JavaBeans Example

Consider a car class with a few characteristics:

```
package com.myCar;

public class CarsBean implements java.io.Serializable {
   private String carName = null;
   private String brandName = null;
   private int price = 0;

   public CarsBean() {
   }
   public String getCarName(){
      return carName;
   }
   public String getBrandName(){
      return brandName;
   }
   public int getPrice(){
      return price;
   }
   public void setCarName(String carName){
      this.carName = carName;
   }
   public void setBrandName(String brandName){
      this.brandName = brandName;
   }
   public void setPrice(Integer price){
      this.price = price;
   }
}
```

How do we access JavaBeans in JSP?

The useBean action creates a JavaBean object that may be used in a JSP. The bean becomes a scripting variable once it is declared, and it may be accessed by both scripting elements and other custom tags in the JSP. The useBean tag has the following full syntax:

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Depending on our needs, the scope attribute can be set to a page, request, session, or application. The id attribute's value can be anything as long as it's a distinct name among other useBean declarations in the same JSP.

The useBean action is demonstrated in the example below:

```
<html>
   <head>
      <title>JavaBeans Sample Example</title>
   </head>

   <body>
      <jsp:useBean id = "date" class = "java.util.Date" />
      <p>The date/time is <%= date %>
   </body>
</html>
```

We will get the following outcome:
The date/time is Mon Jan 30 09:20:44 GST 2022

Getting to the Properties of JavaBeans

We can utilise the <jsp:getProperty/> action to access the get methods and the <jsp:setProperty/> action to access the set methods in addition to the <jsp:useBean...> action. Here is the complete syntax:

```
<jsp:useBean id="idHere" class="beansClass" scope="scopeOfBeans">
   <jsp:setProperty name="beansId" property="nameOfTheProperty" value="valueHere" />
   <jsp:getProperty name="beansId" property="nameOfTheProperty" />
   ...........
</jsp:useBean>
```

The id of a JavaBean previously introduced to the JSP by the useBean operation is referenced by the name attribute. The name of the get or set methods that should be used is specified in the property attribute.
The following example demonstrates how to use the above syntax to obtain data:

```
<html>
<head>
   <title>Example of getting and setting properties</title>
</head>

<body>
   <jsp:useBean id="cars" class="com.myCar.CarsBean">
      <jsp:setProperty name="cars" property="carName" value="Swift" />
      <jsp:setProperty name="cars" property="brandName" value="Maruti Suzuki" />
      <jsp:setProperty name="cars" property="price" value="700000" />
   </jsp:useBean>

   <p>Car Name:
      <jsp:getProperty name="cars" property="CarName" />
   </p>

   <p>Brand Name:
      <jsp:getProperty name="cars" property="brandName" />
   </p>

   <p>Price:
      <jsp:getProperty name="cars" property="price" />
   </p>

</body>

</html>
```

Let's add CarsBean.class to the CLASSPATH variable. The JSP mentioned above can be accessed. The following outcome will be shown:
Car Name: Swift
Brand Name: Maruti Suzuki
Price: 700000

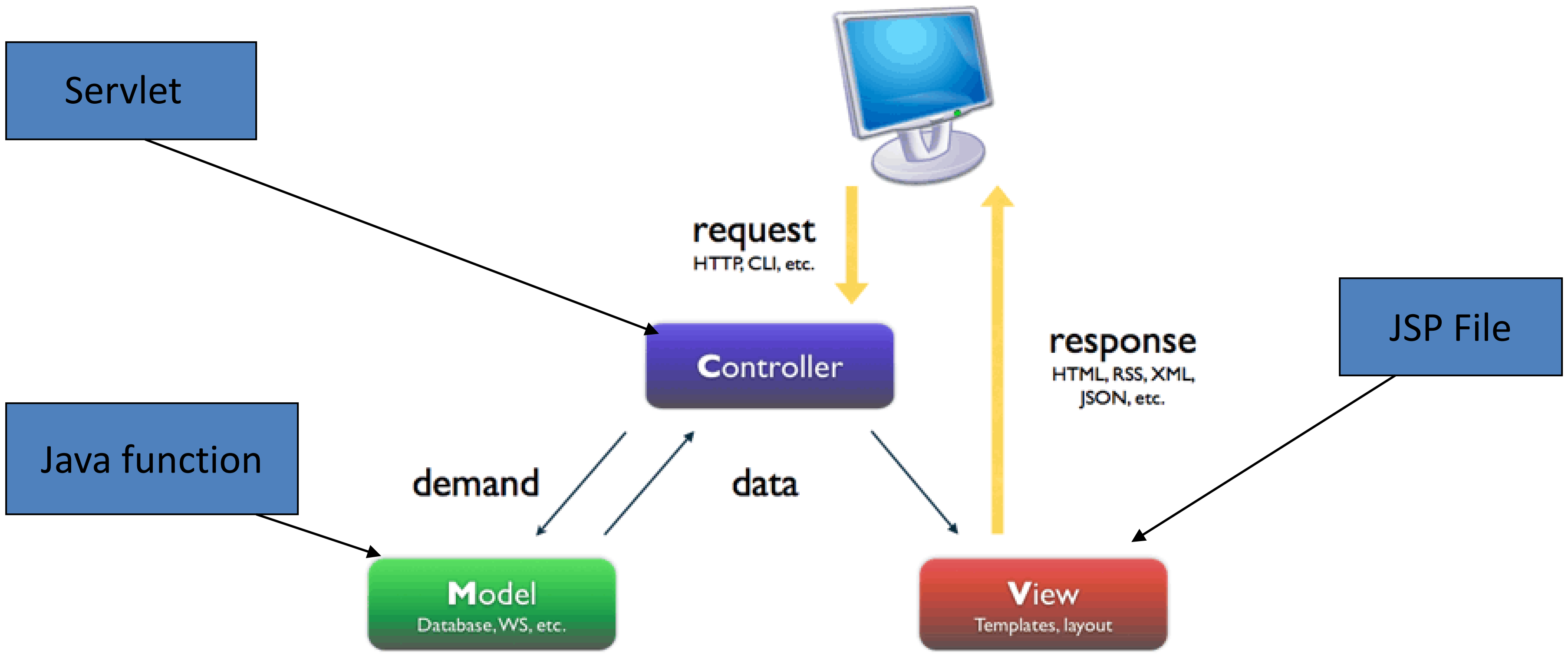
What is Struts?

