.url}"/>
    <sql driver="${db.driver}"
        url="${db.url}"
        userid="${db.user}"
        password="${db.pw}"
        onerror="continue">
        <classpath refid="master-classpath"/>

        DROP TABLE products;

    </sql>
</target>

<target name="loadData">
    <echo message="LOAD DATA USING: ${db.driver} ${db.url}"/>
    <sql driver="${db.driver}"
        url="${db.url}"
        userid="${db.user}"
        password="${db.pw}"
        onerror="continue">
        <classpath refid="master-classpath"/>

        INSERT INTO products (id, description, price) values(1, 'Lamp', 5.78);
        INSERT INTO products (id, description, price) values(2, 'Table', 75.29);
        INSERT INTO products (id, description, price) values(3, 'Chair', 22.81);
```

```
        COMMIT;
        SHUTDOWN
    </sql>
</target>

<target name="printData">
    <echo message="PRINT DATA USING: ${db.driver} ${db.url}"/>
    <sql driver="${db.driver}"
        url="${db.url}"
        userid="${db.user}"
        password="${db.pw}"
        onerror="continue"
        print="true">
        <classpath refid="master-classpath"/>

        SELECT * FROM products;

    </sql>
</target>

<target name="clearData">
    <echo message="CLEAR DATA USING: ${db.driver} ${db.url}"/>
    <sql driver="${db.driver}"
        url="${db.url}"
        userid="${db.user}"
        password="${db.pw}"
        onerror="continue">
        <classpath refid="master-classpath"/>

        DELETE FROM products;

    </sql>
</target>
```

Now I execute some of these tasks to set up the test database. This will create a db folder under the springapp directory. Run 'ant createTables loadData printData' – I have included my output below.

```
[trisberg@localhost springapp]$ ant createTables loadData printData
Buildfile: build.xml

createTables:
    [echo] CREATE TABLES USING: org.hsqldb.jdbcDriver jdbc:hsqldb:db/test
    [sql] Executing commands
    [sql] 2 of 2 SQL statements executed successfully

loadData:
    [echo] LOAD DATA USING: org.hsqldb.jdbcDriver jdbc:hsqldb:db/test
    [sql] Executing commands
    [sql] 3 of 3 SQL statements executed successfully

printData:
    [echo] PRINT DATA USING: org.hsqldb.jdbcDriver jdbc:hsqldb:db/test
    [sql] Executing commands
    [sql] ID,DESCRIPTION,PRICE
    [sql] 1,Lamp,5.78
    [sql] 2,Table,75.29
    [sql] 3,Chair,22.81

    [sql] 1 of 1 SQL statements executed successfully

BUILD SUCCESSFUL
Total time: 2 seconds
```

## Step 24 – Create a Data Access Object (DAO) implementation for JDBC.

I begin with creating a new 'springapp/src/db' directory to contain any classes that are used for database access. In this directory I create a new interface called 'ProductManagerDao.java'. This will be the interface that defines the functionality that the DAO implementation classes will provide – we could choose to have more than one implementation.

**springapp/src/db/ProductManagerDao.java**

```
package db;

import bus.Product;
import java.util.List;

public interface ProductManagerDao {

    public List getProductList();

    public void increasePrice(Product prod, int pct);

}
```

I'll follow this with a class called 'ProductManagerDaoJdbc.java' that will be the JDBC implementation of this interface. Spring provides a JDBC abstraction framework that we will make use of. The biggest difference between using straight JDBC and Spring's JDBC framework is that you don't have to worry about opening and closing the connection or any statements. It is all handled for you. Another advantage is that you won't have to catch any exceptions, unless you want to. Spring wraps all `SQLExceptions` in it's own unchecked exception hierarchy inheriting from `DataAccessException`. If you want to you can catch this exception, but since most database exceptions are impossible to recover from anyway, you might as well just let the exception propagate up to a higher level.

#### springapp/src/db/ProductManagerDaoJdbc.java

