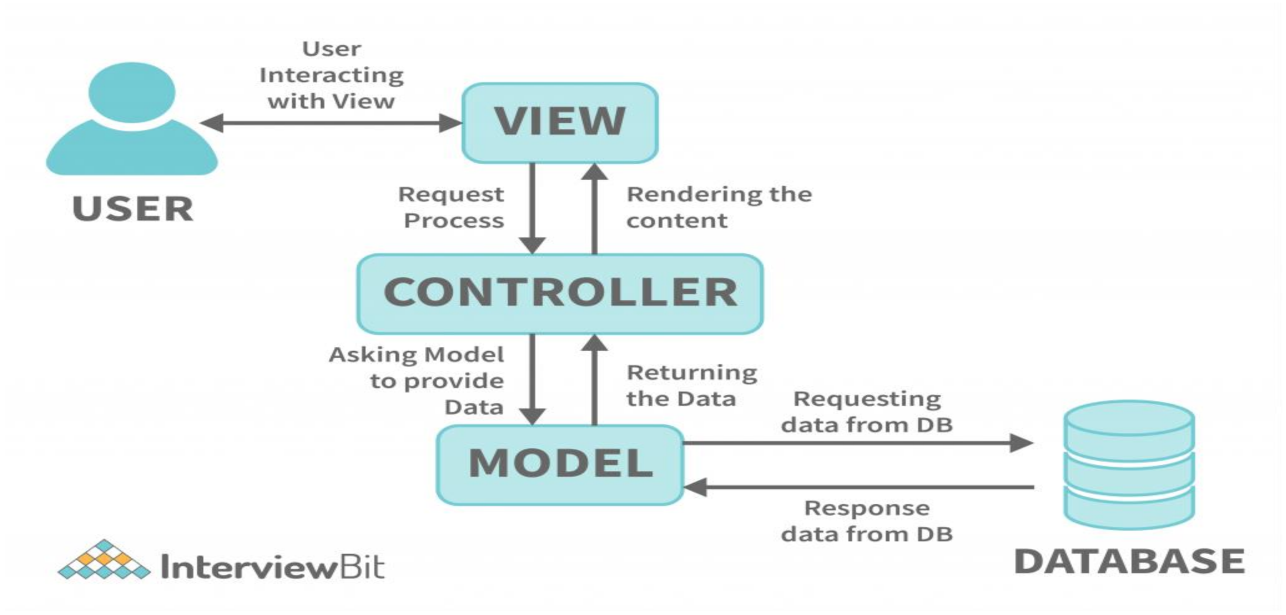
What is MVC ?

MVC stands for Model View Controller. It is a design approach to separate dataprocessing code from data presentation code.

Need For MVC :

- A single request will result in multiple substantially different looking results.
- You perform complicated data processing, but have a relatively fixed layout.

MVC Architecture



Components of MVC

Generally, we know that MVC has 3 components. Model, View, and Controller.–

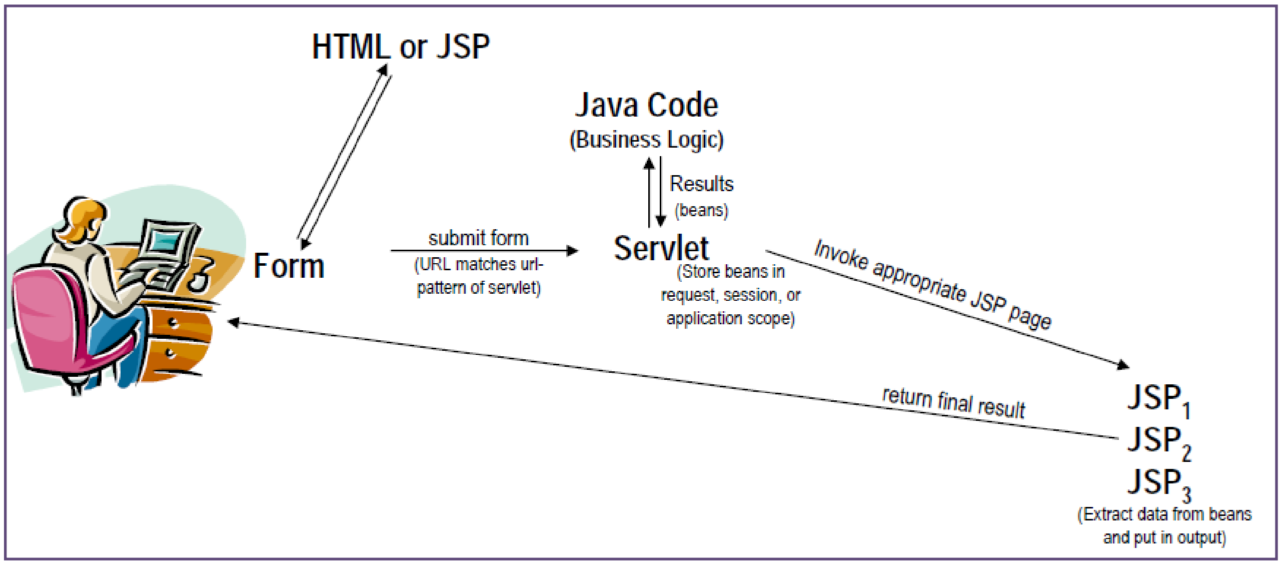
1. ***View*** *– Where the user interacts.*
2. ***Controller*** *– Process the request and send the response to view.*
3. ***Model*** *– Middleware that handles the database operations.*

View Essentially, the View represents the way that data is presented in the application. The views are created based on the data collected from the model. By requesting information from the model, the user is presented with the output presentation. Besides representing the data from charts, diagrams, and tables, the view also displays data from other sources. All user interface components, such as text boxes, drop-down menus, etc. will appear in any **customer view.**

Controller Controllers are those components of an application that handle user interaction. User input is interpreted by the controller, causing the model and view to change based on the information it receives.By communicating with a controller's associated view, a user can change the view's appearance (for example, scrolling through a document), and update the state of its associated model (for example, saving a document).

Model Essentially, a model component stores data and logic. For instance, a Controller object will retrieve customer information from a database. Data is transferred between the controller components or between business logic elements. It manipulates data and sends it back to the database, or it is used to render the same information.Additionally, it responds to views' requests and has instructions from the controller that allow it to update itself. It is also the lowest level of the pattern responsible for maintaining the data.

