

## LESSON I - Introduction - Your First Struts App

The application you are going to create mimics entering an employee into a database. The user will be required to enter an employee's name and age.

Concepts introduced in Lesson I:

- Setting up your environment
- Data Transfer Object
- ActionForm
- web.xml
- struts-config.xml
- ApplicationResources.properties
- BeanUtils
- Tag usage

The steps (see left menu) in this lesson will walk you through building all of the necessary components for this small application. If you want to download the complete application you can do so. (You should be able to plop this war file into your application server webapps directory and it should work fine).

[Download rr\\_lesson\\_1 application.war](#)

Begin lesson now by clicking [START](#).

## LESSON I - 1 - Install Tomcat

Download and install the latest stable version of Tomcat:

You can download Tomcat here: <http://jakarta.apache.org/site/binindex.cgi>

Setting up Tomcat is not difficult but is out of the scope of this tutorial. (Most of these basic tutorials should run fine on Tomcat versions 4.0.x and above or any other decent application server).

## LESSON I - 2 - Create Application Directory

Create the web application directory:

Create the directory "rr\_lesson\_1" in the {tomcat}/webapps/ directory (where {tomcat} equals the root directory of your Tomcat installation).

The following directory structure should look like:

webapps

```
|
rr_lesson_1
|
--- WEB-INF
|
|--- classes
| |
|   --- net
|     |
|     -- reumann
|--- lib
|
--- src
|
--- net
|
-- reumann
```

## LESSON I - 3 - Add Struts Files

Download & Install Struts:

Download the latest version of Struts here: <http://www.apache.org/dist/jakarta/struts/binaries/>  
(binary version)

The lesson assumes you are using Struts 1.1. The directory structure at the time you are reading this may be different than the above binary you download. (If you can not seem to find the files you need, you can just use the files that come with the download of this lesson application in the war file).

After downloading your Struts archive, extract it into a directory of your choosing (preferably outside of the entire Tomcat directory).

Copy .tld files from struts into rr\_lesson\_1 application:

Go to the {StrutsDirectory}/contrib/struts-el/lib directory and copy the following .tld files into the rr\_lesson\_1/WEB-INF directory:

```
c.tld
struts-bean-el.tld
struts-html-el.tld
struts-logic-el.tld
```

Copy .jar files from struts into rr\_lesson\_1 application:

Next copy the following .jar files from {StrutsDirectory}/contrib/struts-el/lib into rr\_lesson\_1/WEB-INF/lib directory:

```
commons-beanutils.jar
commons-collections.jar
commons-digester.jar
commons-logging.jar
jstl.jar
standard.jar
struts-el.jar
struts.jar
```

(Note we are using the tld files and jars in the contributed struts-el directory since this will help us to use the standard JSTL tags whenever possible).

## LESSON I - 4 - Create Data Transfer Object

Since we are dealing with an Employee that we want to insert, we need a way to store information about this Employee that we could hand off to a business object (the model layer of MVC). Our model layer will be responsible for doing the actual insert. So the first thing we need is a class representing our employee. We'll make a bean that has just a couple of fields and appropriate get and set methods.

Since this object will transfer stored information from one part of our application to another it is called a Data Transfer Object (or often Value Object).

Create EmployeeDTO:

```
package net.reumann;
```

```
public class EmployeeDTO {
    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
```

```
        return age;
    }
}
```

## LESSON I - 5 - Create Business Service

Probably the most complex part of your web application will be dealing with the business logic. Since this area is beyond solutions that Struts provides, we aren't going to get into that area in this lesson. However, we need to create at least one object that will act as a transition object to the model layer of our MVC application.

We will create an `EmployeeService` class to handle the small amount of business logic we have. In a more complex set up you might have a factory that returns an `EmployeeService` implementation of a `Service` interface.

Create `EmployeeService`:

```
package net.reumann;

public class EmployeeService {
    public EmployeeDTO insertEmployee( EmployeeDTO employee ) {
        //in real life call other business classes to do insert
        //ie:
        //EmployeeDAO.insertEmployee( employee );
        return employee;
    }
}
```

Note: we do not have to return the `EmployeeDTO` back. You might want to return a boolean or some other variable instead that indicates success. Failure should throw an `Exception` and our `insertEmployee` method in real life would probably handle some sort of `DAOException` if it were thrown.

## LESSON I - 6 - Create ActionForm

We have an `EmployeeDTO` object to hold the employee information that will get passed to our `EmployeeService` object which is responsible for calling some other business model components to actually do the insert into the database. Remember, though, our `DTO` is a bean that holds the correct data types of the information that we care about inserting (for example type `int` for age). Information that the user inputs on JSP form will always be submitted as `String` parameters in the request. We may also want to validate the information that is submitted in the form. Struts uses `ActionForm` objects to hold the JSP form data that is submitted. (They could also be used for displaying results back on a page, although this is a less common function). The `ActionForm` should contain `String` fields representing the properties in the `EmployeeDTO`.

Create EmployeeForm:

```
package net.reumann;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class EmployeeForm extends ActionForm {

    private String name;
    private String age;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(String age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public String getAge() {
        return age;
    }
}
```

## LESSON I - 7 - Create web.xml

Normally you would probably create this file first but we are doing it here just to show how it fits in with the struts application as a whole.

We just created our EmployeeForm which will end up holding the fields that the user enters on a JSP form, but before we create the JSP page, there are few things to do first. Since we are using an MVC architecture all requests will pass through a controller. The main controller of Struts is the org.apache.struts.action.ActionServlet. A lot takes place when this controller servlet is called. ( Chuck Cavaness' book 'Programming Jakarta Struts' explains this process well in Chapter 5 ). The basic thing to understand at this point is that all your requests will pass to this servlet and eventually mappings that you set up will be called as a result of requests passed to this controller servlet. This servlet is the main thing that we need to declare in your application's web.xml file.

Create the web.xml file in the rr\_lesson\_1/WEB-INF directory:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!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>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>application</param-name>
    <param-value>ApplicationResources</param-value>
  </init-param>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>3</param-value>
  </init-param>
  <init-param>
    <param-name>detail</param-name>
    <param-value>3</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<!-- Action Servlet Mapping -->
```

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>/do/*</url-pattern>
</servlet-mapping>
```

```
<!-- The Welcome File List -->
```

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

```

<!-- tag libs -->
<taglib>
  <taglib-uri>struts/bean-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts/html-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-html-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts/logic-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>jstl/c</taglib-uri>
  <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>

</web-app>

```

Note: Any URI call to /do/\* will send the request to this ActionServlet. (You don't have to use /do/\*, you can use whatever you want- ie. /\*.action, /\*.do, etc). Also note that all of the tags we will use are defined in this file as well.

## LESSON I - 8 - Create struts-config.xml

If you look at the ActionServlet declaration in the web.xml file you see the config parameter "struts-config.xml." The controller ActionServlet will use this struts-config file to determine all kinds of important things. This struts-config.xml file will be the 'road map' of our application. This file will tell our requests where to go (usually an Action class), what Form beans to use, and where to go after the request submits. Don't worry about understanding all the parts of this file right now. You will understand more as you go along.

### Create struts-config.xml file in rr\_lesson\_1/WEB-INF directory:

```

<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd" >
<struts-config>
  <!-- Form Bean Definitions -->
  <form-beans>
    <form-bean name="employeeForm" type="net.reumann.EmployeeForm"/>

```

```

</form-beans>
<!-- Action Mapping Definitions -->
<action-mappings>
  <action path="/setupEmployeeForm" forward="/employeeForm.jsp"/>

  <action path="/insertEmployee"
    type="net.reumann.InsertEmployeeAction"
    name="employeeForm"
    scope="request"
    validate="false"
    >
    <forward
      name="success"
      path="/confirmation.jsp"/>
    </action>
</action-mappings>
<!-- message resources -->
<message-resources
  parameter="ApplicationResources"
  null="false" />
</struts-config>

```

Some points about the struts-config.xml file:

Notice we defined a form-bean definition for the EmployeeForm (ActionForm) that we created in Step 6. The name property was called "employeeForm." Now look at our the action-mapping for the path "/insertEmployee" in the struts-config file. When a user submits a form (or link) with the path /do/insertEmployee the request will be sent to the ActionServlet which we defined in the web.xml. Eventually a RequestProcessor object called from the ActionServlet will find this mapping and send the request to the type of Action class that we define in this mapping. (You'll learn about the Action class next). This Action class object is defined where you see type="net.reumann.InsertEmployeeAction". Our request will go to this class and when it exists you will see it try to get the 'forward' mapping with the name "success" and will forward to the page defined in our mapping (/confirmation.jsp). The last thing to note is the definition of the message-resources. The ApplicationResources value refers to a properties file (ApplicationResources.properties) that we are going to add to our classes directory. This file will hold key/value pairs that will save us from having to hard code information directly in our JSPs.

## LESSON I - 9 - Create Action class

The main classes used in the Struts framework are those of org.apache.struts.action.Action. The Action classes are the true concrete bridges between the client's request and the business layer. For the most part, each Action class will handle one particular type of business operation the client wants to perform (in this case the insert of an Employee).

Notice our action mapping in the struts-config file:

```
<action path="/insertEmployee"
        type="net.reumann.InsertEmployeeAction"
        name="employeeForm"
        scope="request"
        >
    <forward
        name="success"
        path="/confirmation.jsp"/>
</action>
```

The path attribute will correspond to the action we will define for our JSP form. When the form is submitted with this action path our request will make it to InsertEmployeeAction. When it exits the InsertEmployeeAction it will forward to confirmation.jsp by looking up the forward name "success."