```
package db;

import bus.Product;
import java.util.List;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import javax.sql.DataSource;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.object.MappingSqlQuery;
import org.springframework.jdbc.object.SqlUpdate;
import org.springframework.jdbc.core.SqlParameter;

public class ProductManagerDaoJdbc implements ProductManagerDao {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    private DataSource ds;

    public List getProductList() {
        logger.info("Getting products!");
        ProductQuery pq = new ProductQuery(ds);
```

```

        return pq.execute();
    }

    public void increasePrice(Product prod, int pct) {
        logger.info("Increasing price by " + pct + "%");
        SqlUpdate su =
            new SqlUpdate(ds, "update products set price = price * (100 + ?) / 100
where id = ?");
        su.declareParameter(new SqlParameter("increase", Types.INTEGER));
        su.declareParameter(new SqlParameter("ID", Types.INTEGER));
        su.compile();
        Object[] oa = new Object[2];
        oa[0] = new Integer(pct);
        oa[1] = new Integer(prod.getId());
        int count = su.update(oa);
        logger.info("Rows affected: " + count);
    }

    public void setDataSource(DataSource ds) {
        this.ds = ds;
    }

    class ProductQuery extends MappingSqlQuery {

        ProductQuery(DataSource ds) {
            super(ds, "SELECT id, description, price from products");
            compile();
        }

        protected Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Product prod = new Product();
            prod.setId(rs.getInt("id"));
            prod.setDescription(rs.getString("description"));
            prod.setPrice(new Double(rs.getDouble("price")));
            return prod;
        }

    }
}

```

Lets go over the two DAO methods in this class. First, getProductList() creates a query object that will retrieve all the products. This query is executed and the results are passed back as a list of Products. At the end of the class we can see the definition for this query class. I have implemented it as an inner class since we are not going to reference it anywhere else, but I could just as well have made it a separate class. This query class extends MappingSqlQuery, which is a central class in Spring's JDBC

framework. The idea behind this class is that you have to specify a SQL query when this class is created, and you are also responsible for implementing the `mapRow` method to map the data from each row into a class that represents the entity you are retrieving in your query. That's it, that's all you have to provide. The rest is managed by the framework. In the constructor of the `ProductQuery` class I pass in the data source. This data source will be provided in a similar fashion to the way we wired up the business objects in Part 2, so we don't have to worry about retrieving a data source in our DAO class. In the constructor I also define the SQL query that we will use to retrieve the products. The `mapRow` method will be called once for each row returned by the query. It creates a new `Product` and populates it based on the data retrieved from the current row of the resultset. You should not call `getNext` on the resultset, it is all handled by the framework. This is another example of an Inversion of Control solution.

The second method `increasePrice` is utilizing an `SqlUpdate` class. This class is passed the data source and an SQL update statement with placeholders for parameters. Same syntax as a prepared statement in JDBC. In fact, that is what `SqlUpdate` creates behind the scenes. For the parameters, we have to give them a name and declare what type they are so that the framework can set them before the prepared statement is executed. After all parameters have been declared, we “compile” the statement in JDO fashion. This will signal that we are done declaring parameters, and the framework will check to make sure that we have a matching declaration for each placeholder in the SQL statement. Next we declare an Object array that will hold the parameter values that we are going to pass in to the prepared statement. This array gets passed into the update method of `SqlUpdate`. The update method does return the count of rows affected.

I need to store the value of the primary key for each product in the `Product` class. This key will be used when I persist any changes to the object back to the database. To hold this key I add a private field named 'id' complete with setters and getters to `Product.java`.

**springapp/src/bus/Product.java**

```
package bus;

import java.io.Serializable;

public class Product implements Serializable {

    private int id;
    private String description;
    private Double price;

    public void setId(int i) {
        id = i;
    }

    public int getId() {
        return id;
    }

    public void setDescription(String s) {
        description = s;
    }

    public String getDescription() {
        return description;
    }
}
```



```

    public void setPrice(Double d) {
        price = d;
    }

    public Double getPrice() {
        return price;
    }
}

```

Now it is time to test this whole DAO setup. Actually, we probably should have written the tests first, but since this is a tutorial style document I think it makes more sense to introduce the actual code before the tests. I decided to test with a live database, so I will have to add dependencies for `clearData` and `loadData` to the `junit` task in the `build.xml`. This will ensure that we will have a consistent starting point for our tests.