MVC Flow of Control


HTML or JSP
Java Code
(Business Logic)
Results
(beans)
Form
submit form
(URL matches url-pattern of servlet)
Servlet
(Store beans in request, session, or application scope)
Invoke appropriate JSP page
return final result
JSP1
JSP2
JSP3
(Extract data from beans and put in output)

The original request is handled by a servlet. The servlet invokes the business-logic and data-access code and creates beans to represent the results (that's the model). Then,the servlet decides which JSP page is appropriate to present those particular results and forwards the request there (the JSP page is the view). The servlet decides what business logic code applies and which JSP page should present the results (the servlet is the controller).

MVC Implementation

- **Implementing MVC with the built-in RequestDispatcher**
- **Frameworks as Struts**

Implementing MVC with RequestDispatcher

1- **Define beans to represent the data**

In this case, since a servlet (never a JSP page) will be creating the beans, the requirement for an empty (zero-argument) constructor is waived.

2- **Write a servlet to handle requests**

In most cases, the servlet reads request parameters. In fact, with the MVC approach the servlets do not create any output; the output is completely handled by the JSP pages.So, the servlets do not call response.setContentType, response.getWriter, or out.println.

3- **Populate the beans**

The results are placed in the beans that were defined in step 1.

ValueObject value = new ValueObject(...);

4- **Store the bean in the request, session, or servlet context**

- **Storing Data in request:**

  request.setAttribute("key", value);

- **Storing Data in session:**

  HttpSession session = request.getSession();session.setAttribute("key", value);

- **Storing Data in application:**

  ServletContext application= getServletContext();
  application.setAttribute("key", value);

5- **Forward the request to a JSP page**

The servlet determines which JSP page is appropriate to the situation and uses theforward method of RequestDispatcher to transfer control to that page.

RequestDispatcher dispatcher=request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);

6- **Extract the data from the beans**

Once the request arrives at the JSP page, the JSP page uses jsp:useBean and jsp:getProperty to extract the data.

- The servlet, not the JSP page, should create all the data objects. So, to guaranteethat the JSP page will not create objects, you should use:

  <jsp:useBean ... type="package.Class" />

  instead of

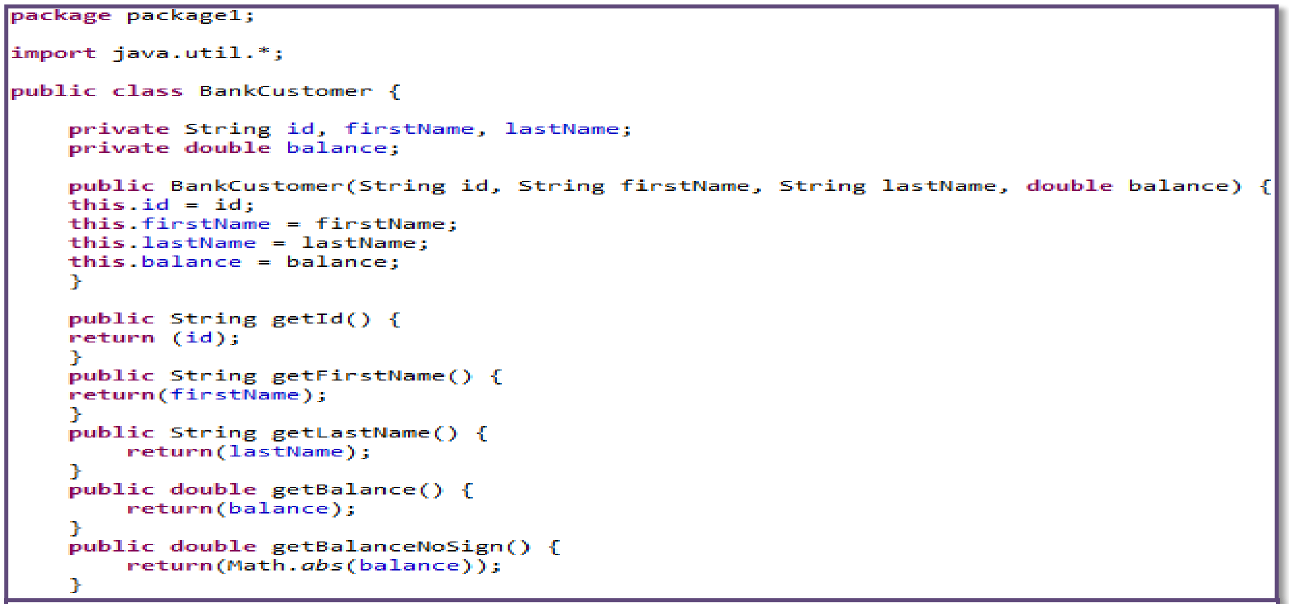
  <jsp:useBean ... class="package.Class" />.

- The JSP page should not modify the objects. So, you should use jsp:getPropertybut not jsp:setProperty.

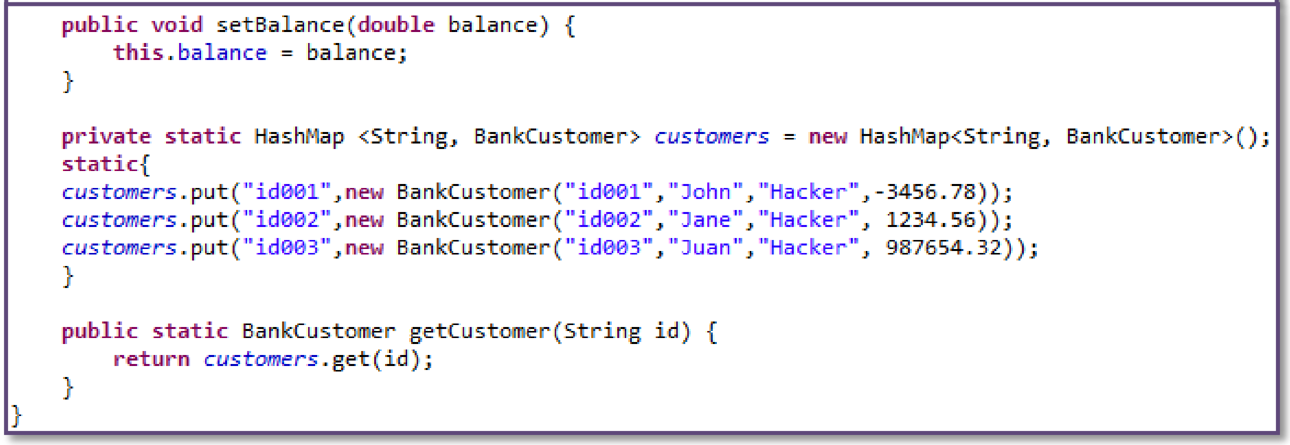
Example: Bank Account Balances

BankCustomer.java(Bean)