### Create InsertEmployeeAction:

```
package net.reumann;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.commons.beanutils.BeanUtils;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public final class InsertEmployeeAction extends Action {

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        EmployeeService service = new EmployeeService();
        EmployeeForm employeeForm = (EmployeeForm) form;
        EmployeeDTO employeeDTO = new EmployeeDTO();
        BeanUtils.copyProperties( employeeDTO, employeeForm );
        service.insertEmployee( employeeDTO );
        request.setAttribute("employee",employeeDTO);
        return (mapping.findForward("success"));
    }
}
```

```
}
```

The Action class has one method to worry about "execute(..)." When we submit our JSP page the behind the scenes Struts stuff (Action Servlet and RequestProcessor) will find this Action class that we associated with the /insertEmployee action in the struts-config.xml file and the execute method will be called. Since the /insertEmployee action uses the EmployeeForm the InsertEmployeeAction will have this form bean of type ActionForm, so we first cast this into type EmployeeForm. Now we want to get the information in our EmployeeForm into the EmployeeDTO so we can pass that off to our Model layer. We created an instance of EmployeeDTO and then, like magic, we can use BeanUtils (from the Jakarta commons package) to transfer the data from EmployeeForm into the EmployeeDTO with one easy line: BeanUtils.copyProperties( employeeDTO, employeeForm ). This is a HUGE time saver, for if you are not going to use BeanUtils (or PropertyUtils) than you would normally build a helper class to do all the get and set stuff and all the data type conversions (a real pain, especially for large beans).

After copyProperties() the now populated EmployeeDTO is passed off to the service object and the insert would be done. (We'll learn more in a later lesson about dealing with Exceptions that your model/business objects may throw). After the insert is done the EmployeeDTO is stuck into request scope so we could use the employee information for display purposes.

The last thing to notice is that we need to return an ActionForward object which will tell our behind the scenes controller where we should forward to when the execute method is completed. In this case I want to forward to whatever I defined as "success" in my struts-config mapping for this action. Calling mapping.findFoward("success") will return an ActionForward object based on looking up the foward parameter in our mapping that matches "success" (in this case /confirmation.jsp). (Note: Rather than hard-code the word "success" here you should create a Constants class/interface that has these commonly used String variables).

## LESSON I - 10 - Create Resources File

In the struts-config.xml file there is the definition:

```
<message-resources
    parameter="ApplicationResources"
    null="false" />
```

This tells our application that we are using a properties file called "ApplicationResources.properties" which is located in our WEB-INF/classes directory. (If you aren't using Ant, or some other kind of build tool, you might want to put this same file in you src directory as well, since your IDE might delete classes and rebuild from src.) This file will be use mostly to hold display items for our JSPs. It allows us to not have to hard code titles, error messages, button names, etc. We could even have different files based on different languages. This file will normally contain a lot of information, and you will see why in further lessons.

**Create ApplicationResources.properties in rr\_lesson\_1/WEB-INF/classes:**

```
#-- titles --
title.employeeApp=EMPLOYEE APPLICATION
title.employee.employeeform=EMPLOYEE FORM
```

title.employee.insert.confirmation=EMPLOYEE INSERT CONFIRMATION

#-- buttons --

button.submit=SUBMIT

## LESSON I - 11 - Create Style Sheet

Our application is going to use a simple style sheet called rr.css.

### Create rr.css in directory rr\_lesson\_1/ :

```
body { font-family: sans-serif;
       color: black;
       background-color: #F1FFD2;
       font-size: 13pt;
}
#success { color: green; font-weight: bold; }
#error { color: red; font-weight: bold; }
```

(Note: #'s are not comment markers in style sheets).

## LESSON I - 12 - Create index.jsp

### Create index.jsp in rr\_lesson\_1/ :

```
<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employeeApp"/></title>
</head>
<body>
<h1><bean:message key="title.employeeApp"/></h1>
<br>
<html:link page="/do/setupEmployeeForm">Add An Employee</html:link>
</body>
</html>
```

In this application, this is the first page the user will see. Notice the taglibs needed are defined at the top. The first html:rewrite tag will prepend our webapp context (in this case rr\_lesson\_1) to /rr.css. This is a useful tag since it enables us to not have to hard code your webapp name in your code. Next the bean:message tag is used to look up the key 'title' in the ApplicationResources.properties file and write that to the page. Then notice the use of the html:link page. The html:link tag is useful since it will write out an <a href.> tag using the correct path to your root directory added to the beginning of your link. It also appends a sessionId in case the user has cookies disabled. Also notice that the link is calling /do/setupEmployeeForm. This will actually forward us through the controller servlet (since it's mapped to /do/\*). The controller knows where to forward to when the link is clicked based on the mapping in the struts-config file:

```
<action path="/setupEmployeeForm" forward="/employeeForm.jsp"/>
```

Clicking on the link will thus forward us to the employeeForm.jsp page.

(Note: Even better than using `html:link page=""` is using `html:link forward=""`. Using `html:link forward="someForward"` you can set up the actual `/do/whatever` mapping that "someForward" will call in the strutsConfig file ).

## LESSON I - 13 - Create employeeForm.jsp

**Create employeeForm.jsp in rr\_lesson\_1/ :**

```
<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employee.employeeform"/></title>
</head>
<body>
<h1><bean:message key="title.employee.employeeform"/></h1>
<html:form action="insertEmployee" focus="name">
<table>
<tr>
<td >Name:</td>
<td><html:text property="name"/></td>
</tr>
<tr>
<td>Age:</td>
<td><html:text property="age"/></td>
</tr>
</table>
<html:submit><bean:message key="button.submit"/></html:submit>
</html:form>
</body>
</html>
```

Here you see the use of `html:form` tag. The form tag will end up displaying:

```
<form name="employeeForm" method="post" action="/rr_lesson_1/do/insertEmployee">
```

The name of the form- "employeeForm" comes from the name of the formBean that we associated in the mapping for `/insertEmployee` in our `struts-config.xml` file (name="employeeForm"). All you have to worry about is using the correct action when you use the `<html:form action=""/>` tag and the rest is handled by Struts. Using `focus="name"` will

write some javascript to the page to have our form begin with focus on the name field (name in this case refers to the "name" field in the actual EmployeeForm bean). Struts makes extensive use of form field tags like <html:text ..> In our example Struts took care of putting the EmployeeForm we defined in our struts-config.xml mapping into request scope and the html:text tags will look for the fields mentioned as property values in the EmployeeForm and will set them up as the name values for our input buttons. If there are any values associated with the fields in the form it will also display that value. If our EmployeeForm had an initial age value of "33" then <html:text property="age"/> would end up displaying in the source as <input type="text" name="age" value="33">.

## LESSON I - 14 - Create confirmation.jsp

### Create confirmation.jsp in rr\_lesson\_1/ :

```
<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<jsp:useBean id="employee" scope="request" class="net.reumann.EmployeeDTO"/>
<html>
<head>
<title><bean:message key="title.employee.insert.confirmation"/></title>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
</head>
<body>
<h1><bean:message key="title.employee.insert.confirmation"/></h1>
<br>
<span id="success">SUCCESSFUL INSERT OF EMPLOYEE</span>
<br><br>
NAME: <c:out value="{employee.name}"/><br>
AGE: <c:out value="{employee.age}"/>
</body>
</html>
```

This is the page our Action will forward to after leaving the InsertEmployeeAction. Remember in the execute method we put an EmployeeDTO into request scope. Using the JSTL c tag we can display the properties of this object which was placed in request scope.

This wraps up Lesson I. I hope you found it helpful. Please send any comments or suggestions to [struts@reumann.net](mailto:struts@reumann.net)

## LESSON II - Introduction

Just like in Lesson 1, the application you will create mimics entering an employee into a database. The user will be required to enter an employee's name, age, and department.

Concepts introduced in Lesson II:

- Validation (in ActionForm)
- ErrorMessages
- ActionMessages
- Pre-poluation of form
- Dealing with Errors thrown in Actions
- html:select and html:options tag

In case you are starting with this lesson and skipping Lesson I, remember this Lesson does assume you understand the concepts of Lesson I.

If you have already completed Lesson I you can simply make a copy of `rr_lesson_1` in your `webapps` directory and rename it `rr_lesson_2`. As you procede through this lesson simply modify any existing code so that it matches what is presented in this lesson. (Obviously you will have to add any new components as well). Of course you can always just copy and paste from this lesson or, if you prefer, you can download the entire `rr_lesson_2` application:

[Download rr\\_lesson\\_2 application.war](#)

Begin lesson now by clicking [START](#).

## LESSON II - 1 - Setup

These Lessons assume you are using Tomcat 4.1 and have downloaded Struts 1.1-rc2 or greater.

Directory structure under tomcat should look like:

`webapps`

```
|
  rr_lesson_2
    |
    --- WEB-INF
      |
      |--- classes
      |  |
      |  --- net
      |    |
      |    -- reumann
      |--- lib
```

```
|
--- src
  |
  --- net
    |
    -- reumann
```

Copy .tld files from struts into rr\_lesson\_2 application:

In the struts/contrib/struts-el/lib you should find the following files which you need to add to your **rr\_lesson\_3/WEB-INF directory**.

```
c.tld
struts-bean-el.tld
struts-html-el.tld
struts-logic-el.tld
```

Copy .jar files from struts into rr\_lesson\_2 application:

In the same struts/contrib/struts-el/lib directory, copy the following jar files to the **rr\_lesson\_3/WEB-INF/lib** directory.

```
commons-beanutils.jar
commons-collections.jar
commons-digester.jar
commons-logging.jar
jstl.jar
standard.jar
struts-el.jar
struts.jar
```

(Note we are using the tld files and jars in the contributed struts-el directory since this will force us to use the standard JSTL tags whenever possible).