```

<target name="junit" depends="build,clearData,loadData" description="Run JUnit
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="**/Test*.*/>
            </fileset>
        </batchtest>

    </junit>

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

```

Next, I add a `TestProductManagerDaoJdbc.java` to our collection of unit tests. In the setup part I create a data source that will be used for testing.

**springapp/src/test/TestProductManagerDaoJdbc.java**

```
package tests;

import java.util.List;
import java.util.ArrayList;
import junit.framework.TestCase;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import db.ProductManagerDaoJdbc;
import bus.Product;

public class TestProductManagerDaoJdbc extends TestCase {

    private ProductManagerDaoJdbc pmdao;

    public void setUp() {
        pmdao = new ProductManagerDaoJdbc();
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("org.hsqldb.jdbcDriver");
        ds.setUrl("jdbc:hsqldb:db/test");
        ds.setUsername("sa");
        ds.setPassword("");
        pmdao.setDataSource(ds);
    }

    public void testGetProductList() {
        List l = pmdao.getProductList();
        Product p1 = (Product) l.get(0);
        assertEquals("Lamp", p1.getDescription());
        Product p2 = (Product) l.get(1);
        assertEquals("Table", p2.getDescription());
    }

    public void testIncreasePrice() {
        List l1 = pmdao.getProductList();
        Product p1 = (Product) l1.get(0);
        assertEquals(new Double("5.78"), p1.getPrice());
        pmdao.increasePrice(p1, 10);
        List l2 = pmdao.getProductList();
        Product p2 = (Product) l2.get(0);
        assertEquals(new Double("6.36"), p2.getPrice());
    }
}
```

When we pass the unit tests, we can move on and modify the web application to make use of these new database persistence capabilities.

### Step 25 – Modify web application to use database persistence.

If we structured our application properly, we should only have to change the business classes to take advantage of the database persistence. The view and controller classes should not have to be modified, since they should be unaware of any implementation details of the business classes. So let's add the persistence to the ProductManager class. I add a reference to a ProductManagerDao interface plus a setter method for this reference. Which implementation that we actually use should be irrelevant to the ProductManager class, and we will set this through a configuration option.

springapp/src/bus/ProductManager.java

```
package bus;

import java.io.Serializable;
import java.util.ListIterator;
import java.util.List;
import db.ProductManagerDao;

public class ProductManager implements Serializable {

    private ProductManagerDao pmd;
    private List products;

    public void setProductManagerDao(ProductManagerDao pmd) {
        this.pmd = pmd;
    }

    /*
    public void setProducts(List p) {
        products = p;
    }
    */

    public List getProducts() {
        products = pmd.getProductList();
        return products;
    }

    public void increasePrice(int pct) {
        ListIterator li = products.listIterator();
        while (li.hasNext()) {
            Product p = (Product) li.next();
        }
    }

    /*
```

```

        double newPrice = p.getPrice().doubleValue() * (100 + pct)/100;
        p.setPrice(new Double(newPrice));
    }