- **BankCustomer Bean**

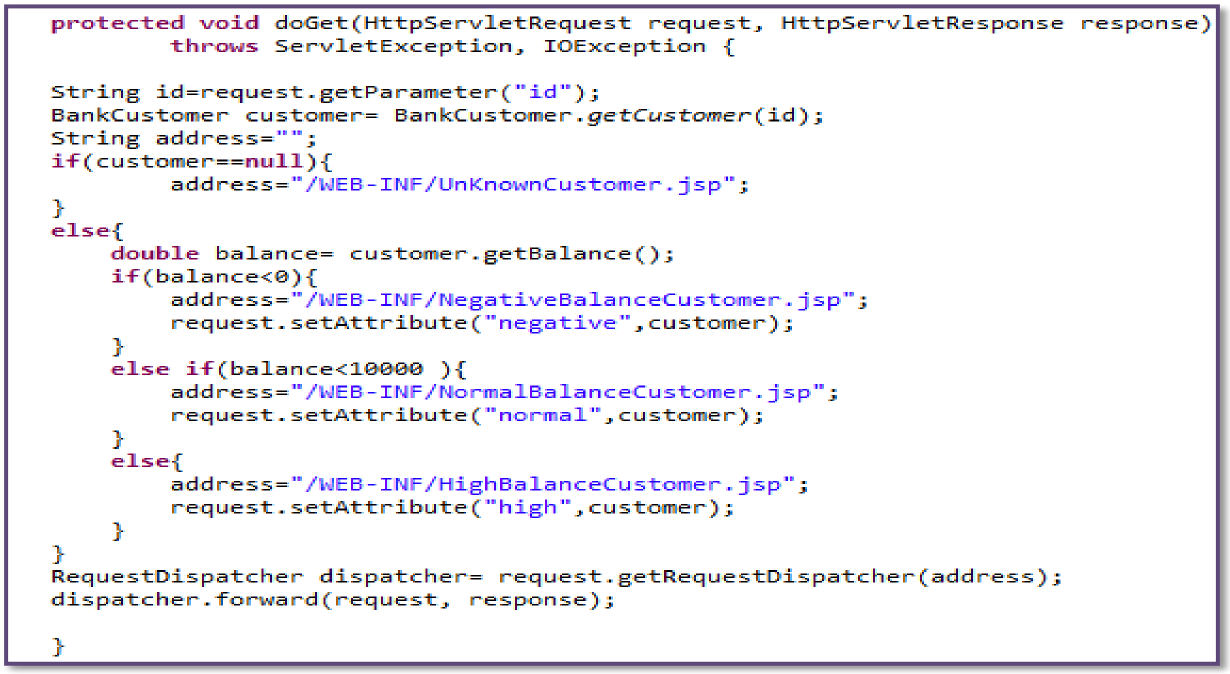


```java
package package1;

import java.util.*;

public class BankCustomer {

    private String id, firstName, lastName;
    private double balance;

    public BankCustomer(String id, String firstName, String lastName, double balance) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.balance = balance;
    }

    public String getId() {
    return (id);
    }
    public String getFirstName() {
    return(firstName);
    }
    public String getLastName() {
        return(lastName);
    }
    public double getBalance() {
        return(balance);
    }
    public double getBalanceNoSign() {
        return(Math.abs(balance));
    }
```



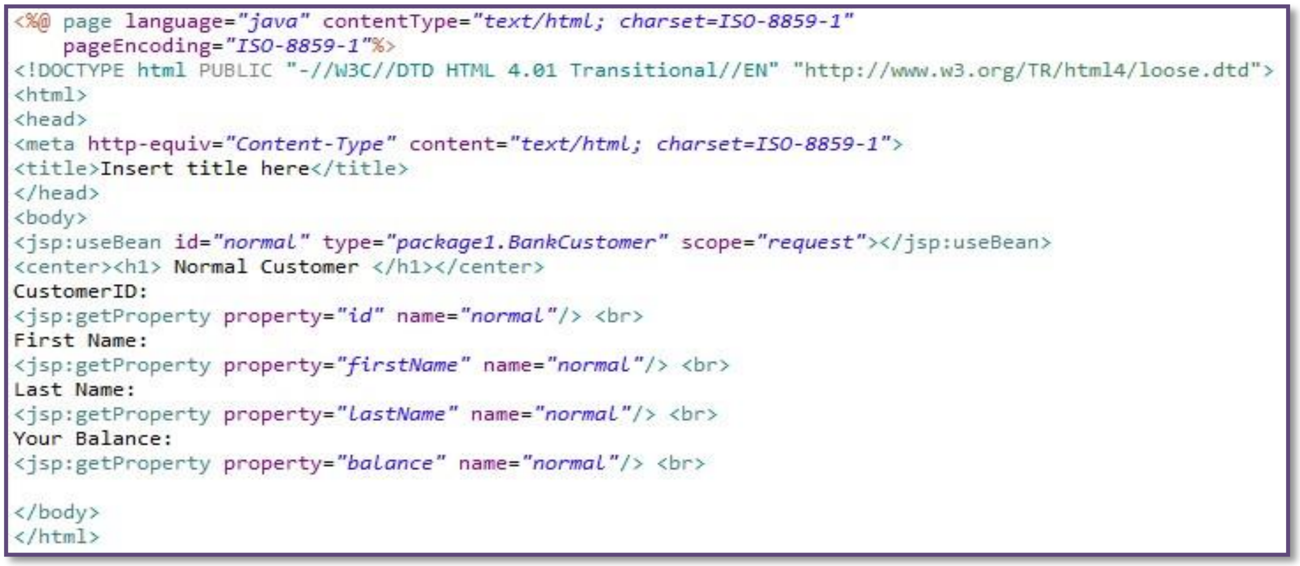
```java
    public void setBalance(double balance) {
        this.balance = balance;
    }

    private static HashMap <String, BankCustomer> customers = new HashMap<String, BankCustomer>();
    static{
    customers.put("id001",new BankCustomer("id001","John","Hacker",-3456.78));
    customers.put("id002",new BankCustomer("id002","Jane","Hacker", 1234.56));
    customers.put("id003",new BankCustomer("id003","Juan","Hacker", 987654.32));
    }

    public static BankCustomer getCustomer(String id) {
        return customers.get(id);
    }
}
```