- is a framework to develop web application easily.
- makes easier to develop web application and maintain them.
- is an open source framework which makes building web applications easier, based on Java Servlets and JSP technologies.
- The Struts framework was created by Craig R. McClanahan and was donated to the Apache software foundation in 2000. Since then it is a open source software.

Why struts? What's wrong with jsp/servlet coding?

- **Using only Servlets** – difficult to output a html and needs lot of out.printlns – hard to read and clumsy
- **Using only JSP** – added scriptlets and implicit objects into jsp - awkward to see java inside html– hard to read and maintain – useful if very small application
- **Using JSP+ Java beans** – Code inside bean and jsp to display . Good choice for small applications. But what if there is need of multiple type of views? making request to a general servlet, which outputs data according to the requirements, for same url request, will be good choice –Model 2 architecture evolved.
- **Using JSP+Servlets+JavaBeans** →Model 2 architecture
  Request made to servlet, servlet does business calculation using simple java POJO gets the result. Also decides the view and give back the response using the view to the client.
  servlet – Controller, Business calculation POJO – Model and JSP - View
  Uses : the business logic is separated from JSPs and JSP gets displayed depending upon the result of model (the business function). →similar behavior like all applications above, but the code is more structured now. Changing business logic will not affect view and vice versa.

Stucture of JSP + Servlets + JavaBeans : Model 2 architecture



What is a Web framework?

- Web framework is a basic readymade underlying structure, where you have to just add components related to your business.
  - For example, if you take struts, it comes with all jars you might need to develop basic request response cycle, and with basic configuration. It provides the controller servlet or filter which will read your request and convert it to integer or float etc according to your business requirements. If there is any exception while converting, you don't have to deal with that. Framework deals with the problem and displays you exact message.
  - After all conversion, the framework automatically populate all your data needed from form to the java object.
  - You don't have to write much code to validate all your form data. Frame work provides some basic automatic validations.
  - After you write business logic, you don't have to write code to dispatch request to another page. Etc
- It forces the team to implement their code in a standard way. (helps debugging, fewer bugs etc).
  - For Example, in struts immediate backend logic should be in action classes's method. Action class functions intern can call other components to finish business logic.
- Framework might also help you develop complex User Interface easily like iterating tables, the menu etc (provide some tag libraries)
- Using frameworks like struts 2.0, one need not have to have deep knowledge of Http protocol and its request and responses interfaces to write business logic

Why Application Frameworks?

- Most Model 2 architecture based web applications share a common set of functionality. For example, they all do receive and dispatch HTTP requests, invoking model methods, selecting and assembling views.
- If everybody is doing the same thing over and over every time - Spending less time on business logic.
- Its good idea to have a common framework that support these set of functionalities
- only thing developer have to do is basically using or extending the frameworks using common interface and classes and can concentrate on business logic.
- Provides rich set of features

When to use what?

- If your application is very small and have only few pages, implementing Model 2 pattern by coding servlets / JSP / and java beans would be easier and simpler.
- If Application is large have got complicated flow and business logic, need to be maintained for long terms, using frameworks would save time and resource.
- Using frameworks like struts make its easy to maintain and develop and application will be more structured.

Why struts?

- Free to develop & deploy –open source
- Stable & Mature
- Many supported third-party tools
- Feature-rich
- Flexible & Extendable
- Large User Community, Expert Developers and Committers
- Rich tag library (html, bean tags etc)
- Easy to test and debug

How does struts make code simpler?
A Sample Jsp / servlet code:

- your application might have to do following in you beans or in jsp to get a value of user input in double:

Jsp file: <input name="txtAmount"> </input>

In Bean or Jsp file:

```
String strAmount = request.getParameter("txtAmount");
double amount = 0.0;
try{
   double amount = Double.parseDouble(strAmount );
}catch(Exception e){
  // got error so return back.
// Big code for dispatching – also used in several places
  // return back to same page - hard coded
}

bean.setAmout(amount);
boolean flgResult = ejbObject.processAmount(amount);;
if(flgResult){
// success
// dispatch request to same or different page - hard coded
}else{
 // dispatch request to same or different page - hard coded
}
```

Using web framework like struts 2.0 it will look simpler

Jsp file:

```
<s:textfield label=“Amount" name=“amount" value="%{amount}" />
```