## LESSON II - 2 - Create web.xml

Create web.xml file in rr\_lesson\_2/WEB-INF/ :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!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>
  <display-name>Struts rr lesson 2</display-name>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>application</param-name>
      <param-value>ApplicationResources</param-value>
    </init-param>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>3</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>3</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/do/*</url-pattern>
  </servlet-mapping>

  <!-- The Welcome File List -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <!-- tag libs -->
  <taglib>
    <taglib-uri>struts/bean-el</taglib-uri>
```

```
<taglib-location>/WEB-INF/struts-bean-el.tld</taglib-location>
</taglib>
```

```
<taglib>
  <taglib-uri>struts/html-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-html-el.tld</taglib-location>
</taglib>
```

```
<taglib>
  <taglib-uri>struts/logic-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic-el.tld</taglib-location>
</taglib>
```

```
<taglib>
  <taglib-uri>jstl/c</taglib-uri>
  <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>
```

```
</web-app>
```

## LESSON II - 3 - Create struts-config.xml

Create struts-config.xml file in rr\_lesson\_2/WEB-INF/ :

```
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd" >
<struts-config>
  <!-- Form Bean Definitions -->
  <form-beans>
    <form-bean name="employeeForm" type="net.reumann.EmployeeForm"/>
  </form-beans>

  <!-- Global forwards -->
  <global-forwards>
    <forward name="error" path="/error.jsp"/>
  </global-forwards>

  <!-- Action Mapping Definitions -->
  <action-mappings>
```

```

    <action path="/setUpEmployeeForm"
    type="net.reumann.SetUpEmployeeAction"
    name="employeeForm"
    scope="request"
    validate="false"
    >
    <forward
        name="continue"
        path="/employeeForm.jsp"/>
</action>

<action path="/insertEmployee"
    type="net.reumann.InsertEmployeeAction"
    name="employeeForm"
    scope="request"
    validate="true"
    input="/employeeForm.jsp"
    >
    <forward
        name="success"
        path="/confirmation.jsp"/>
</action>

</action-mappings>
<!-- message resources -->
<message-resources
    parameter="ApplicationResources"
    null="false" />
</struts-config>

```

Comments: First thing to notice is the addition of a global-forward. Global forwards allow your entire application to share the same forward. In our case since we defined one for "error," any time one of our actions comes up with an error we can have our mapping `findForward("error")` and be forwarded to an error page. If we didn't use a global forward but wanted to always forward to this error page in case of an error, then all of our action mappings would have to declare this forward.

The action mapping: `action path="/setUpEmployeeForm"` calls `SetupEmployeeAction` which will be used to populate some information that our `employeeForm.jsp` will need.

The action mapping: `action path="/insertEmployee"` has `validate="true"` and also has `input="/employeeForm.jsp"`. Setting `validate` equal to `true` will make sure the `validate` method that we will write in our `ActionForm` is called. The `input` attribute will also be necessary so

that if validation returns any errors our application will know what page (or possibly an action) to return to. In this case if validating returns any errors the user is returned back to the same employeeForm.jsp page.

## LESSON II - 4 - Create ActionForm

Create EmployeeForm

```
package net.reumann;
```

```
import org.apache.struts.action.ActionForm;  
import org.apache.struts.action.ActionMapping;  
import org.apache.struts.action.ActionErrors;  
import org.apache.struts.action.ActionError;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
public class EmployeeForm extends ActionForm {
```

```
    private String name;  
    private String age;  
    private String department;
```

```
    public void setName(String name) {  
        this.name = name;  
    }
```

```
    public void setAge(String age) {  
        this.age = age;  
    }
```

```
    public void setDepartment(String department) {  
        this.department = department;  
    }
```

```
    public String getName() {  
        return name;  
    }
```

```
    public String getAge() {  
        return age;  
    }
```

```
    public String getDepartment() {  
        return department;  
    }
```

```

}

public ActionErrors validate( ActionMapping mapping, HttpServletRequest request ) {
    ActionErrors errors = new ActionErrors();

    if ( getName() == null || getName().length() < 1 ) {
        errors.add("name",new ActionError("error.name.required"));
    }

    if ( getAge() == null || getAge().length() < 1 ) {
        errors.add("age",new ActionError("error.age.required"));
    }
    else {
        try {
            Integer.parseInt( getAge() );
        } catch( NumberFormatException ne ) {
            errors.add("age",new ActionError("error.age.integer"));
        }
    }
    return errors;
}
}

```

The EmployeeForm contains a validate method which it inherits from ActionForm. By setting validate to true in the struts-config file we can have our form elements validated here before we get to the Action class. In the case above we are validating to make sure that the user enters a name and an age and that the age is an integer.

The ActionErrors object holds a Map of all the ActionError objects that we add to it. Using some Struts tags we can then easily display these messages nicely on a JSP page (in the case of validation errors the display is usually back on the same JSP form that the user tried to submit).

Looking more closely at adding an ActionError to our ActionErrors object, if we want to add to add an error message if our name does not validate how we want we can do:

```
new ActionError("error.name.required")
```

Later when we create our ApplicationResources.properties you will see the line:

```
error.name.required=Name is required.
```

What happens is our ActionError will look up error.name.required in this ApplicationResources.properties file and display the appropriate message when we using

error/message tags on our JSP pages.

Note: Before you create validate methods in your ActionForms make sure to first check out the next Lesson. There is a much cleaner (and easier) way to handle your form validation which is described in the next lesson.

## LESSON II - 5 - Create EmployeeDTO

We need a Data Transfer Object like we used in Lesson I.

Create EmployeeDTO:

```
package net.reumann;

public class EmployeeDTO {
    private String name;
    private int age;
    private int department;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setDepartment(int department) {
        this.department = department;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public int getDepartment() {
        return department;
    }
}
```

## LESSON II - 6 - Create DepartmentBean

Our JSP page is going to display a list of departments that the user can select from. Create a DepartmentBean to hold the department information: id and description.

Create DepartmentBean:

```
package net.reumann;

public class DepartmentBean {

    private int id;
    private String description;

    public DepartmentBean() {
    }
    public DepartmentBean( int id, String description ) {
        this.id = id;
        this.description = description;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

## LESSON II - 7 - Create DatabaseException

DatabaseException is an Exception that our Model/business layer can throw so we need to create this object.

Create DatabaseException:

```
package net.reumann;

public class DatabaseException extends Exception {
    public DatabaseException() {
```

```

        super();
    }

    public DatabaseException(String message) {
        super(message);
    }
}

```

## LESSON II - 8 - Create EmployeeService

The EmployeeService class is an object that represents our bridge to the Model layer of our architecture. You can build your business/model components in many different ways, using different design patterns. Since Struts really only deals with the View and Controller aspects of an MVC architecture we only need a basic class here to mimic some business requirements that, in real life, we would handle in other components.

The EmployeeService is going to take care of doing the insertEmployee operation and we're also going to add in a method to return a list of DepartmentBeans. I also want to simulate what would happen if an Exception was thrown when calling a method from our service, so a method doInsert() was added in this class which will throw a DatabaseException if you uncomment out the appropriate line.

Create EmployeeService:

```

package net.reumann;

import java.util.Collection;
import java.util.ArrayList;

public class EmployeeService {
    public EmployeeDTO insertEmployee( EmployeeDTO employee ) throws DatabaseException {
        //in real life call other business classes to do insert
        try {
            doInsert( employee );
        }
        catch( DatabaseException de ) {
            //log error
            throw de;
        }
        return employee;
    }
    public Collection getDepartments() {
        //call business layer to return Collection of Department beans
    }
}

```

```

//since we aren't dealing with the model layer, we'll mimic it here
ArrayList list = new ArrayList(3);
list.add( new DepartmentBean( 1, "Accounting"));
list.add( new DepartmentBean( 2, "IT"));
list.add( new DepartmentBean( 3, "Shipping"));
return list;
}

//this wouldn't be in this service class, but would be in some other business class/DAO
private void doInsert( EmployeeDTO employee ) throws DatabaseException {
    //to test an Exception thrown uncomment line below
    //throw new DatabaseException();
}
}

```

## LESSON II - 9 - Create SetUpEmployeeAction

Instead of going directly to a JSP page with a form, you often might want to pre-populate some information the JSP page might need or you may need to pre-populate a corresponding FormBean with some information. One way you can do this is by submitting to an Action class before forwarding to your JSP page.

In lesson 1, we had the following mapping in our struts-config file:

```
<action path="/setupEmployeeForm" forward="/employeeForm.jsp"/>
```

Before we forward directly to the employeeForm.jsp as shown above, we are going to first submit to a SetupEmployeeAction which you see in the mapping:

```

<action path="/setupEmployeeForm"
    type="net.reumann.SetUpEmployeeAction"
    name="employeeForm"
    scope="request"
    validate="false"
    >
    <forward
        name="continue"
        path="/employeeForm.jsp"/>
</action>

```

Create SetUpEmployeeAction:

```
package net.reumann;
```

```

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.util.Collection;

public final class SetUpEmployeeAction extends Action {

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {

        EmployeeService service = new EmployeeService();
        Collection departments = service.getDepartments();
        HttpSession session = request.getSession();
        session.setAttribute( "departments", departments );
        EmployeeForm employeeForm = (EmployeeForm)form;
        employeeForm.setDepartment("2");
        return (mapping.findForward("continue"));
    }
}

```

Notice we have put a Collection of DepartmentBeans into session scope and we also set the department value of our ActionForm to "2".

NOTE: It is not really a good idea to hard code your forwards as String literals like I have been doing. It's better to define these as constants in some constants Interface. If ever you decide to change the names that you use for these forwards in your struts-config.xml file you then only need to change the forward definitions in one other class.

## LESSON II - 10 - Create InsertEmployeeAction

Create InsertEmployeeAction:  
package net.reumann;

```

import org.apache.struts.action.*;
import org.apache.commons.beanutils.BeanUtils;

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

public final class InsertEmployeeAction extends Action {

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        EmployeeService service = new EmployeeService();
        EmployeeForm employeeForm = (EmployeeForm) form;
        EmployeeDTO employeeDTO = new EmployeeDTO();
        BeanUtils.copyProperties( employeeDTO, employeeForm );
        try {
            service.insertEmployee( employeeDTO );
            ActionMessages messages = new ActionMessages();
            ActionMessage message = new
ActionMessage("message.employee.insert.success",employeeDTO.getName() );
            messages.add( ActionMessages.GLOBAL_MESSAGE, message );
            saveMessages( request, messages );
            request.setAttribute("employee",employeeDTO);
            return (mapping.findForward("success"));
        }
        catch( DatabaseException de ) {
            ActionErrors errors = new ActionErrors();
            ActionError error = new ActionError("error.employee.databaseException");
            errors.add( ActionErrors.GLOBAL_ERROR, error );
            saveErrors( request, errors );
            return (mapping.findForward("error"));
        }
    }
}

```

The InsertEmployeeAction is responsible for calling our EmployeeService class and passing it the EmployeeDTO to perform the insert. First we use BeanUtils to copy the properties from the EmployeeForm to our EmployeeDTO. Since insertEmployee() can throw a

DatabaseException we have to handle that.