    pmd.increasePrice(p, pct);
}
}
}

```

We will no longer rely on the product list that we have in memory. Every time `getProducts` is called, we will go out and query the database. The `increasePrice` method will now delegate to the DAO to increase the price and the new price will be reflected the next time we call the `getProducts` method.

Next we need to modify the configuration file for the web application – `springapp-servlet.xml`.

#### springapp/war/WEB-INF/springapp-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--
    - Application context definition for "springapp" DispatcherServlet.
-->

<beans>

    <!-- Controller for the initial "Hello" page -->
    <bean id="springappController" class="web.SpringappController">
        <property name="productManager">
            <ref bean="prodMan"/>
        </property>
    </bean>

    <!-- Validator and Form Controller for the "Price Increase" page -->
    <bean id="priceIncreaseValidator" class="bus.PriceIncreaseValidator"/>
    <bean id="priceIncreaseForm" class="web.PriceIncreaseFormController">
        <property name="sessionForm"><value>true</value></property>
        <property name="commandName"><value>priceIncrease</value></property>
        <property name="commandClass"><value>bus.PriceIncrease</value></property>
        <property name="validator"><ref bean="priceIncreaseValidator"/></property>
        <property name="formView"><value>priceincrease</value></property>
        <property name="successView"><value>hello.htm</value></property>
        <property name="productManager">
            <ref bean="prodMan"/>
        </property>
    </bean>

```

```

        </property>
    </bean>

    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property
name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
        <property name="url">
            <value>jdbc:hsqldb:/Users/trisberg/projects/springapp/db/test</value>
        </property>
        <property name="username"><value>sa</value></property>
        <property name="password"><value></value></property>
    </bean>

    <bean id="prodManDao" class="db.ProductManagerDaoJdbc">
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
    </bean>

    <bean id="prodMan" class="bus.ProductManager">
        <property name="productManagerDao">
            <ref bean="prodManDao" />
        </property>
<!--
        <property name="products">
            <list>
                <ref bean="product1" />
                <ref bean="product2" />
                <ref bean="product3" />
            </list>
        </property>
-->
    </bean>

<!--
    <bean id="product1" class="bus.Product">
        <property name="description"><value>Lamp</value></property>
        <property name="price"><value>5.75</value></property>
    </bean>

    <bean id="product2" class="bus.Product">
        <property name="description"><value>Table</value></property>
        <property name="price"><value>75.25</value></property>
    </bean>

```

```

    <bean id="product3" class="bus.Product">
        <property name="description"><value>Chair</value></property>
        <property name="price"><value>22.79</value></property>
    </bean>
-->

    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename"><value>messages</value></property>
    </bean>

    <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/hello.htm">springappController</prop>
                <prop key="/priceincrease.htm">priceIncreaseForm</prop>
            </props>
        </property>
    </bean>

    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.JstlView</value>
        </property>
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>
</beans>

```

I remove the population of a set of products that were passed in to the ProductManager. I replace this by adding beans for a DataSource and a ProductManagerDaoJdbc implementation. The URL provided for the datasource includes the full path for the database location (/Users/trisberg/projects/springapp/db/test). You will of course have to adjust this to match your setup.

Now we can build and deploy the modified application. Run 'ant undeploy deploy' to clean out any of the old classes and replace them with the new ones. Then fire up Tomcat and pull up the application. The only difference you should see is the decimals limited to two positions due to the column being declared that way in the database. Also, you should see any price increases remaining in effect even after you cycle the application server.

## Step 26 – Fix the broken tests.

We have radically changed the ProductManager implementation – we pushed all the functionality down to the ProductManagerDao implementation, so now the old tests fail. To fix this I will create a mock implementation of the ProductManagerDao. This implementation will basically mimic the functionality we had in the original ProductManager.

**springapp/src/tests/MockProductManagerDaoImpl.java**

```
package tests;

import bus.Product;
import java.util.List;
import db.ProductManagerDao;
import bus.Product;

public class MockProductManagerDaoImpl implements ProductManagerDao {

    private List products;

    public void setProducts(List p) {
        products = p;
    }

    public List getProductList() {
        return products;
    }

    public void increasePrice(Product prod, int pct) {
        double newPrice = prod.getPrice().doubleValue() * (100 + pct)/100;
        prod.setPrice(new Double(newPrice));
    }

}
```

Now we have to make sure to configure the 'TestSpringappController' and 'TestProductManager' tests to use this mock implementation. For the 'TestSpringappController' I will modify the 'springapp-servlet.xml' file that I copied into the tests/WEB-INF directory (I knew it would be a good idea to make a copy of this file). Do not modify the one located in war/WEB-INF – we need that one to stay as is so we can run the application in Tomcat. We move the list of products from the ProductManager to the ProductManagerDao and give ProductManager a reference to this DAO.