In action file you must have simple getter and setter:

```
double amount;
public double getAmount(){ return amount;}
public void setAmount(double amount){this.amount = amount;}
```

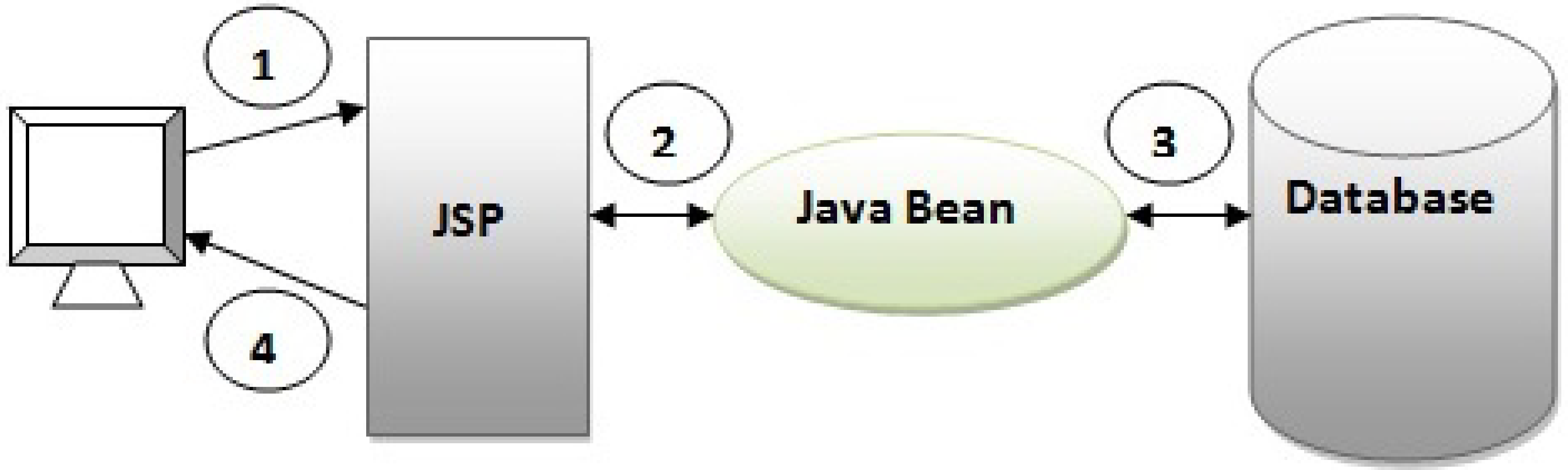
That’s it. You can directly use the amount in action method without get extra code:

```
public String execute() throws Exception{
  // use amount directly
return “success”;
}
```

Also there is no need of extra code for forwarding request.Action method is just returning a string “success”

Struts Framework Features

- Model 2 -MVC Implementation
- Internationalization(I18N) Support
- Rich JSP Tag Libraries
- Annotation and XML configuration options
- POJO-based actions that are easy to test
- Based on JSP, Servlet, XML, and Java
- Less xml configuration
- Easy to test and debug with new features
- Supports Java’s Write Once, Run Anywhere Philosophy
- Supports different model implementations (JavaBeans, EJB, etc.)
- Supports different presentation implementations(JSP, XML/XSLT, etc)



Model 1 Architecture

- *Browser sends request for the JSP page*
- *JSP accesses Java Bean and invokes business logic*
- *Java Bean connects to the database and get/save data*
- *Response is sent to the browser which is generated by JSP*

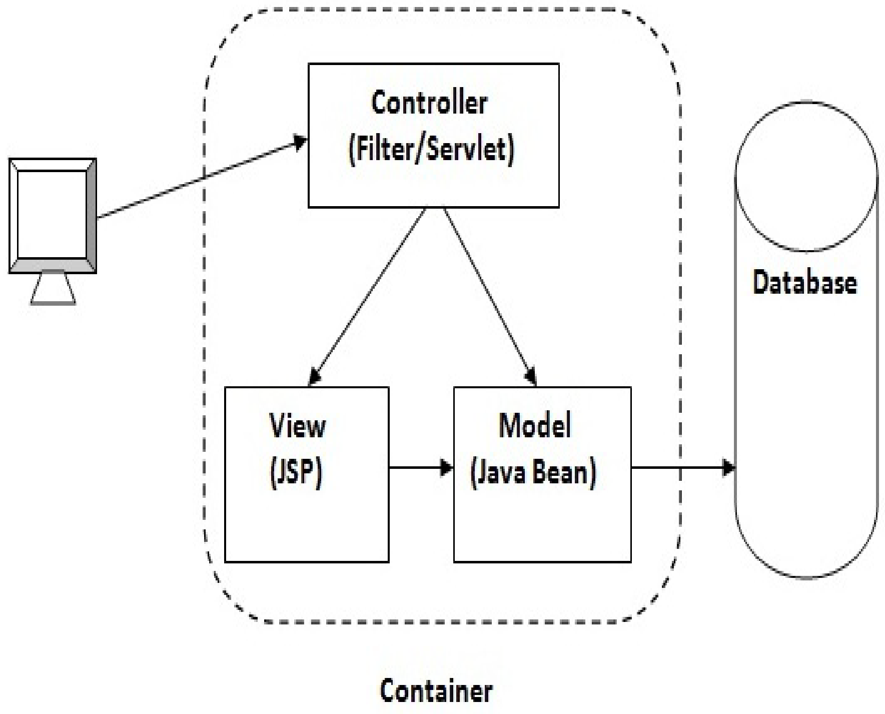
Advantage & Disadvantage of Model 1 Architecture

- Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

- **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- **Hard to extend** It is better for small applications but not for large applications.