- **showBalance.java (Servlet)**



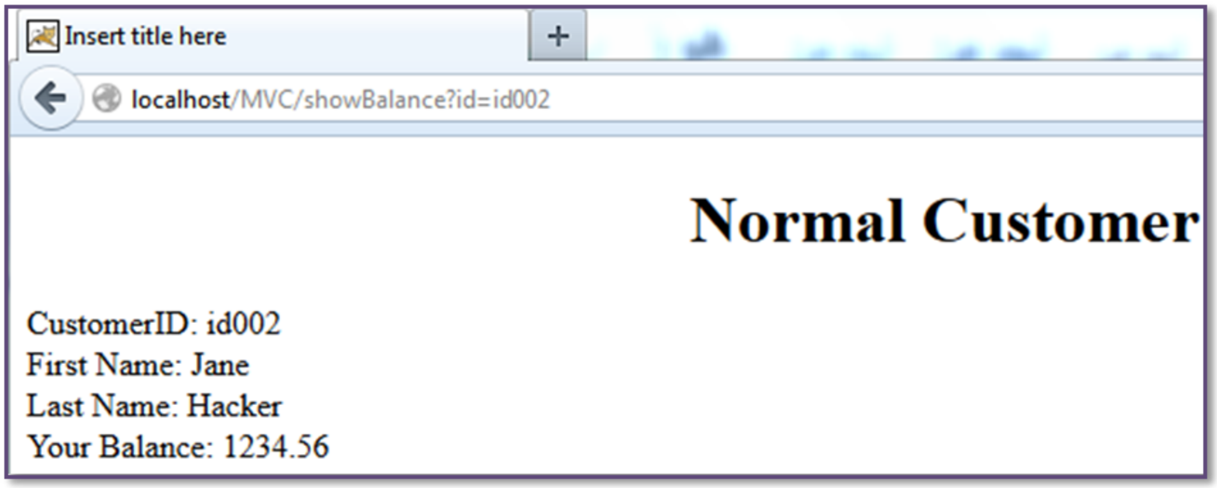
```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    String id=request.getParameter("id");
    BankCustomer customer= BankCustomer.getCustomer(id);
    String address="";
    if(customer==null){
            address="/WEB-INF/UnKnownCustomer.jsp";
    }
    else{
        double balance= customer.getBalance();
        if(balance<0){
            address="/WEB-INF/NegativeBalanceCustomer.jsp";
            request.setAttribute("negative",customer);
        }
        else if(balance<10000 ){
            address="/WEB-INF/NormalBalanceCustomer.jsp";
            request.setAttribute("normal",customer);
        }
        else{
            address="/WEB-INF/HighBalanceCustomer.jsp";
            request.setAttribute("high",customer);
        }
    }
    RequestDispatcher dispatcher= request.getRequestDispatcher(address);
    dispatcher.forward(request, response);

    }
```

NormalBalanceCustomer.jsp



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="normal" type="package1.BankCustomer" scope="request"></jsp:useBean>
<center><h1> Normal Customer </h1></center>
CustomerID:
<jsp:getProperty property="id" name="normal"/> <br>
First Name:
<jsp:getProperty property="firstName" name="normal"/> <br>
Last Name:
<jsp:getProperty property="lastName" name="normal"/> <br>
Your Balance:
<jsp:getProperty property="balance" name="normal"/> <br>

</body>
</html>
```

The Output



Normal Customer

CustomerID: id002
First Name: Jane
Last Name: Hacker
Your Balance: 1234.56

Three Data-Sharing Approaches

`1. Request-Based Sharing
2.Session-Based Sharing
3.Application-Based Sharing

1.Request-Based Data Sharing

With request-based sharing, the servlet stores the beans in the HttpServlet-Request, where they are accessible only to the destination JSP page.

Servlet

ValueObjectvalue = new ValueObject(...);

request.setAttribute("key", value);

RequestDispatcherdispatcher =request.getRequestDispatcher("/WEB-INF/SomePage.jsp");

dispatcher.forward(request, response);

JSP Page

<jsp:useBeanid="key" type="somePackage.ValueObject“ **scope="request"** />

<jsp:getPropertyname="key" property="someProperty" />

2.Session-Based Data Sharing

With session-based sharing, the servlet stores the beans in the HttpSession, where they are accessible to the same client in the destination JSP page or in other pages.

Servlet

```
ValueObjectvalue = new ValueObject(...);

HttpSessionsession = request.getSession();

session.setAttribute("key", value);

RequestDispatcherdispatcher =request.getRequestDispatcher("/WEB-INF/SomePage.jsp");

dispatcher.forward(request, response);
```

JSP Page

<jsp:useBeanid="key" type="somePackage.ValueObject“ **scope="session"** />

<jsp:getPropertyname="key" property="someProperty" />

3.Application-Based Data Sharing

With application-based sharing, the servlet stores the beans in the Servlet-Context, where they are accessible to any servlet or JSP page in the Web application.

To guarantee that the JSP page extracts the same data that the servlet inserted,youshould synchronize your code as below.

Servlet

```
synchronized(this) {

ValueObjectvalue = new ValueObject(...);

getServletContext().setAttribute("key", value);

RequestDispatcherdispatcher =

request.getRequestDispatcher("/WEB-INF/SomePage.jsp");

dispatcher.forward(request, response);