\*\*\* In the next lesson you will see how Exception handling is greatly improved by using declarative Exception handling, which eliminates the need of try/catch blocks and redundant error handling code.

Notice we use the same type of ActionErrors set up as we did in the EmployeeForm, the only difference being we are adding the error with the key ActionErrors.GLOBAL\_ERROR which is a generic way you can add error messages to display at the top of a results page.

If a DatabaseException is not thrown we are assuming the insert was successful and we set up an ActionMessage and add it to an ActionMessages instance which we will use on our confirmation.jsp page.

## LESSON II - 11 - Create Application Resources

Create ApplicationResources.properties file in rr\_lesson\_2/WEB-INF/classes/ and in src/ :

```
#-- titles --
title.error=ERROR PAGE
title.employeeApp=EMPLOYEE APPLICATION
title.employee.employeeform=EMPLOYEE FORM
title.employee.insert.confirmation=EMPLOYEE INSERT CONFIRMATION

#-- messages --
message.employee.insert.success=You successfully added the employee {0}.

#-- error messages --
error.employee.databaseException=Database error please contact support center.
error.name.required=Name is required.
error.age.required=Age is required.
error.age.integer=Age must be a whole number.

#-- errors headers
errors.validation.header=Validation Error:

#-- buttons --
button.submit=SUBMIT
```

## LESSON II - 12 - Create Style Sheet

Create rr.css file in rr\_lesson\_2/ :

```
body { font-family: arial,sans-serif;
```

```

        color: black;
        background-color: #F1FFD2;
        font-size: 13pt;
    }
    #success { color: green; font-weight: bold; }
    #error { color: red; font-weight: bold; }
    #errorsHeader { color: red; font-weight: bold; font-size: 14pt; }

```

## LESSON II - 13 - Create index.jsp

Create index.jsp file in rr\_lesson\_2/ :

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employeeApp"/></title>
</head>
<body>
<h1><bean:message key="title.employeeApp"/></h1>
<br>
<html:link page="/do/setupEmployeeForm">Add An Employee</html:link>
</body>
</html>

```

## LESSON II - 14 - Create employeeForm.jsp

Create employeForm.jsp in rr\_lesson\_2/ :

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/logic-el" prefix="logic" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employee.employeeform"/></title>
</head>
<body>

<h1><bean:message key="title.employee.employeeform"/></h1>

```

```

<logic:messagesPresent>
  <span id="errorsHeader"><bean:message key="errors.validation.header"/></span>
  <html:messages id="error">
    <li><c:out value="{error}"/></li>
  </html:messages>
  <hr>
</logic:messagesPresent>

<html:form action="insertEmployee" focus="name">
<table>
<tr>
  <td >Name:</td>
  <td><html:text property="name"/></td>
</tr>
<tr>
  <td>Age:</td>
  <td><html:text property="age"/></td>
</tr>
<tr>
  <td>Department:</td>
  <td>
    <html:select name="employeeForm" property="department">
      <html:options collection="departments" property="id" labelProperty="description"/>
    </html:select>
  </td>
</tr>
</table>
<html:submit><bean:message key="button.submit"/></html:submit>
</html:form>
</body>
</html>

```

Few new things introduced here. First notice how we are displaying error messages (ActionError messages we created). The logic:messagesPresent tag checks to see if any ActionMessages are in scope (remember ActionErrors is a subclass of ActionMessages). The html:messages tag then will loop through all messages in scope. Setting the id="error" just gives us a handle to use for our JSTL c:out tag to display the message in the current iteration of our loop.

Note: You'll see some Struts developers simply use the <html:errors/> tag to display error messages. This does work well and the tag will also look in your resources file for an

errors.header and errors.footer definition and use them as a header and footer for your error message display. You'll also have to add some HTML code such as <LI> or <BR> before all of your validation errors definitions in the resources file. Personally I'd rather have a bit more code as above and keep the formatting of HTML done by style sheets vs having HTML formatting code in your resources file.

The Department form field provides a Select box and we simply use the struts html:select and html:options tags to generate our options. The html:select tag will set the initial selection to whatever the department property is set to in our EmployeeForm (remember we set it to "2" in our SetUpEmployeeAction). The select tag will also populate the department property in our EmployeeForm with whatever option the user selects.

html:options will loop through whatever collection we provide. In this case we had put the Collection departments into request scope in our EmployeeSetUpAction. The property="id" portion will set the value of the option tag for that iteration to the id property in the current DepartmentBean and the labelValue will set what the user sees- in this case the 'description' field in the current DepartmentBean. Just view the source code at runtime and you'll see what gets generated from this JSP based on the select and options tags.

Finally, to test how the validation works try not entering a name and/or age. (Also try entering a non-number for age).

## LESSON II - 15 - Create confirmation.jsp

Create confirmation.jsp in rr\_lesson\_2 /:

```
<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/logic-el" prefix="logic" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<jsp:useBean id="employee" scope="request" class="net.reumann.EmployeeDTO"/>
<html>
<head>
<title><bean:message key="title.employee.insert.confirmation"/></title>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
</head>
<body>
<h1><bean:message key="title.employee.insert.confirmation"/></h1>
<br>
<logic:messagesPresent message="true">
  <html:messages id="message" message="true">
    <span id="success"><c:out value="{message}"/></span><br>
  </html:messages>
</logic:messagesPresent>
```

```

<br><br>
NAME: <c:out value="\${employee.name}"/><br>
AGE: <c:out value="\${employee.age}"/>
</body>
</html>

```

Here we will display the ActionMessages created in our InsertEmployeeAction. It works the same way as it does on the employeeForm.jsp except we added message="true" to the logic:messagesPresent portion. We need to add message="true" so that the tag will look for ActionMessages and not actionErrors.

If this was going to be a generic confirmation page used for other processes, we would remove the code specific to Employee such as the title and the NAME and AGE displays. Then we could use this page to display whatever message we wanted by just setting up an appropriate ActionMessage before forwarding to this page.

## LESSON II - 16 - Create error.jsp

Create error.jsp in rr\_lesson\_2/ :

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.error"/></title>
</head>
<body>
<h1><bean:message key="title.error"/></h1>

```

You have reached this page because of the following error(s):<br>

```

<br>

<html:messages id="error">
  <span id="error"><c:out value="\${error}"/></span><br>
</html:messages>

</body>
</html>

```

This page is really simple. Here we just use the <html:messages> tag to loop through and display the error messages that are in the ActionErrors object in scope. I didn't bother to use <logic:messagesPresent> construct since I'm not using headers or anything that could be

displayed even if an `ActionError` was not present.

This ends Lesson II. I hope you found it helpful. Please send any comments or suggestions to [struts@reumann.net](mailto:struts@reumann.net).

## LESSON III - Introduction

This lesson assumes you understand the basic of Struts. If you are new to Struts you should start with Lesson I and Lesson II.

Concepts introduced in Lesson III:

DynaActionForm (DynaValidatorForm)  
Declarative Exception Handling  
Validation made easy  
html:multibox tag

This application mimics the business requirement of inserting and updating an employee. The flow of this example doesn't mimic what you would really have set up in real life, since in real life you'd probably choose from a list of employees to update. This example, since it doesn't do any real business logic, simply lets you update the employee you just pretended to insert.

This Lesson is a lot like Lesson II in functionality except the flow is a bit different. After an insert of an Employee you are forwarded to the filled out employeeForm and you could resubmit it as an update. The confirmation page that you saw in Lesson II now is displayed only after you do an update.

[Download rr\\_lesson\\_3.war](#)

Begin lesson now by clicking [START](#).

## LESSON III - 1 - Setup

These Lessons assume you are using Tomcat 4.1 and have downloaded Struts 1.1-rc2 or greater.

Create following directory structure under tomcat:

webapps

```
|
rr_lesson_2
|
--- WEB-INF
|
|--- classes
| |
|   --- net
|     |
```

```

|         -- reumann
|--- lib
|
|--- src
|
|         --- net
|
|         -- reumann

```

Add The Appropriate Struts Files:

In the struts/contrib/struts-el/lib you should find the following files which you need to add to your **rr\_lesson\_3/WEB-INF** directory:

```

c.tld
struts-bean-el.tld
struts-html-el.tld
struts-logic-el.tld

```

From the same struts/contrib/struts-el/lib copy the following jar files to your **rr\_lesson\_3/WEB-INF/lib** directory:

```

commons-beanutils.jar
commons-collections.jar
commons-digester.jar
commons-logging.jar
commons-validator.jar
jstl.jar
standard.jar
struts-el.jar
struts.jar

```

(Note we are using the tld files and jars in the contributed struts-el directory since this will force us to use the standard JSTL tags whenever possible).

In the main struts/lib directory you'll find:

```

validation-rules.xml
commons-validator.jar

```

Both of these files need to also be added to your **rr\_lesson\_3/WEB-INF/lib** directory.

Create web.xml:

Finally, we need to make sure we have a web.xml file in our **rr\_lesson\_3/WEB-INF/**

directory. The web.xml looks exactly the same as it did in the other two lessons except that we changed the display name to rr lesson 3.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!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>
```

```
  <display-name>Struts rr lesson 3</display-name>
```

```
  <servlet>
```

```
    <servlet-name>action</servlet-name>
```

```
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
```

```
    <init-param>
```

```
      <param-name>application</param-name>
```

```
      <param-value>ApplicationResources</param-value>
```

```
    </init-param>
```

```
    <init-param>
```

```
      <param-name>config</param-name>
```

```
      <param-value>/WEB-INF/struts-config.xml</param-value>
```

```
    </init-param>
```

```
    <init-param>
```

```
      <param-name>debug</param-name>
```

```
      <param-value>3</param-value>
```

```
    </init-param>
```

```
    <init-param>
```

```
      <param-name>detail</param-name>
```

```
      <param-value>3</param-value>
```

```
    </init-param>
```

```
    <load-on-startup>1</load-on-startup>
```

```
</servlet>
```

```
<!-- Action Servlet Mapping -->
```

```
<servlet-mapping>
```

```
  <servlet-name>action</servlet-name>
```

```
  <url-pattern>/do/*</url-pattern>
```

```
</servlet-mapping>
```

```
<!-- The Welcome File List -->
```

```

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

<!-- tag libs -->
<taglib>
  <taglib-uri>struts/bean-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts/html-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-html-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts/logic-el</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic-el.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>jstl/c</taglib-uri>
  <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>

</web-app>