Model 2



Advantage of Model 2 (MVC)

- **Navigation control is centralized** Now only controller contains the logic to determine the next page.
- **Easy to maintain**
- **Easy to extend**
- **Easy to test**
- **Better separation of concerns**
- Disadvantage of Model 2 (MVC)
- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

What are the pieces of struts?

- The filter dispatcher, by which all the requests to the applications gets filtered. - Controller
- The interceptors which called after filters, before action methods, apply common functionalities like validation, conversion etc
- Action classes with execute methods, usually storing and/or retrieving information from a database the view,i.e the jsp files
- Result will be figured out and the jsp file will be rendered.

Struts 2 Action

- In struts 2, action class is **POJO** (Plain Old Java Object).
- POJO means you are not forced to implement any interface or extend any class.
- Generally, **execute** method should be specified that represents the business logic. The simple action class may look like:

Welcome.java

```
package com.javatpoint;
public class Welcome {
public String execute(){
   return "success";
}
}
```

- Action Interface
- A convenient approach is to implement the **com.opensymphony.xwork2.Action** interfa ce that defines 5 constants and one execute method.
- 5 Constants of Action Interface
- Action interface provides 5 constants that can be returned form the action class. They are:
- **SUCCESS** indicates that action execution is successful and a success result should be shown to the user.
- **ERROR** indicates that action execution is failed and a error result should be shown to the user.
- **LOGIN** indicates that user is not logged-in and a login result should be shown to the user.
- **INPUT** indicates that validation is failed and a input result should be shown to the user again.
- **NONE** indicates that action execution is successful but no result should be shown to the user.

Actions

Welcome.java

```
package com.javatpoint;
import com.opensymphony.xwork2.Action;
public class Welcome implements Action{
public String execute(){
   return SUCCESS;
}
}
```

- **ActionSupport class**
- It is a convenient class that implements many interfaces such as Action, Validateable, ValidationAware, TextProvider, LocaleProvider and Serializable . So it is mostly used instead of Action.
-

Struts 2 Configuration File

- The struts application contains two main configuration files
- **struts.xml** file and **struts.properties** file.

- <?xml version="1.0" encoding="UTF-8" ?>
- <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
- Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
- <struts>
- <**package** name="default" **extends**="struts-default">
-
- <action name="product" **class**="com.stud.Product">
- <result name="success">welcome.jsp</result>
- </action>
-
- </**package**>
- </struts>

Result Types in Struts2

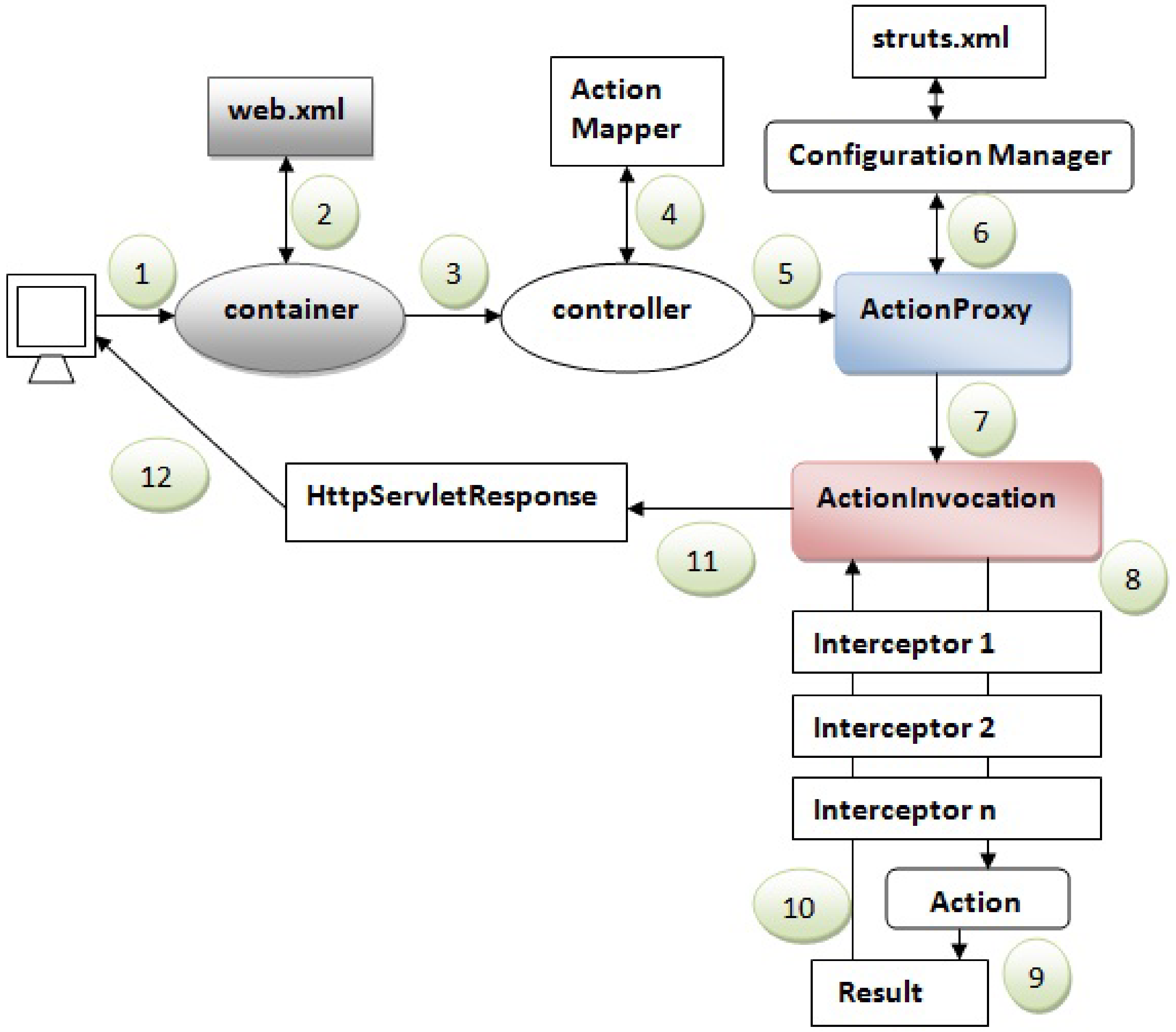
- **<results>** tag plays the role of a **view** in the Struts2 MVC framework.
- The action is responsible for executing the business logic. The next step after executing the business logic is to display the view using the **<results>** tag.
- there are three possible outcomes Successful Login, Unsuccessful Login - Incorrect username or password