}
```

JSP Page

<jsp:useBeanid="key" type="somePackage.ValueObject“ **scope="application"** />

<jsp:getPropertyname="key" property="someProperty" />

Comparing the Three Data-Sharing Approaches
In the MVC approach, a servlet responds to the initial request. The servlet invokes code that fetches or creates the business data, places that data in beans, stores the beans, and forwards the request to a JSP page to present the results. But, *where* does the servlet store the beans?

The most common answer is, in the request object. That is the only location to which the JSP page has sole access. However, you sometimes want to keep the results around for the same client (session-based sharing) or store Web-application-wide data (application-based sharing).

Request-Based Sharing : In this example, our goal is to display a random number to the user. Each request should result in a new number, so request-based sharing is appropriate

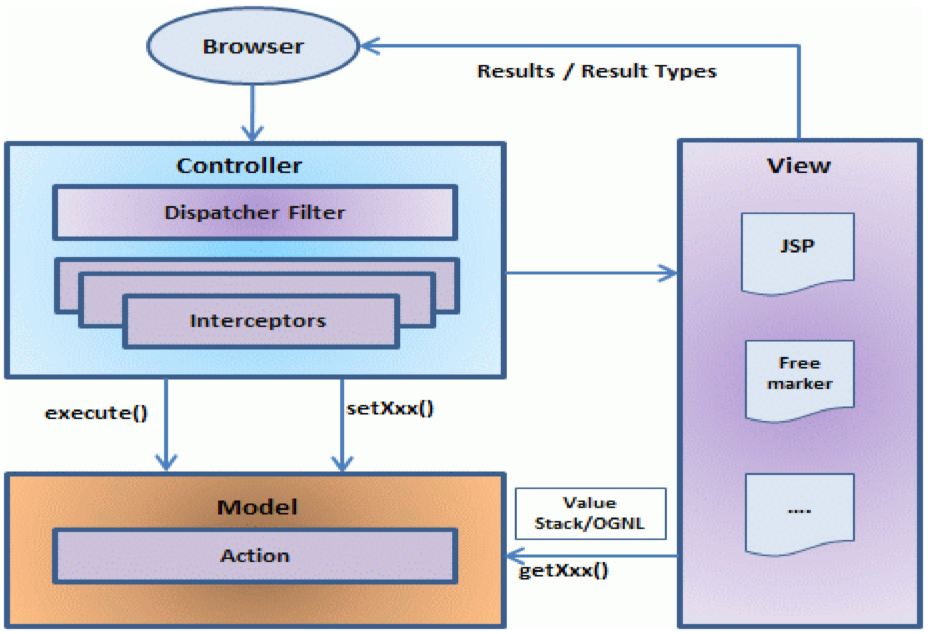
Session-Based Sharing : In this example, our goal is to display users' first and last names. If the users fail to tell us their name, we want to use whatever name they gave us previously. If the users do not explicitly specify a name and no previous name is found, a warning should be displayed. Data is stored for each client, so session-based sharing is appropriate.

Application-Based Sharing : In this example, our goal is to display a prime number of a specified length. If the user fails to tell us the desired length, we want to use whatever prime number we most recently computed for *any* user. Data is shared among multiple clients, so application-based sharing is appropriate.

Struts2

The **struts 2 framework** is used to develop **MVC-based web application**.

Struts 2 - Architecture



The above diagram depicts the **M**odel, **V**iew and **C**ontroller to the Struts2 high level architecture. The controller is implemented with a **Struts2** dispatch servlet filter as well as interceptors, this model is implemented with actions, and the view is a combination of result types and results. The value stack and OGNL provides common thread, linking and enabling integration between the other components.

Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc.

This is the architectural overview of the Struts 2 MVC pattern. We will go through each component in more detail in the subsequent chapters.

Request Life Cycle

Based on the above diagram, you can understand the work flow through user's request life cycle in **Struts 2** as follows −

- User sends a request to the server for requesting for some resource (i.e. pages).
- The Filter Dispatcher looks at the request and then determines the appropriate Action.
- Configured interceptor functionalities applies such as validation, file upload etc.
- Selected action is performed based on the requested operation.
- Again, configured interceptors are applied to do any post-processing if required.
- Finally, the result is prepared by the view and returns the result to the user.

core components of Struts 2

Struts2 is implemented with the following five core components −

- Actions
- Interceptors
- Value Stack / OGNL
- Results / Result types
- View technologies

Actions

Actions are the core of the Struts2 framework, as they are for any MVC (Model View Controller) framework. Each URL is mapped to a specific action, which provides the processing logic which is necessary to service the request from the user.

Example :Create Action Class

Optionally you can extend the **ActionSupport** class which implements six interfaces including **Action** interface.

```
public class HelloWorldAction extends ActionSupport {
  private String name;

  public String execute() throws Exception {
    if ("SECRET".equals(name)) {
      return SUCCESS;
    } else {
      return ERROR;
```

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

Interceptors

Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors −

- Providing preprocessing logic before the action is called.
- Providing postprocessing logic after the action is called.
- Catching exceptions so that alternate processing can be performed.

Many of the features provided in the **Struts2** framework are implemented using interceptors;

Struts2 Framework Interceptors

Struts 2 framework provides a good list of out-of-the-box interceptors that come preconfigured and ready to use. Few of the important interceptors are listed below −

Sr.No	Interceptor & Description
1	**alias** Allows parameters to have different name aliases across requests.
2	**checkbox** Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked.
3	**conversionError** Places error information from converting strings to parameter types into the action's field errors.
4	**createSession** Automatically creates an HTTP session if one does not already exist.
5	**debugging** Provides several different debugging screens to the developer.
6	**execAndWait** Sends the user to an intermediary waiting page while the action executes in the background.

7	**exception** Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection.
8	**fileUpload** Facilitates easy file uploading.
9	**i18n** Keeps track of the selected locale during a user's session.
10	**logger** Provides simple logging by outputting the name of the action being executed.
11	**params** Sets the request parameters on the action.
12	**prepare** This is typically used to do pre-processing work, such as setup database connections.
13	**profile** Allows simple profiling information to be logged for actions.
14	**scope** Stores and retrieves the action's state in the session or application scope.
15	**ServletConfig** Provides the action with access to various servlet-based information.
16	**timer** Provides simple profiling information in the form of how long the action takes to execute.
17	**token** Checks the action for a valid token to prevent duplicate formsubmission.
18	**validation** Provides validation support for actions

Example :Create Interceptor Class