```

## LESSON III - 2 - Create Beans

We need a Data Transfer Object to store the Employee information that our business layer can use (fields with the correct data types as opposed to just String properties that ActionForm beans hold). We also need to create a few other objects. Create all of these objects as listed below:

EmployeeDTO:

```
package net.reumann;
```

```
public class EmployeeDTO {
  private String name;
  private int age;
  private int department;
}
```

```

private String[] flavorIDs;

public void setName(String name) {
    this.name = name;
}
public void setAge(int age) {
    this.age = age;
}
public void setDepartment(int department) {
    this.department = department;
}
public void setFlavorIDs(String[] flavorIDs) {
    this.flavorIDs = flavorIDs;
}

public String getName() {
    return name;
}
public int getAge() {
    return age;
}
public int getDepartment() {
    return department;
}
public String[] getFlavorIDs() {
    return flavorIDs;
}
}

```

DepartmentBean:

```
package net.reumann;
```

```

public class DepartmentBean {

    private int id;
    private String description;

    public DepartmentBean() {
    }
}

```

```

public DepartmentBean( int id, String description ) {
    this.id = id;
    this.description = description;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getDescription() {
    return description;
}
public void setDescription(String description) {
    this.description = description;
}
}

```

FlavorBean:

```

package net.reumann;

```

```

public class FlavorBean {
    private String flavorID;
    private String description;

    public FlavorBean() {}
    public FlavorBean( String flavorID, String description ) {
        this.flavorID = flavorID;
        this.description = description;
    }
    public void setFlavorID(String flavorID) {
        this.flavorID = flavorID;
    }
    public String getFlavorID() {
        return flavorID;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}

```

```

    public String getDescription() {
        return description;
    }
}

```

DatabaseException:  
package net.reumann;

```

public class DatabaseException extends Exception {
    public DatabaseException() {
        super();
    }
    public DatabaseException(String message) {
        super(message);
    }
}

```

## LESSON III - 3 - Create EmployeeService

The EmployeeService class is an object that represents our bridge to the Model layer of our architecture. You can build your business/model components in many different ways, using many different design patterns. Since Struts really only deals with the View and Controller aspects of an MVC architecture we only need a basic class here to mimic some business requirements that, in real life, we would handle using other components.

The EmployeeService is going to take care of doing our Employee insert and update operation. For simplicity's sake we are also adding a method to return a Collection of departments and another to return a Collection of flavors. You can also test what happens when a DatabaseException is thrown by uncommenting out the appropriate line in either the mimicInsert or mimicUpdate methods.

EmployeeService:  
package net.reumann;

```

import java.util.Collection;
import java.util.ArrayList;

```

```

public class EmployeeService {
    public EmployeeDTO insertEmployee( EmployeeDTO employee ) throws DatabaseException {
        //in real life call other business classes to do insert
        try {
            mimicInsert( employee );

```

```

    }
    catch( DatabaseException de ) {
        //log error
        throw de;
    }
    return employee;
}

public EmployeeDTO updateEmployee( EmployeeDTO employee ) throws DatabaseException {
    //in real life call other business classes to do update
    try {
        mimicUpdate( employee );
    }
    catch( DatabaseException de ) {
        //log error
        throw de;
    }
    return employee;
}

//this wouldn't be in this service class, but would be in some other business class/DAO
private void mimicInsert( EmployeeDTO employee ) throws DatabaseException {
    //to test an Exception thrown uncomment line below
    //throw new DatabaseException("Error trying to insert Employee");
}

//this wouldn't be in this service class, but would be in some other business class/DAO
private void mimicUpdate( EmployeeDTO employee ) throws DatabaseException {
    //to test an Exception thrown uncomment line below
    //throw new DatabaseException("Error trying to update Employee");
}

public Collection getDepartments() {
    //call business layer to return Collection of Department beans
    //since we aren't dealing with the model layer, we'll mimic it here
    ArrayList list = new ArrayList(3);
    list.add( new DepartmentBean( 1, "Accounting" ));
    list.add( new DepartmentBean( 2, "IT" ));
    list.add( new DepartmentBean( 3, "Shipping" ));
    return list;
}

```

```

public Collection getFlavors() {
    //call business layer to return Collection of Flavors
    //since we aren't dealing with the model layer, we'll mimic it here
    ArrayList list = new ArrayList(3);
    list.add( new FlavorBean( "101", "Chocolate"));
    list.add( new FlavorBean( "201", "Vanilla"));
    list.add( new FlavorBean( "500", "Strawberry"));
    return list;
}
}

```

## LESSON III - 4 - Setup DynaActionForm (DynaValidatorForm)

In previous lessons we created an EmployeeForm object which extended the Struts ActionForm. There is, however, a much easier way to create ActionForm objects by using DynaActionForm which allows you to configure the bean in the struts-config.xml file. We are going to use a subclass of DynaActionForm called a DynaValidatorForm which provides greater functionality when used with the validation framework.

As we proceed through the rest of these steps we'll be building the struts-config.xml file as we go along.

Create first part of struts-config.xml (declare DynaValidatorForm ):

```

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd" >
<struts-config>
  <!-- Form Bean Definitions -->
  <form-beans>
    <form-bean name="employeeForm" type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="name" type="java.lang.String"/>
      <form-property name="age" type="java.lang.String"/>
      <form-property name="department" type="java.lang.String" initial="2" />
      <form-property name="flavorIDs" type="java.lang.String[]"/>
      <form-property name="methodToCall" type="java.lang.String"/>
    </form-bean>
  </form-beans>

```

Looking at the form-bean definition above you see that we are using type org.apache.struts.validator.DynaValidatorForm. Magically Struts will make sure to create this bean for us using the form-property fields defined. Since we are actually going to capture more than one ice cream flavor that an employee likes, flavorIDs is defined as a String array. The "methodToCall" property is a property that we are going to use in our DispatchAction

class (more about that later). Remember, just like when using the standard ActionForm, you should keep these form fields set as Strings since an HTML form will only submit the request information in String format.

This DynaValidatorForm is used just like a normal ActionForm when it comes to how we define it's use in our action mappings. The only 'tricky' thing is that standard getter and setters are not made since the DynaActionForms are backed by a HashMap. In order to get fields out of the DynaActionForm you would do:

```
String age = formBean.get("age");
```

Similarly, if you need to define a property:

```
formBean.set("age", "33");
```

You'll see an example of this usage in the SetUpEmployeeAction in the next step. If you want, though, you could set initial values for the properties as you see done above for the department property. This will always set the initial value of the department id to "2."

## LESSON III - 5 - Create SetUpEmployeeAction

The SetUpEmployeeAction will map to /setUpEmployeeAction. First let's continue adding to the struts-config.xml file.

Add global-forwards and /setUpEmployeeForm mapping:

```
<!-- Global forwards -->
<global-forwards>
  <forward name="error" path="/error.jsp"/>
</global-forwards>

<!-- Action Mapping Definitions -->
<action-mappings>

  <action path="/setUpEmployeeForm"
    type="net.reumann.SetUpEmployeeAction"
    name="employeeForm"
    scope="request"
    validate="false">
    <forward
      name="continue"
      path="/employeeForm.jsp"/>
  </action>
```

Our first action-mapping directs us to the SetUpEmployeeAction. If you looked at Lesson 2 you will see that this mapping is almost exactly the same. The only difference is behind the

scenes where "employeeForm" is now a DynaValidatorForm instead of the basic ActionForm. Now we can create the actual SetUpEmployeeAction.

Create SetUpEmployeeAction:

```
package net.reumann;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.DynaActionForm;

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

import java.util.Collection;

public final class SetUpEmployeeAction extends Action {

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {

        EmployeeService service = new EmployeeService();
        Collection departments = service.getDepartments();
        Collection flavors = service.getFlavors();
        HttpSession session = request.getSession();
        session.setAttribute( "departments", departments );
        session.setAttribute( "flavors", flavors );
        DynaActionForm employeeForm = (DynaActionForm)form;
        employeeForm.set( "methodToCall", "insert" );
        return (mapping.findForward("continue"));
    }
}
```

This Action is almost identical to the SetUpEmployeeAction in Lessons I and II. Pay attention to the cast of the ActionForm into the type DynaActionForm and the setting of the

'methodToCall' property:

```
DynaActionForm employeeForm = (DynaActionForm)form; employeeForm.set(
"methodToCall", "insert" );
```

The methodToCall being set to "insert" will make more sense after the next step in our lesson. (What we are going to do is use this property to set a hidden form field to "insert" so that our DispatchAction will know to do an insert when the employeeForm.jsp is submitted). Also note that we are not setting the 'department' property here since we set up that an initial value in our DynaValidatorForm.

NOTE: It's not really a good idea to hard code your forwards as String literals like I have been doing. It's better to define these as constants in some constants Interface. If ever you decide to change the names that you use for these forwards in your struts-config.xml file you then only need to change the forward definitions in one other class.

## LESSON III - 6 - DispatchAction / Declarative Exceptions

Normally you set up each action event to always correspond to a unique Action class. There is nothing wrong with doing this but if you want to condense related events into one Action class you can do so using a DispatchAction (or one of its subclasses like LookupDispatchAction). Typical CRUD events (create, retrieve, update, and delete) can be seen as related events. In this example we have an option for inserting an employee and also for updating an employee. We could have just kept the Lesson II InsertEmployeeAction and then created another UpdateEmployeeAction for handling the update request, but instead we are going to put both of these events into one EmployeeDispatchAction.