Result Types in Struts2

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

Struts2 architecture flow



web.xml

- The web.xml configuration file is a J2EE configuration file that determines how elements of the HTTP request are processed by the servlet container.
- It is not strictly a Struts2 configuration file, but it is a file that needs to be configured for Struts2 to work.
- This file provides an entry point for any web application.
- The entry point of Struts2 application will be a filter defined in deployment descriptor (web.xml).
- Hence we will define an entry of *FilterDispatcher* class in web.xml.
- The web.xml file needs to be created under the folder **WebContent/WEB-INF**.

This is the first configuration file you will need to configure if you are starting without the aid of a template or tool that generates it (such as Eclipse or Maven2).

Following is the content of web.xml file which we used in our last example.

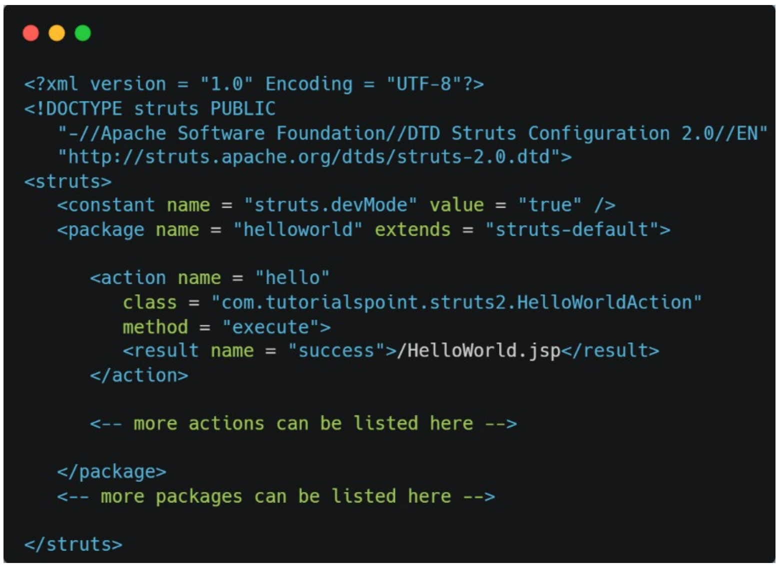
```
<?xml version = "1.0" Encoding = "UTF-8"?>
<web-app
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xmlns = "http://java.sun.com/xml/ns/javaee"
   xmlns:web = "http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   id = "WebApp_ID"
   version = "3.0"
>
   <display-name>Struts 2</display-name>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>

   <filter>
      <filter-name>struts2</filter-name>
      <filter-class>
         org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

- <web-app> represents the whole application.
- <servlet> is sub element of <web-app> and represents the servlet.
- <servlet-name> is sub element of <servlet> represents the name of the servlet.
- <servlet-class> is sub element of <servlet> represents the class of the servlet.
- <servlet-mapping> is sub element of <web-app>. It is used to map the servlet.
- <url-pattern> is sub element of <servlet-mapping>. This pattern is used at client side
- to invoke the servlet.

struts.xml

- The **struts.xml** file contains the configuration information that you will be modifying as actions are developed.
- This file can be used to override default settings for an application, for example *struts.devMode = false* and other settings which are defined in property file.
- This file can be created under the folder **WEB-INF/classes**.

Let us have a look at the struts.xml file we created in the Hello World example explained in previous chapter.