```
import java.util.*;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class MyInterceptor extends AbstractInterceptor {

   public String intercept(ActionInvocation invocation)throws Exception {

      /* let us do some pre-processing */
      String output = "Pre-Processing";
      System.out.println(output);

      /* let us call action or next interceptor */
      String result = invocation.invoke();

      /* let us do some post-processing */
      output = "Post-Processing";
      System.out.println(output);

      return result;
   }
}
```

Results & Result Types

the **<results>** tag plays the role of a **view** in the Struts2 MVC framework. The action is responsible for executing the business logic. The next step after executing the business logic is to display the view using the **<results>** tag.

Struts comes with a number of predefined **result types** and whatever we've already seen that was the default result type **dispatcher**, which is used to dispatch to JSP pages. Struts allow you to use other markup languages for the view technology to present the results and popular choices include **Velocity, Freemaker, XSLT** and **Tiles**.

1.The Dispatcher Result Type

The **dispatcher** result type is the default type, and is used if no other result type is specified. It's used to forward to a servlet, JSP, HTML page, and so on, on the server. It uses the *RequestDispatcher.forward()* method.

We saw the "shorthand" version in our earlier examples, where we provided a JSP path as the body of the result tag.

```
<result name = "success">
   /HelloWorld.jsp
</result>
```

We can also specify the JSP file using a <param name = "location"> tag within the <result...> element as follows −

```
<result name = "success" type = "dispatcher">
   <param name = "location">
      /HelloWorld.jsp
   </param >
</result>
```

We can also supply a **parse** parameter, which is true by default. The parse parameter determines whether or not the location parameter will be parsed for OGNL expressions.

2.The FreeMaker Result Type

In this example, we are going to see how we can use **FreeMaker** as the view technology. Freemaker is a popular templating engine that is used to generate output using predefined templates. Let us now create a Freemaker template file called **hello.fm** with the following contents −

```
Hello World ${name}
```

The above file is a template where **name** is a parameter which will be passed from outside using the defined action. You will keep this file in your CLASSPATH.

Next, let us modify the **struts.xml** to specify the result as follows −

```
<?xml version = "1.0" Encoding = "UTF-8"?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
   <constant name = "struts.devMode" value = "true" />
   <package name = "helloworld" extends = "struts-default">

      <action name = "hello"
         class = "com.tutorialspoint.struts2.HelloWorldAction"
         method = "execute">
         <result name = "success" type = "freemarker">
            <param name = "location">/hello.fm</param>
         </result>
      </action>

   </package>

</struts>
```

3.The Redirect Result Type

The **redirect** result type calls the standard *response.sendRedirect()* method, causing the browser to create a new request to the given location.

We can provide the location either in the body of the <result...> element or as a <param name = "location"> element. Redirect also supports the **parse** parameter. Here's an example configured using XML −

```
<action name = "hello"
   class = "com.tutorialspoint.struts2.HelloWorldAction"
   method = "execute">
   <result name = "success" type = "redirect">
      <param name = "location">
         /NewWorld.jsp
      </param >
   </result>
</action>
```

Value Stack/OGNL

The Value Stack

The value stack is a set of several objects which keeps the following objects in the provided

Sr.No	Objects & Description
1	**Temporary Objects** There are various temporary objects which are created during execution of a page. For example the current iteration value for a collection being looped over in a JSP tag.
2	**The Model Object** If you are using model objects in your struts application, the current model object is placed before the action on the value stack.
3	**The Action Object** This will be the current action object which is being executed.
4	**Named Objects** These objects include #application, #session, #request, #attr and #parameters and refer to the corresponding servlet scopes.

Once you have a ValueStack object, you can use the following methods to manipulate that object −

Sr.No	ValueStack Methods & Description
1	**Object findValue(String expr)** Find a value by evaluating the given expression against the stack in the default search order.
2	**CompoundRoot getRoot()** Get the CompoundRoot which holds the objects pushed onto the stack.

3	**Object peek()** Get the object on the top of the stack without changing the stack.
4	**Object pop()** Get the object on the top of the stack and remove it from the stack.
5	**void push(Object o)** Put this object onto the top of the stack.
6	**void set(String key, Object o)** Sets an object on the stack with the given key so it is retrievable by findValue(key,...)
7	**void setDefaultType(Class defaultType)** Sets the default type to convert to if no type is provided when getting a value.
8	**void setValue(String expr, Object value)** Attempts to set a property on a bean in the stack with the given expression using the default search order.
9	**int size()** Get the number of objects in the stack.

The OGNL

The **Object-Graph Navigation Language** (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack. OGNL also helps in data transfer and type conversion.

The OGNL is very similar to the JSP Expression Language. OGNL is based on the idea of having a root or default object within the context. The properties of the default or root object can be referenced using the markup notation, which is the pound symbol.

As mentioned earlier, OGNL is based on a context and Struts builds an ActionContext map for use with OGNL. The ActionContext map consists of the following −

- **Application** − Application scoped variables
- **Session** − Session scoped variables
- **Root / value stack** − All your action variables are stored here
- **Request** − Request scoped variables

- **Parameters** − Request parameters
- **Atributes** − The attributes stored in page, request, session and application scope

Struts 2 - File Uploads

The Struts 2 framework provides built-in support for processing file upload using "Form-based File Upload in HTML". When a file is uploaded, it will typically be stored in a temporary directory and they should be processed or moved by your Action class to a permanent directory to ensure the data is not lost.

File uploading in Struts is possible through a pre-defined interceptor called **FileUpload** interceptor which is available through the org.apache.struts2.interceptor.FileUploadInterceptor class and included as part of the **defaultStack**. Still you can use that in your struts.xml to set various parameters as we will see below.

Create View Files

Let us start with creating our view which will be required to browse and upload a selected file. So let us create an **index.jsp** with plain HTML upload form that allows the user to upload a file −

```
<%@ page language = "java" contentType = "text/html; charset = ISO-8859-1"
   pageEncoding = "ISO-8859-1"%>
<%@ taglib prefix = "s" uri = "/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
   <head>
      <title>File Upload</title>
   </head>

   <body>
      <form action = "upload" method = "post" enctype = "multipart/form-data">
         <label for = "myFile">Upload your file</label>
         <input type = "file" name = "myFile" />
         <input type = "submit" value = "Upload"/>
      </form>
   </body>
</html>
```

There is couple of points worth noting in the above example. First, the form's enctype is set to **multipart/form-data**. This should be set so that file uploads are handled successfully by the file upload interceptor. The next point noting is the form's action method **upload** and the name of the file upload field - which is **myFile**. We need this information to create the action method and the struts configuration.

Next, let us create a simple jsp file **success.jsp** to display the outcome of our file upload in case it becomes success.