## LESSON III - 7 - Create Application Resources

Create the ApplicationResources.properties file and place in rr\_lesson\_3/WEB-INF/classes directory. Also a good idea to include it in in rr\_lesson\_3/WEB-INF/src directory as well.

Create ApplicationResources.properties:

`#-- titles`

`title.error=ERROR PAGE`

`title.employeeApp=EMPLOYEE APPLICATION`

`title.employee.employeeform=EMPLOYEE FORM`

`title.employee.confirmation=CONFIRMATION`

`title.employee.insert=ADD AN EMPLOYEE`

`title.employee.update=UPDATE AN EMPLOYEE`

`#--display names`

`name.displayname=Name`

`age.displayname=Age`

```

#-- messages
message.employee.update.success=Successful update of Employee.
message.employee.insert.success=Successfully added Employee.

#-- exceptions
exception.database.error=Problem occurred when performing database operation.
Error Message: {0}

#-- validation errors
errors.required={0} is required.
errors.integer={0} must be a whole number.

#-- errors headers
errors.validation.header=Validation Error:

#-- buttons --
button.submit=SUBMIT
button.update=UPDATE

```

Notice the `exception.database.error` declaration. This is the key we provided in our declarative exception for `DatabaseException` in the `struts-config.xml` file. The actual error message itself will be passed in to the `{0}` argument.

The next step will explain our validation framework. Take a look at the two validation error declarations above. You'll see how those fit in after reviewing the next step.

## LESSON III - 8 - Validation Made Easy

Using the `org.apache.struts.validator` package makes validating very simple and efficient. A default `validation-rules.xml` file comes with Struts which provides rules for validating many common situations. In order to use the framework you need to create a `validation.xml` file and define that you are using the validator plugin in the `struts-config.xml`.

Complete the `struts-config.xml` file:

```

<!-- message resources -->
<message-resources
  parameter="ApplicationResources"
  null="false" />

<!-- plugins -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">

```

```
<set-property property="pathnames" value="/WEB-INF/validator-rules.xml,  
        /WEB-INF/validation.xml"/>  
  
</plug-in>
```

```
</struts-config>
```

Create validation.xml file and place it in rr\_lesson\_3/WEB-INF/:

```
<form-validation>  
  <formset>  
    <form name="employeeForm">  
      <field property="name"  
        depends="required">  
        <arg0 key="name.displayname"/>  
      </field>  
      <field property="age"  
        depends="required,integer">  
        <arg0 key="age.displayname"/>  
      </field>  
    </form>  
  </formset>  
</form-validation>
```

Setting `validate="true"` for an action-mapping to `employeeForm` will execute validation based on the fields declared in the `validation.xml` file. The validation above declares that we are going to validate "name" and "age." The possible values for the 'depends' attributes are those that are declared in the `validation-rules.xml`. (Of course you can add your own plugins to the framework as well, but most of the basic validations are already defined for you). In our example we are defining both 'name' and 'age' with the 'required' value, and for age we are also providing 'integer.' Upon submission of the `employeeForm` both fields will be checked that they are not null or left blank and age will also be checked to be sure it is a valid number.

One thing you have to make sure of is that you provide the necessary key/value pairs in your `ApplicationResources` file for the validation errors that could occur. If you go back to step 7 you see we defined:

```
errors.required={0} is required.  
errors.integer={0} must be a whole number.
```

These are global errors that we could use for all of our validations. The key that we provide for `arg0` in the `validation.xml` file definitions will substitute the proper value in place of `{0}` if an error occurs. (In the `ApplicationResources` file you will see `name.displayname` and `age.displayname` defined and those values will be substituted for `{0}` in the appropriate validation).

## LESSON III - 9 - employeeForm.jsp

The employeeForm.jsp is going to be used for both our employee insert and employee update. You will see logic tags ( c:choose ) for different information to display based on whether the form is used for an insert or an update. Our Action classes are responsible for setting the hidden form field "methodToCall" to either "insert" or "update" before the user is forwarded to this form. You will also see the use of the Strut's html:multibox tag which is a great tag for setting up multiple checkboxes.

Note: checkboxes can be tricky if your form bean will ever have session scope. If that is the case, you will need to reset all checkboxes to false in the reset method of your form bean. Using a DynaActionForm this means you will have to use a subclass of DynaActionForm and create the reset method in this class. You'll then use this subclass as your type definition in defining the DynaActionForm properties as usual in your struts-config.xml file.

Create employeeForm.jsp:

```
<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="struts/logic-el" prefix="logic" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<html>
<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employee.employeeform"/></title>
</head>
<body>

<c:choose>
  <c:when test="\${employeeForm.map.methodToCall == 'insert'}">
    <h1><bean:message key="title.employee.insert"/></h1>
  </c:when>
  <c:otherwise>
    <h1><bean:message key="title.employee.update"/></h1>
  </c:otherwise>
</c:choose>

<logic:messagesPresent>
  <span id="errorsHeader"><bean:message key="errors.validation.header"/></span>
  <html:messages id="error">
    <li><c:out value="\${error}"/></li>
  </html:messages>
<hr>
```

```
</logic:messagesPresent>
```

```
<logic:messagesPresent message="true">
```

```
  <html:messages id="message" message="true">
```

```
    <span id="success"><c:out value="{message}"/></span><br>
```

```
  </html:messages>
```

```
</logic:messagesPresent>
```

```
<html:form action="employeeAction" focus="name">
```

```
<table>
```

```
<tr>
```

```
  <td >Name:</td>
```

```
  <td><html:text property="name"/></td>
```

```
</tr>
```

```
<tr>
```

```
  <td>Age:</td>
```

```
  <td><html:text property="age"/></td>
```

```
</tr>
```

```
<tr>
```

```
  <td>Department:</td>
```

```
  <td>
```

```
    <html:select name="employeeForm" property="department">
```

```
      <html:options collection="departments" property="id" labelProperty="description"/>
```

```
    </html:select>
```

```
  </td>
```

```
</tr>
```

```
<tr>
```

```
  <td>Favorite ice cream flavors:</td>
```

```
  <td>
```

```
    <c:forEach var="flavor" items="{flavors}">
```

```
      <html:multibox name="employeeForm" property="flavorIDs">
```

```
        <c:out value="{flavor.flavorID}"/>
```

```
      </html:multibox>
```

```
      <c:out value="{flavor.description}"/>
```

```
    </c:forEach>
```

```
  </td>
```

```
</tr>
```

```
</table>
```

```
<html:hidden name="employeeForm" property="methodToCall"/>
```

```

<c:choose>
  <c:when test="\${employeeForm.map.methodToCall == 'insert'}">
    <html:submit><bean:message key="button.submit"/></html:submit>
  </c:when>
  <c:otherwise>
    <html:submit><bean:message key="button.update"/></html:submit>
  </c:otherwise>
</c:choose>

</html:form>
</body>
</html>

```

You might be wondering why there are two different versions of the `logic:messagesPresent` tag blocks. The reason is that one set is used to display `ActionErrors` which, in this example could be on the page when validation errors are present, and the other other set is used to display normal `ActionMessages` which, in this case, would be a successful insert message. By adding `message="true"` to the `messages` tags we make sure that only `ActionMessages` are displayed. (Leaving out the `message="true"` means only `ActionErrors` will be displayed if present).

## LESSON III - 10 - Create other JSPs

Adding these remaining files completes our application and lesson. Hopefully you found the lesson helpful. These remaining files are all placed in `rr_lesson_3/` and should be easy to understand. Pay attention to the `html:messages` tags.

`rr.css:`

```

body { font-family: arial,sans-serif;
  color: black;
  background-color: #F1FFD2;
  font-size: 12pt;
}
#success { color: green; font-weight: bold; }
#error { color: red; font-weight: bold; }
#errorsHeader { color: red; font-weight: bold; font-size: 13pt; }

```

`index.jsp:`

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<html>

```

```

<head>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
<title><bean:message key="title.employeeApp"/></title>
</head>
<body>
<h1><bean:message key="title.employeeApp"/></h1>
<br>
<html:link page="/do/setupEmployeeForm">Add An Employee</html:link>
</body>
</html>

```

confirmation.jsp:

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="struts/logic-el" prefix="logic" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<jsp:useBean id="employee" scope="request" class="net.reumann.EmployeeDTO"/>
<html>
<head>
<title><bean:message key="title.employee.confirmation"/></title>
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
</head>
<body>
<h1><bean:message key="title.employee.confirmation"/></h1>
<br>
<logic:messagesPresent message="true">
  <html:messages id="message" message="true">
    <span id="success"><c:out value="{message}"/></span><br>
  </html:messages>
</logic:messagesPresent>
</body>
</html>

```

error.jsp:

```

<%@ taglib uri="struts/bean-el" prefix="bean" %>
<%@ taglib uri="struts/html-el" prefix="html" %>
<%@ taglib uri="jstl/c" prefix="c" %>
<html>
<head>

```

```
<link href="<html:rewrite page="/rr.css" />" rel="stylesheet" type="text/css">
```

```
<title><bean:message key="title.error"/></title>
```

```
</head>
```

```
<body>
```

```
<h1><bean:message key="title.error"/></h1>
```

```
<html:messages id="error">
```

```
  <span id="error"><c:out value="{error}" escapeXml="false"/></span><br>
```

```
</html:messages>
```

```
</body>
```

```
</html>
```

## LESSON Struts And iBATIS - Introduction

Probably the most difficult decision you have to make as an architect/developer is deciding on what kind of persistence framework you are going to use. Struts really only deals with the View and Controller parts of your MVC architecture. You still are left to determine how your data is going to persist in the model layer (normally a relational database back-end).

There are many different ways you can architect your model layer and this demo only presents one of many possible ways it can be done. The persistence framework in this example uses iBATIS created by Clinton Begin (<http://www.ibatis.com/>). iBATIS is just nothing short of incredible. After going over the few pages in this lesson and looking at the source code in the war file provided, I think you will see how easy it is to utilize this powerful framework. All of the annoying, difficult parts you would have to worry about in creating your own decent database mapping framework are taken care of by the iBATIS layer (reflection, caching).