```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 = "helloworld" extends = "struts-default">

      <action name = "hello"
         class = "com.tutorialspoint.struts2.HelloWorldAction"
         method = "execute">
         <result name = "success">/HelloWorld.jsp</result>
      </action>

      <-- more actions can be listed here -->

   </package>
   <-- more packages can be listed here -->

</struts>
```

All struts configuration file needs to have the correct doctype as shown in our little example.

<struts> is the root tag element, under which we declare different packages using <package> tags. <package> allows separation and modularization of the configuration.

This is very useful when you have a large project and project is divided into different modules.

Sr.No	Attribute & Description
1	**name (required)** The unique identifier for the package
2	**extends** Which package does this package extend from? By default, we use struts-default as the base package.
3	**abstract** If marked true, the package is not available for end user consumption.

4	**namespace** Unique namespace for the actions

For example, if your project has three domains - business_application, customer_application and staff_application, then you could create three packages and store associated actions in the appropriate package.

The package tag has the following attributes –

The **constant** tag along with name and value attributes should be used to override any of the following properties defined in **default.properties**, like we just set **struts.devMode** property.

Setting **struts.devMode** property allows us to see more debug messages in the log file.

We define **action** tags corresponds to every URL we want to access and we define a class with execute() method which will be accessed whenever we will access corresponding URL.

Results determine what gets returned to the browser after an action is executed.

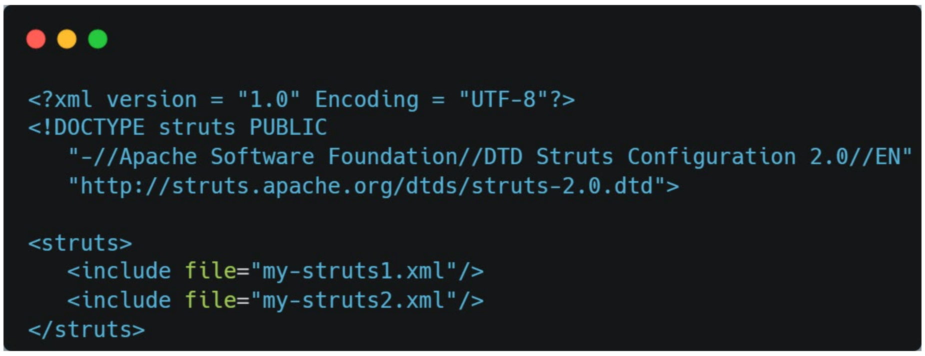
The string returned from the action should be the name of a result.

Results are configured per-action as above, or as a "global" result, available to every action in a package

Results have optional **name** and **type** attributes. The default name value is "success".

Struts.xml file can grow big over time and so breaking it by packages is one way of modularizing it, but **Struts** offers another way to modularize the struts.xml file.

You could split the file into multiple xml files and import them in the following fashion.



```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>
   <include file="my-struts1.xml"/>
   <include file="my-struts2.xml"/>
</struts>
```

The other configuration file that we haven't covered is the struts-default.xml.

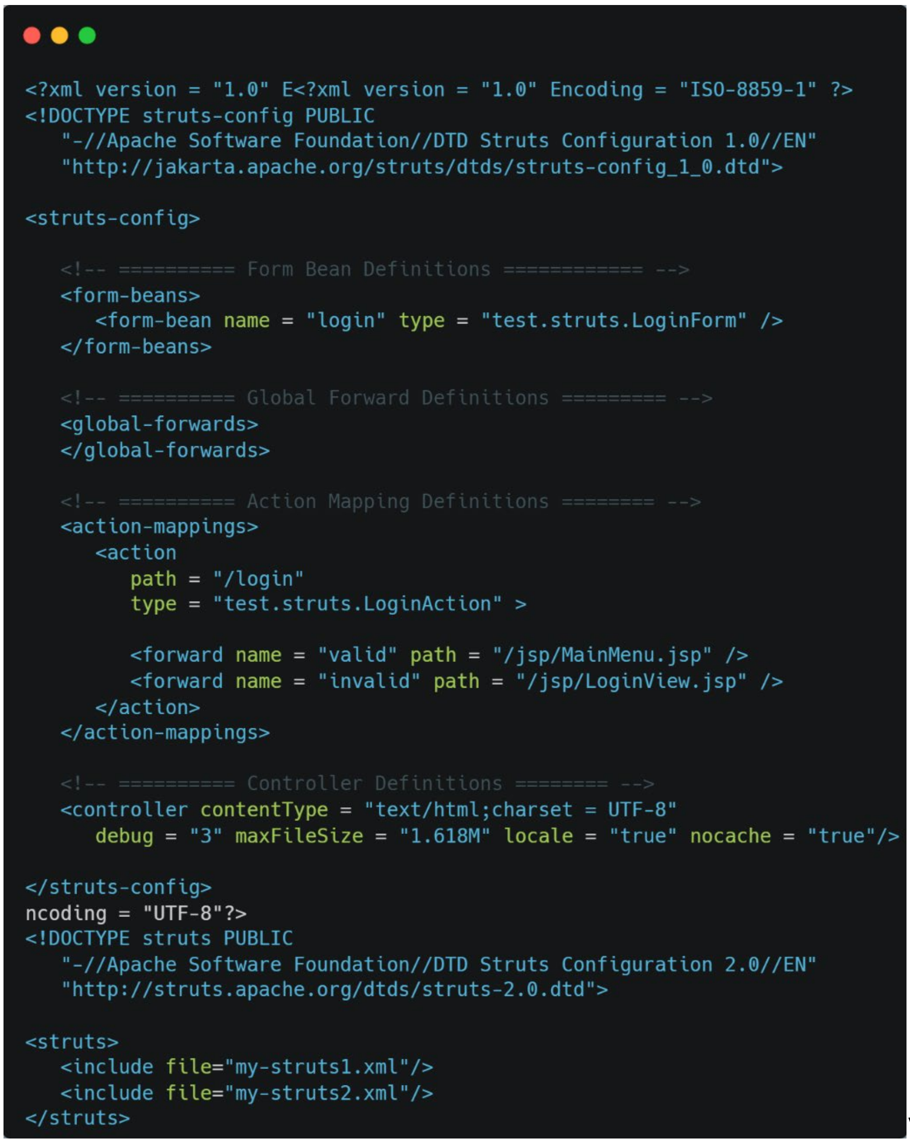
This file contains the standard configuration settings for Struts and you would not have to touch these settings for 99.99% of your projects.

struts-config.xml

- The struts-config.xml configuration file is a link between the View and Model components in the Web Client but you would not have to touch these settings for 99.99% of your projects.

The configuration file basically contains following main elements −

Sr.No	Interceptor & Description
1	**struts-config** This is the root node of the configuration file.
2	**form-beans** This is where you map your ActionForm subclass to a name. You use this name as an alias for your ActionForm throughout the rest of the strutsconfig.xml file, and even on your JSP pages.
3	**global forwards** This section maps a page on your webapp to a name. You can use this name to refer to the actual page. This avoids hardcoding URLs on your web pages.
4	**action-mappings** This is where you declare form handlers and they are also known as action mappings.
5	**controller** This section configures Struts internals and rarely used in practical situations.
6	**plug-in** This section tells Struts where to find your properties files, which contain prompts and error messages



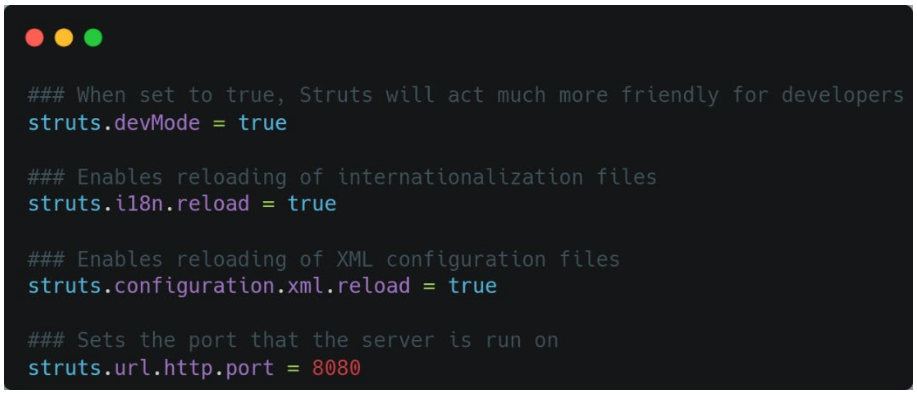
```
<?xml version = "1.0" E<?xml version = "1.0" Encoding = "ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>