```
<%@ page contentType = "text/html; charset = UTF-8" %>
<%@ taglib prefix = "s" uri = "/struts-tags" %>
```

```
<html>
   <head>
      <title>File Upload Success</title>
   </head>

   <body>
      You have successfully uploaded <s:property value = "myFileFileName"/>
   </body>
</html>
```

Following will be the result file **error.jsp** in case there is some error in uploading the file −

```
<%@ page contentType = "text/html; charset = UTF-8" %>
<%@ taglib prefix = "s" uri = "/struts-tags" %>

<html>
   <head>
      <title>File Upload Error</title>
   </head>

   <body>
      There has been an error in uploading the file.
   </body>
</html>
```

Create Action Class

Next, let us create a Java class called **uploadFile.java** which will take care of uploading file and storing that file at a secure location −

```
package com.tutorialspoint.struts2;

import java.io.File;
import org.apache.commons.io.FileUtils;
import java.io.IOException;

import com.opensymphony.xwork2.ActionSupport;

public class uploadFile extends ActionSupport {
   private File myFile;
   private String myFileContentType;
   private String myFileFileName;
   private String destPath;

   public String execute() {
      /* Copy file to a safe location */
      destPath = "C:/apache-tomcat-6.0.33/work/";

      try {
         System.out.println("Src File name: " + myFile);
         System.out.println("Dst File name: " + myFileFileName);

         File destFile  = new File(destPath, myFileFileName);
         FileUtils.copyFile(myFile, destFile);

      } catch(IOException e) {
         e.printStackTrace();
         return ERROR;
      }

      return SUCCESS;
   }

   public File getMyFile() {
      return myFile;
   }

   public void setMyFile(File myFile) {
      this.myFile = myFile;
   }

   public String getMyFileContentType() {
      return myFileContentType;
   }

   public void setMyFileContentType(String myFileContentType) {
      this.myFileContentType = myFileContentType;
   }

   public String getMyFileFileName() {
      return myFileFileName;
   }

   public void setMyFileFileName(String myFileFileName) {
      this.myFileFileName = myFileFileName;
   }
}
```

The **uploadFile.java** is a very simple class. The important thing to note is that the FileUpload interceptor along with the Parameters Interceptor does all the heavy lifting for us.

The FileUpload interceptor makes three parameters available for you by default. They are named in the following pattern −

- **[your file name parameter]** − This is the actual file that the user has uploaded. In this example it will be "myFile"
- **[your file name parameter]ContentType** − This is the content type of the file that was uploaded. In this example it will be "myFileContentType"

- **[your file name parameter]FileName** − This is the name of the file that was uploaded. In this example it will be "myFileFileName"

The three parameters are available for us, thanks to the Struts Interceptors. All we have to do is to create three parameters with the correct names in our Action class and automatically these variables are auto wired for us. So, in the above example, we have three parameters and an action method that simply returns "success" if everything goes fine otherwise it returns "error".

Configuration Files

Following are the Struts2 configuration properties that control file uploading process −

Sr.No	Properties & Description
1	**struts.multipart.maxSize** The maximum size (in bytes) of a file to be accepted as a file upload. Default is 250M.
2	**struts.multipart.parser** The library used to upload the multipart form. By default is **jakarta**
3	**struts.multipart.saveDir** The location to store the temporary file. By default is javax.servlet.context.tempdir.

Struts 2 - Database Access
Struts is a MVC framework and not a database framework

Hibernate

Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

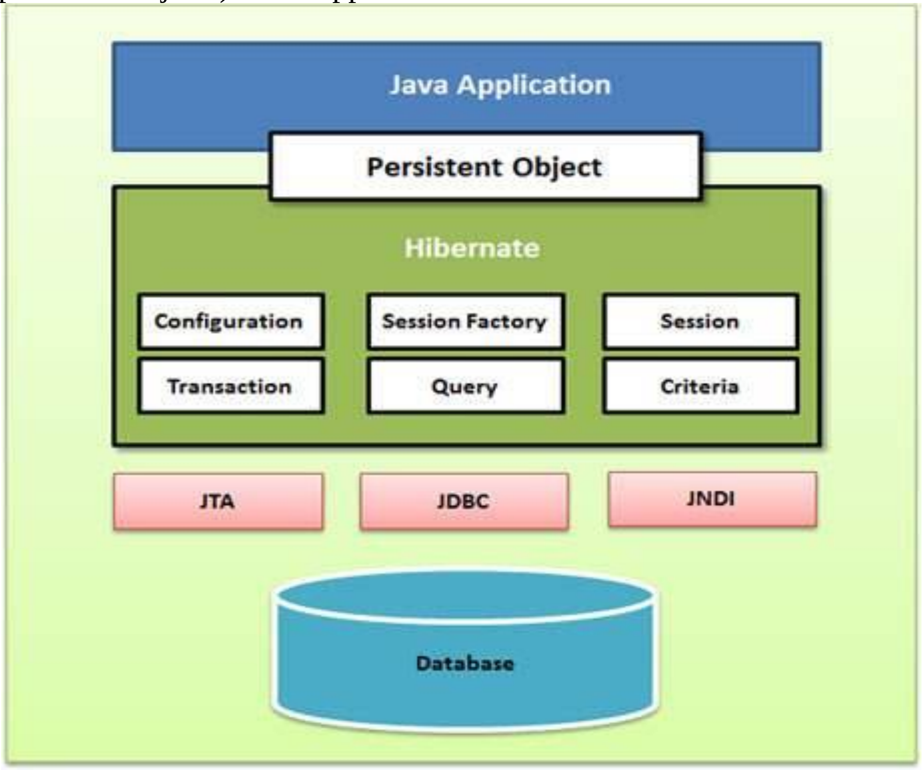
Hibernate Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.

- Hibernate does not require an application server to operate.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

Hibernate - Architecture

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.



gives brief description of each of the class objects involved in Hibernate Application Architecture.

Configuration Object

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

The Configuration object provides two keys components −

- **Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
- **Class Mapping Setup** − This component creates the connection between the Java classes and database tables.

SessionFactory Object

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object

Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

JPA/Hibernate integration

Hibernate is a high-performance Object/Relational persistence and query service which is licensed under the open source GNU Lesser General Public License (LGPL) and is free to download. In this chapter. we are going to learn how to achieve Struts 2 integration with Hibernate.

Action Class

Following file **AddStudentAction.java** defines our action class. We have two action methods here - execute() and listStudents(). The execute() method is used to add the new student record. We use the dao's save() method to achieve this.

The other method, listStudents() is used to list the students. We use the dao's list method to get the list of all students.