Another great thing about using iBATIS is it comes with some of the best documentation I have ever seen. (Amazing when you also consider it is an open source project! - open source and good documentation?:). The documentation is so good that I'm not going to waste a lot of time explaining much in this lesson since the iBATIS documentation covers everything. I'll provide some brief comments but the main reason for this lesson it to provide an example of how you could utilize iBATIS from Struts. I'll emphasize again, this example just demonstrates one of many ways you could do this. Also, in keeping in line with the other lessons, I wanted to provide a 'simple' example that people could easily follow. Although simple in design, it is still a decent architecture to use. (A super complex application does not necessarily equate to "better").

After going through this lesson, I strongly suggest you check out the Mac-Daddy of examples "JPetStore" found at <http://www.ibatis.com/> The JPetStore3 application is incredible (and it uses Struts as well). JPetStore demonstrates a much more robust model (using a DaoManager and showing how to use multiple DataSources). Before you plan to code a complex, production-ready Struts application using iBATIS, I'd seriously consider implementing the Dao framework found on the iBATIS site. The latest DAO manager will actually be flexible enough to allow you to swap in other persistence frameworks if you decide iBATIS is not right for you.

This lesson assumes you are familiar with Struts. Unlike lessons 1 through 3, this is not a walk through building the application. If you are completely new to Struts, do lessons 1 through 3. I'm only discussing the pertinent parts related to iBATIS in this lesson.

[Download rr\\_lesson\\_ibatis.zip](#)

[Continue To Step 1 - Setup.](#)

## LESSON Struts And iBATIS - Setup

This lesson assumes you are using Tomcat 4.1 and that you understand the basics of Struts.

Download the `rr_lesson_ibatis.zip` archive and unzip inside your `{tomcat}/webapps` directory. [rr\\_lesson\\_ibatis.zip](#)

The application requires a Database to connect to. You need to create and populate the three small tables listed in `tables_script.sql`. The script will work as is for postgres. Tweak accordingly for your db set up. Make sure you also place your driver in `tomcat/common/lib` (If you don't have a database set up you could of course just view the source code without running the application).

There is also a `log4j.properties` file in `src` and classes which you can configure to your liking. Just add the correct path to where you want the log file to be built.

The Ant build file will also work if you change the path to where the `servlet.jar` is located in the build file. Obviously you only need this if you want to tinker around and recompile the code.

## LESSON Struts And iBATIS - Configure your DataSource

There are various ways a web application can implement a DataSource for iBATIS to use. I like to use the container (in this case Tomcat) to handle this. iBATIS works perfectly with grabbing the connections it needs from your container's DataSource setup.

In Tomcat's `server.xml` file define the Datasource resource in a Context for this application. (If you do not want to put this code in your `server.xml` file check the iBATIS docs for other ways to define a DataSource).

(NOTE: The JPetStore demo shows a more powerful and flexible way to deal with multiple DataSources).

The important things you will have to change below are the username, password, and url to your database.

```
<Context path="/rr_lesson_ibatis" docBase="rr_lesson_ibatis" debug="0" reloadable="true" crossContext="true">
```

```
  <Resource name="lessonIbatisDatasource" auth="Container" type="javax.sql.DataSource"/>
```

```
  <ResourceParams name="lessonIbatisDatasource">
```

```
    <parameter>
```

```
      <name>password</name>
```

```
      <value>passwordForDB</value>
```

```
    </parameter>
```

```
    <parameter>
```

```
      <name>url</name>
```

```
      <value>jdbc:postgresql://127.0.0.1:5432/yourDatabaseName</value>
```

```
    </parameter>
```

```
  </parameter>
```

```

        <name>driverClassName</name>
        <value>org.postgresql.Driver</value>
    </parameter>
    <parameter>
        <name>username</name>
        <value>userNameForDB</value>
    </parameter>
    <parameter>
        <name>factory</name>
        <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
        <name>maxIdle</name>
        <value>1</value>
    </parameter>
    <parameter>
        <name>maxActive</name>
        <value>3</value>
    </parameter>
    <parameter>
        <name>maxWait</name>
        <value>5000</value>
    </parameter>
    <parameter>
        <name>removeAbandoned</name>
        <value>true</value>
    </parameter>
    <parameter>
        <name>removeAbandonedTimeout</name>
        <value>20</value>
    </parameter>
</ResourceParams>

```

```
</Context>
```

If you change the name of this DataSource to something other than lessonIbatisDatasource make sure to change it also in the resource-ref definition of this lesson's web.xml file:

```

<resource-ref>
    <res-ref-name>lessonIbatisDatasource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>

```

## LESSON Struts And iBATIS - The sql-map-config.xml File

For a simple application where you are only going to need to connect to one database, you will usually only need one of these files. (See JPetStore for connection to different DataSources or wait until the next lesson:). Just like the struts-config file does not need to be called 'struts-config' this configuration file does not have to be named 'sql-map-config.xml.'

Calling it something more meaningful is a good idea if you are going to have to need multiple config files.

Below is the sql-map-config file:

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

<!DOCTYPE sql-map-config
  PUBLIC "-//iBATIS.com//DTD SQL Map Config 1.0//EN"
  "http://www.ibatis.com/dtd/sql-map-config.dtd">

<sql-map-config>

  <properties resource="net/reumann/conf/database.properties" />

  <datasource name="lessonIbatisDatasource"
    factory-class="com.ibatis.db.sqlmap.datasource.JndiDataSourceFactory"
    default="true" >
    <property name="DBInitialContext" value="java:comp/env" />
    <property name="DBLookup" value="{DatabaseJNDIPath}" />
  </datasource>

  <sql-map resource="net/reumann/sql/EmployeeSQL.xml" />
  <sql-map resource="net/reumann/sql/LabelValueSQL.xml" />

</sql-map-config>
```

First, notice the declaration of a properties file. Using a properties file is a good idea since if you change from production to development. You could have different properties files versus having to have separate sql-config files. In this demo we are using the `{DatabaseJNDIPath}` key which is defined in the `net/reumann/conf/database.properties` file. You can also check out the example in the iBATIS docs to see how to set up a Datasource directly in this configuration file instead of setting it up in the container's configuration file. In that example you'll see the benefits using property files to define such things as the driver name, connection URL, etc. Since we set up our DataSource in Tomcat's server.xml file there are not too many things to substitute using a properties file.

There are also two declarations of sql-maps:

```
<sql-map resource="net/reumann/sql/EmployeeSQL.xml" />
<sql-map resource="net/reumann/sql/LabelValueSQL.xml" />
```

These xml files are the ones where our actual SQL statements will be declared. We will examine EmployeeSQL.xml next.

## LESSON Struts And iBATIS - EmployeeSQL

The iBATIS documentation on configuring sql maps is excellent so read it!

The main SQL map used in this application is EmployeeSQL and is listed below with some comments following the listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sql-map PUBLIC "-//iBATIS.com//DTD SQL Map 1.0//EN"
"http://www.ibatis.com/dtd/sql-map.dtd">
```

```
<sql-map name="EmployeeSQL">
```

```
  <cache-model name="employee_cache" reference-type="WEAK">
    <flush-interval hours="24"/>
    <flush-on-execute statement="insertEmployee" />
    <flush-on-execute statement="updateEmployee" />
  </cache-model>
```

```
  <result-map name="employee_result" class="net.reumann.Employee">
    <property name="id" column="emp_id"/>
    <property name="name" column="name"/>
    <property name="deptId" column="dept_id"/>
    <property name="deptName" column="dept_name"/>
    <property name="location" column="location_id" mapped-
statement="getLocationById"/>
  </result-map>
```

```
  <mapped-statement name="insertEmployee">
    INSERT INTO employee ( id, name, dept_id, location_id )
    VALUES ( #id#, #name#, #deptId#, #locationId# )
  </mapped-statement>
```

```
  <mapped-statement name="updateEmployee">
    UPDATE employee SET name = #name#, dept_id = #deptId#, location_id =
#locationId#
    WHERE id = #id#
  </mapped-statement>
```

```
  <mapped-statement name="getLocationById" result-class="net.reumann.Location">
    SELECT id, name FROM location where id = #value#
  </mapped-statement>
```

```
  <dynamic-mapped-statement name="selectEmployeesFromSearch" result-
map="employee_result" cache-model="employee_cache">
    SELECT employee.id AS emp_id,
    employee.name AS name,
    employee.dept_id AS dept_id,
    department.name AS dept_name,
    employee.location_id AS location_id,
    location.name AS location_name
  FROM
    employee, department, location
  WHERE
    employee.dept_id = department.id AND
```

```

        employee.location_id = location.id
    </dynamic>
    <isNotEmpty prepend=" AND " property="id">
        employee.id = #id#
    </isNotEmpty>
    <isNotEmpty prepend=" AND " property="name">
        employee.name = #name#
    </isNotEmpty>
    <isNotEmpty prepend=" AND " property="deptId">
        employee.dept_id = #deptId#
    </isNotEmpty>
    <isNotEmpty prepend=" AND " property="locationId">
        employee.location_id = #locationId#
    </isNotEmpty>
</dynamic>
</dynamic-mapped-statement>
</sql-map>

```

## COMMENTS:

The first thing you'll notice is the set up of a cache model:

```

<cache-model name="employee_cache" reference-type="WEAK">
    <flush-interval hours="24"/>
    <flush-on-execute statement="insertEmployee" />
    <flush-on-execute statement="updateEmployee" />
</cache-model>

```

Read the iBATIS docs concerning the different cache types. For the majority of the SQL maps "WEAK" will do just fine. Remember iBATIS takes care of caching results for you so you don't have to worry about managing that. Notice that when the insertEmployee or updateEmployee mapped-statement is called the cache is cleared. For this example I just chose an arbitrary 24 hours to also flush the cache.