   <!-- ========== Form Bean Definitions ============ -->
   <form-beans>
      <form-bean name = "login" type = "test.struts.LoginForm" />
   </form-beans>

   <!-- ========== Global Forward Definitions ========= -->
   <global-forwards>
   </global-forwards>

   <!-- ========== Action Mapping Definitions ======== -->
   <action-mappings>
      <action
         path = "/login"
         type = "test.struts.LoginAction" >

         <forward name = "valid" path = "/jsp/MainMenu.jsp" />
         <forward name = "invalid" path = "/jsp/LoginView.jsp" />
      </action>
   </action-mappings>

   <!-- ========== Controller Definitions ======== -->
   <controller contentType = "text/html;charset = UTF-8"
      debug = "3" maxFileSize = "1.618M" locale = "true" nocache = "true"/>

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

<struts>
   <include file="my-struts1.xml"/>
   <include file="my-struts2.xml"/>
</struts>
```

\

struts.properties

- This configuration file provides a mechanism to change the default behavior of the framework.
- Actually, all the properties contained within the **struts.properties** configuration file can also be configured in the **web.xml** using the **init-param**, as well using the constant tag in the **struts.xml** configuration file.
- But, if you like to keep the things separate and more struts specific, then you can create this file under the folder **WEB-INF/classes**.
- The values configured in this file will override the default values configured in **default.properties** which is contained in the struts2-core-x.y.z.jar distribution.

There are a couple of properties that you might consider changing using the **struts.properties** file –



```
### When set to true, Struts will act much more friendly for developers
struts.devMode = true