```
package com.tutorialspoint.struts2;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.tutorialspoint.hibernate.Student;
import com.tutorialspoint.hibernate.StudentDAO;

public class AddStudentAction extends ActionSupport implements ModelDriven<Student> {

   Student student  = new Student();
   List<Student> students = new ArrayList<Student>();
   StudentDAO dao = new StudentDAO();
   @Override

  public Student getModel() {
      return student;
   }

   public String execute() {
      dao.addStudent(student);
      return "success";
   }

   public String listStudents() {
      students = dao.getStudents();
      return "success";
   }

   public Student getStudent() {
      return student;
   }

   public void setStudent(Student student) {
      this.student = student;
   }

   public List<Student> getStudents() {
      return students;
   }

   public void setStudents(List<Student> students) {
      this.students = students;
   }

}
```

You will notice that we are implementing the ModelDriven interface. This is used when your action class is dealing with a concrete model class (such as Student) as opposed to individual properties (such as firstName, lastName). The ModelAware interface requires you to implement a method to return the model. In our case we are returning the "student" object.

Create View Files

Let us now create the **student.jsp** view file with the following content −

```
<%@ page contentType = "text/html; charset = UTF-8"%>
<%@ taglib prefix = "s" uri = "/struts-tags"%>

<html>
   <head>
      <title>Hello World</title>
      <s:head />
   </head>

   <body>
      <s:form action = "addStudent">
         <s:textfield name = "firstName" label = "First Name"/>
         <s:textfield name = "lastName" label = "Last Name"/>
         <s:textfield name = "marks" label = "Marks"/>
         <s:submit/>
         <hr/>

         <table>
            <tr>
               <td>First Name</td>
               <td>Last Name</td>
               <td>Marks</td>
            </tr>

            <s:iterator value = "students">
               <tr>
                  <td><s:property value = "firstName"/></td>
                  <td><s:property value = "lastName"/></td>
                  <td><s:property value = "marks"/></td>
                 </tr>
            </s:iterator>
         </table>
      </s:form>
   </body>
</html>
```

The student.jsp is pretty straightforward. In the top section, we have a form that submits to "addStudent.action". It takes in firstName, lastName and marks. Because the addStudent action is tied to the ModelAware "AddSudentAction", automatically a student bean will be created with the values for firstName, lastName and marks auto populated.

At the bottom section, we go through the students list (see AddStudentAction.java). We iterate through the list and display the values for first name, last name and marks in a table.

Struts Configuration

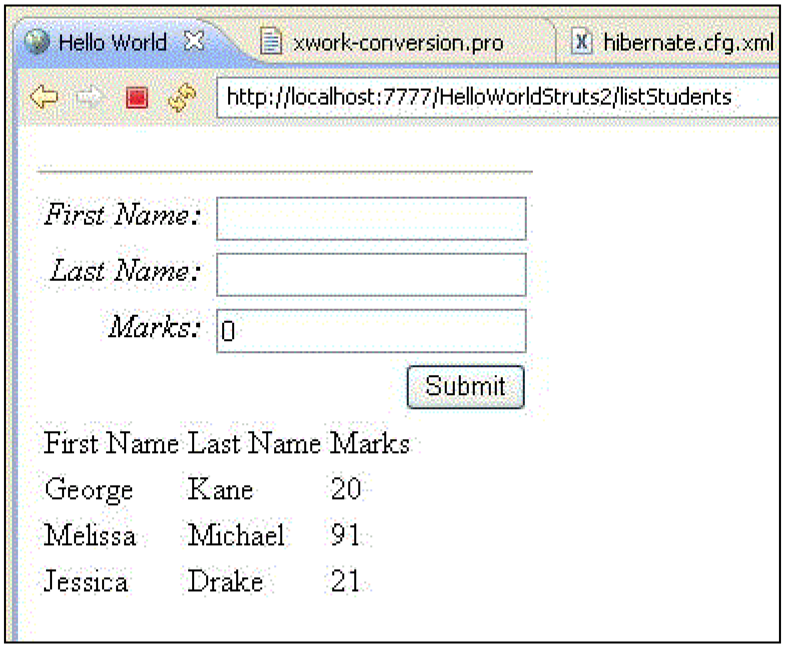
Let us put it all together using **struts.xml** −

```
<?xml version = "1.0" Encoding = "UTF-8"?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name = "struts.devMode" value = "true" />
  <package name = "myhibernate" extends = "hibernate-default">

    <action name = "addStudent" method = "execute"
      class = "com.tutorialspoint.struts2.AddStudentAction">
      <result name = "success" type = "redirect">
        listStudents
      </result>
    </action>

    <action name = "listStudents" method = "listStudents"
      class = "com.tutorialspoint.struts2.AddStudentAction">
      <result name = "success">/students.jsp</result>
    </action>

  </package>
</struts>
```

The important thing to notice here is that our package "myhibernate" extends the struts2 default package called "hibernate-default". We then declare two actions - addStudent and listStudents. addStudent calls the execute() on the AddStudentAction class and then upon successs, it calls the listStudents action method.

The listStudent action method calls the listStudents() on the AddStudentAction class and uses the student.jsp as the view.

Now, right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL **http://localhost:8080/HelloWorldStruts2/student.jsp**. This will produce the following screen −



First Name:

Last Name:

Marks: 0

Submit

First Name	Last Name	Marks
George	Kane	20
Melissa	Michael	91
Jessica	Drake	21