Take a look at the first line of our dynamic-mapped-statement:

```

<dynamic-mapped-statement name="selectEmployeesFromSearch" result-
map="employee_result" cache-model="employee_cache">

```

Later, in our code when we access iBATIS with the SQL name "selectEmployeesFromSearch" the mapped SQL query above is performed and each row that is returned will populate the object defined in our "result-map" and this Object is added to a List. Using iBATIS, you don't have to worry about looping through a ResultSet and creating your beans. All of this tedious work is taken care of by iBATIS. Notice the result-map definition "employee\_result" maps net.reumann.Employee property names with column names:

```

<result-map name="employee_result" class="net.reumann.Employee">
    <property name="id" column="emp_id"/>
    <property name="name" column="name"/>
    <property name="deptId" column="dept_id"/>
    <property name="deptName" column="dept_name"/>

```

```

    <property name="location"    column="location_id" mapped-
statement="getLocationById"/>
</result-map>

```

Each property name in your result-map should match the name of the corresponding property in the net.reumann.Employee bean. If we look at the Employee bean you will also see that it has a property called net.reumann.Location location. Notice in our employee\_result result-map definition we populate this Location bean (which is nested inside of the Employee bean) by calling another mapped-statement: 'getLocationById'.

Looking at the Location object, you can see it is a very simple bean and in real life I wouldn't actually make a separate query to another mapped-statement to populate such a small object since it's possible the List of Employee beans being returned could be quite large, and thus each row would have to make a separate call to the DB to populate Location. However, calling other mapped-statements like this comes in EXTREMELY handy when you have to build a complex bean from a bunch of other beans. For example, maybe I want to build a "Company" bean and I might make one call to a mapped-statement to build the Employee List for this Company object and another call to a mapped-statement to build a "FinancialRecord" bean inside of Company. This might look like...

```

<result-map name="company_result" class="net.reumann.Company">
  <property name="id"          column="company_id"/>
  <property name="name"       column="name"/>
  <property name="address"    column="address"/>
  <property name="employees"  column="company_id" mapped-
statement="getEmployees"/>
  <property name="financeRecord" column="company_id" mapped-
statement="getFinacialRecord"/>
</result-map>

```

You are not required to populate a result-map. If your column names match up to the properties of your bean you could use a result-class, as is the case in the above "getLocationsById" mapping. This is usually very easy to do since you could always rename the columns you get back in your query using an AS construct (ie. first\_name AS firstName). If you want, you could also just return a HashMap as the Object and not even bother declaring a bean object to map to. With a Struts application I usually prefer to populate beans since I think it makes displaying the bean properties on JSPs cleaner than grabbing the properties from a Map ( beanName.someProperty seems easier to understand than mapName.value.keyName). More importantly I find it easier and more intuitive to look at bean fields that I'll need to use in my JSP versus having to go to the SQL file to see what the column names were called. It's great, though, to have the flexibility to just use HashMaps if you feel like it.

Most of our mapped statements will be passed either a Map of properties/values or an Object with standard get/set methods. The ## fields you see in the mapped statements (ie #name#, #id#) need to match the same properties in the bean you pass in (get methods have to match) or they must match the name of the key if passing in a Map. You'll see how these mapped-statements are called shortly.

(Instead of using Beans and Maps, you also could pass in and return a primitive wrapper type which is useful when your query only needs one parameter or you only need one column

back. Examples of all this in the iBATIS docs).

You should also read the docs to get a feel for how the dynamic-mapped statements work. In the dynamic-mapped statement example above we will be searching for Employees but some of the criteria may or may not be there from our search, so the SQL WHERE clause is built dynamically based on whether the Map we pass in has values for the different keys. (For example if deptId is null or an empty String, the SQL will be built without creating a where for the deptId).

These comments only briefly touch on the SQL map configurations. I repeat: read the iBATIS documentation

## LESSON Struts And iBATIS - LabelValueSQL

Web applications almost always rely on drop down lists (Select options). In a web app you can only pass one String value for each option selected from your list. The Struts options tag has a labelProperty and labelValue that you could use to pull out any bean properties you want to use for the label and for the value displayed. This works well, but I tend to like to have actual fields in a generic bean called "label" and "value." I then have queries populate these Lists of LabelValue beans. For example take US State codes. You could make a State bean with properties to represent the abbreviations and state display names, but since you often are just using these beans for form display purposes, I usually just reuse a generic LabelValue bean with two fields "label" and "value." In this demo app, I populate Lists of these beans in a SetUpServlet and put the Lists in application scope. In real life I probably wouldn't give the departments and locations both application scope if they could change on a regular basis. More importantly, because iBATIS will cache queries, there is not much overhead if you just query iBATIS to bring back your Lists and forego even worrying about application scoped Lists.

(Side Note: Back in the EmployeeSQL discussion I made the comment that I don't usually return Maps back from iBATIS and prefer to get back an Object or List of objects. Although, I didn't do it in this application, it probably wouldn't be a bad idea to not even use a LabelValue bean and instead just have iBATIS return you a Map that you could put into scope. Just always name the keys something like 'label' and 'value' so you never really have to worry when displaying a List "Was the label called stateName? or stateDescription?" - just name the state description column "label," and the abbreviation column "value" and you don't have to worry about it. Remember, though, you'll have to use the mapName.value construct to get the value out in your JSP tags).

Here's the LabelValueSQL this application is using:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sql-map PUBLIC "-//iBATIS.com//DTD SQL Map 1.0//EN"
"http://www.ibatis.com/dtd/sql-map.dtd">
```

```
<sql-map name="LabelValueSQL">
```

```
  <cache-model name="label_value_cache" reference-type="STRONG">
    <flush-interval hours="48"/>
  </cache-model>
```

```

<result-map name="label_value_result"    class="net.reumann.LabelValue">
  <property name="value"                column="id" />
  <property name="label"                 column="name" />
</result-map>

<mapped-statement name="selectAllDepartments" result-map="label_value_result"
cache-model="label_value_cache">
  SELECT id, name FROM department ORDER BY name
</mapped-statement>

<mapped-statement name="selectAllLocations" result-map="label_value_result" cache-
model="label_value_cache">
  SELECT id, name FROM location ORDER BY name
</mapped-statement>

</sql-map>

```

## LESSON Struts And iBATIS - Accessing iBATIS

Imagine in a Struts application we want to insert an Employee. A typical Struts application will take us this far:

An ActionForm is set up to hold the Employee information that the user needs to fill out. The ActionForm is then available in the Action class when the form is submitted.

That's pretty much where the Struts framework ends. Now let's see how easy it is to access the iBATIS framework to get this Employee data inserted into our database.

This application uses a DynaActionForm to hold the Employee data. Remember, iBATIS usually needs a Map or bean of properties in order to perform the SQL statements in our config files. It is easy enough to get the Map of parameters from our DynaActionForm with: `employeeForm.getMap()`

so in our Action inserting an Employee ends up being as simple as:

```
employeeDAOinstance.insertEmployee( employeeForm.getMap() );
```

Pretty easy huh?

\*\*\*Although this is easy, I agree with the advice I have received from others (thanks Brandon and Larry:) that it is better to pass in a Java bean versus using a Map. Since a bean will contain the correct data types of your fields, it is much safer to pass this object to iBATIS versus a Map which will only have String properties from your form. (What if you forgot to validate a Date properly and you passed a crummy String representation of the Date to iBATIS? You will end up with some ugly SQL error. This could be avoided by making sure you had your date value stored in a bean with the correct type (`java.util.Date`, `java.sql.Date`)). If you haven't looked at any of the previous lessons, remember there are several ways you can

make sure you end up with a simple bean with the correct types converted from the ActionForm String properties that were entered by the user. Wrapping this plain bean inside of an ActionForm object is one common way. Another common approach is to use the BeanUtils/PropertyUtils copyProperties() method to build your bean from the ActionForm properties.

The DAO used in this example is not complicated either..

Our EmployeeDAO is basically only responsible for making sure it calls the BaseDAO with the proper SQL string. For example, the EmployeeDAO insert method looks like:

```
public int insertEmployee(Object parameterObject) throws DaoException {
    return super.update("insertEmployee", parameterObject);
}
```

which can be called from our Action with:

```
employeeDAOinstance.insertEmployee(employeeForm.getMap());
```

If you are looking at the actual application code you'll notice that I forgot to mention the EmployeeService class. I could have just skipped using the Service class in this demo since it doesn't do much but act as a layer between your Action class and your call to the DAO. I like to use them, though, since sometimes there are things I need to do besides just a simple DAO operation which I find works well being done in a Service class. For most simple CRUD (create, retrieve, update, delete) operations you could skip using the service class if you so desire.

The work of doing the insert then gets handed off to the BaseDAO, which has three simple reusable methods:

```
public List getList(String statementName, Object parameterObject)
public Object getObject(String statementName, Object parameterObject)
public int update(String statementName, Object parameterObject)
```

The update method is responsible for doing any 'executeUpdate' JDBC calls so our insert is handed off to this method.

The update method in the BaseDAO looks like:

```
public int update(String statementName, Object parameterObject) throws DaoException {
    int result = 0;
    try {
        sqlMap.startTransaction();
        result = sqlMap.executeUpdate(statementName, parameterObject);
        sqlMap.commitTransaction();
    } catch (SQLException e) {
        try {
            sqlMap.rollbackTransaction();
        } catch (SQLException ex) {
            throw new DaoException(ex.fillInStackTrace());
        }
        throw new DaoException(e.fillInStackTrace());
    }
    return result;
}
```

```
}
```

There is a static block in the BaseDAO which sets up the sqlMap instance that the dao uses. Notice how the BaseDAO static block grabs our config file:

```
reader = Resources.getResourceAsReader("net/reumann/conf/sql-map-config.xml");
```

Once the BaseDao is built your application could reuse it and you just have to take care of creating a few simple DAOs that call the super class BaseDAO, handing it the correct SQL string to use and your parameter object and you're all set. The great thing about this framework is that it is simple, powerful, and flexible and improves the ease in which your code can be maintained.

A Special thanks to Clinton Begin for creating iBATIS and for his suggestions for this lesson.