**springapp/src/tests/WEB-INF/springapp-servlet.xml**

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<!--
- Application context definition for "springapp" DispatcherServlet.
-->

<beans>
    <bean id="springappController" class="web.SpringappController">
        <property name="productManager">
            <ref bean="prodMan"/>
        </property>
    </bean>

    <bean id="prodManDao" class="tests.MockProductManagerDaoImpl">
        <property name="products">
            <list>
                <ref bean="product1"/>
                <ref bean="product2"/>
                <ref bean="product3"/>
            </list>
        </property>
    </bean>

    <bean id="prodMan" class="bus.ProductManager">
        <property name="productManagerDao">
            <ref bean="prodManDao"/>
        </property>
    </bean>

    <!--
        <property name="products">
            <list>
                <ref bean="product1"/>
                <ref bean="product2"/>
                <ref bean="product3"/>
            </list>
        </property>
    -->

    </bean>

    <bean id="product1" class="bus.Product">
        <property name="description"><value>Lamp</value></property>
        <property name="price"><value>5.75</value></property>
    </bean>

```



```

<bean id="product2" class="bus.Product">
    <property name="description"><value>Table</value></property>
    <property name="price"><value>75.25</value></property>
</bean>

<bean id="product3" class="bus.Product">
    <property name="description"><value>Chair</value></property>
    <property name="price"><value>22.79</value></property>
</bean>

</beans>

```

For the TestProductManager test case we will make similar modifications to the setUp method.

**springapp/src/tests/TestProductManager.java**

```

package tests;
import java.util.List;
import java.util.ArrayList;
import junit.framework.TestCase;
import db.ProductManagerDao;
import bus.ProductManager;
import bus.Product;

public class TestProductManager extends TestCase {

    private ProductManager pm;

    public void setUp() {
        pm = new ProductManager();
        Product p = new Product();
        p.setDescription("Chair");
        p.setPrice(new Double("20.50"));
        ArrayList al = new ArrayList();
        al.add(p);
        p = new Product();
        p.setDescription("Table");
        p.setPrice(new Double("150.10"));
        al.add(p);

        /*
        pm.setProducts(al);
        */

        MockProductManagerDaoImpl pmdao = new MockProductManagerDaoImpl();
    }
}

```

```
        pmdao.setProducts(al);
        pm.setProductManagerDao(pmdao);
        pm.getProducts();
    }

    public void testGetProducts() {
        List l = pm.getProducts();
        Product p1 = (Product) l.get(0);
        assertEquals("Chair", p1.getDescription());
        Product p2 = (Product) l.get(1);
        assertEquals("Table", p2.getDescription());
    }

    public void testIncreasePrice() {
        pm.increasePrice(10);
        List l = pm.getProducts();
        Product p = (Product) l.get(0);
        assertEquals(new Double("22.55"), p.getPrice());
        p = (Product) l.get(1);
        assertEquals(new Double("165.11"), p.getPrice());
    }
}
```

Compile everything and run the tests. They should run successfully now.

There are still a few things I would like to improve upon. To begin with, we are not using a connection pool which is a must for any serious web application deployment. We are also using HSQL in a standalone mode, which is limiting us to one connection at a time. This of course will not scale. Another issue that we are not dealing with is concurrency. Two users could increase prices simultaneously and they would be a bit surprised by seeing the combined effect of their actions. I'll leave these improvements up to you for now – I might add them in an additional part to this document in the future.

[Back](#)

Copyright © 2003-2005, Thomas Risberg