### Enables reloading of internationalization files
struts.i18n.reload = true

### Enables reloading of XML configuration files
struts.configuration.xml.reload = true

### Sets the port that the server is run on
struts.url.http.port = 8080
```

Interceptors are conceptually the same as servlet filters or the JDKs Proxy class. 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;

Examples include exception handling, file uploading, lifecycle callbacks, etc. In fact, as Struts2 emphasizes much of its functionality on interceptors, it is not likely to have 7 or 8 interceptors assigned per action.

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

Please look into Struts 2 documentation for complete detail on the abovementioned interceptors. But I will show you how to use an interceptor in general in your Struts application.

How to Use Interceptors?

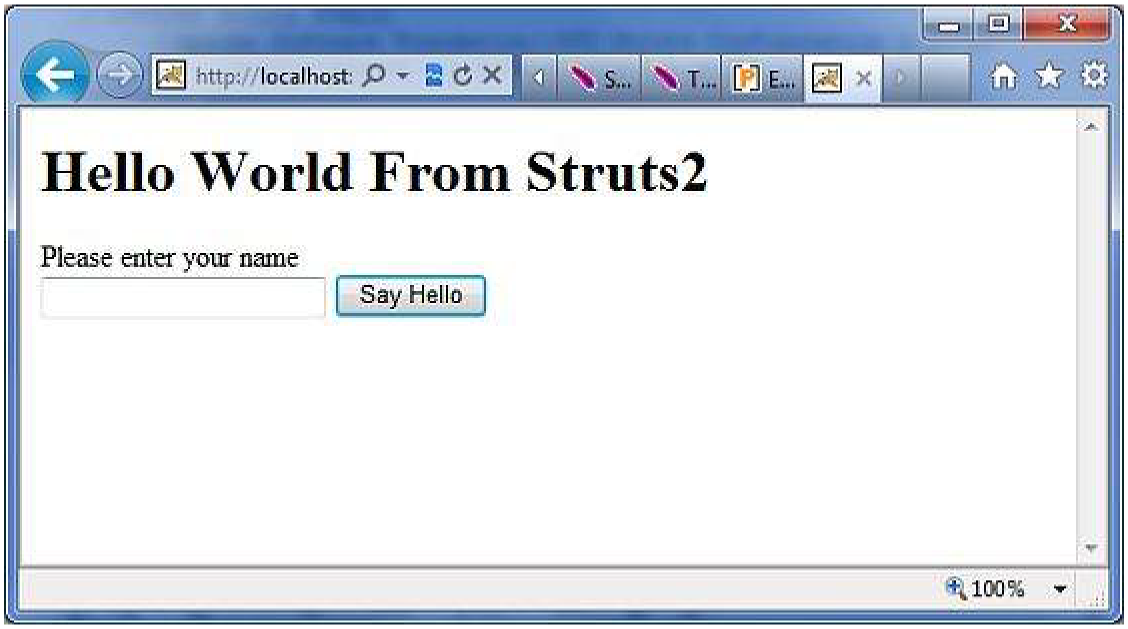
Let us see how to use an already existing interceptor to our "Hello World" program. We will use the **timer** interceptor whose purpose is to measure how long it took to execute an action method. At the same time, I'm using **params** interceptor whose purpose is to send the request parameters to the action. You can try your example without using this interceptor and you will find that **name** property is not being set because parameter is not able to reach to the action.

We will keep HelloWorldAction.java, web.xml, HelloWorld.jsp and index.jsp files as they have been created in **Examples** chapter but let us modify the **struts.xml** file to add an interceptor 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">
         <interceptor-ref name = "params"/>
         <interceptor-ref name = "timer" />
         <result name =
"success">/HelloWorld.jsp</result>
      </action>
   </package>
```

```
</struts>
```

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/index.jsp**. This will produce the following screen −



Hello World From Struts2

Please enter your name

Say Hello

Now enter any word in the given text box and click Say Hello button to execute the defined action. Now if you will check the log generated, you will find the following text −

```
INFO: Server startup in 3539 ms
27/08/2011 8:40:53 PM
com.opensymphony.xwork2.util.logging.commons.CommonsLogge
r info
INFO: Executed action [//hello!execute] took 109 ms.
```

Here bottom line is being generated because of **timer** interceptor which is telling that action took total 109ms to be executed.

In this chapter, we shall look deeper into Struts validation framework. At the Struts core, we have the validation framework that assists the application to run the rules to perform validation before the action method is executed.

Client side validation is usually achieved using Javascript. However, one should not rely upon client side validation alone. The best practices suggest that the validation should be introduced at all levels of your application framework. Now let us look at two ways of adding validation to our Struts project.

Here, we will take an example of an **Employee** whose name and age should be captured using a simple page, and we will put these two validations to make sure that the user always enters a name and age which should be in a range between 28 and 65.

Let us start with the main JSP page of the example.

Create Main Page

Let us write main page JSP file **index.jsp**, which will be used to collect Employee related information mentioned above.

```
<%@ 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>Employee Form</title>
   </head>

   <body>
      <s:form action = "empinfo" method = "post">
         <s:textfield name = "name" label = "Name" size =
"20" />
```

```
         <s:textfield name = "age" label = "Age" size =
"20" />
         <s:submit name = "submit" label = "Submit"
align="center" />
      </s:form>
   </body>
</html>
```

The index.jsp makes use of Struts tag, which we have not covered yet, but we will study them in tags related chapters. But for now, just assume that the s:textfield tag prints a input field, and the s:submit prints a submit button. We have used label property for each tag which creates label for each tag.

Create Views

We will use JSP file success.jsp which will be invoked in case defined action returns SUCCESS.

```
<%@ 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>Success</title>
   </head>

   <body>
      Employee Information is captured successfully.
   </body>
</html>
```

Create Action

So let us define a small action class **Employee**, and then add a method called **validate()** as shown below in **Employee.java** file. Make sure that your action class extends the **ActionSupport** class, otherwise your validate method will not be executed.

```
package com.tutorialspoint.struts2;

import com.opensymphony.xwork2.ActionSupport;

public class Employee extends ActionSupport {
   private String name;
   private int age;

   public String execute() {
       return SUCCESS;
   }

   public String getName() {
       return name;
   }

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

   public int getAge() {
       return age;
   }

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

   public void validate() {
      if (name == null || name.trim().equals("")) {
         addFieldError("name","The name is required");
      }

      if (age < 28 || age > 65) {
         addFieldError("age","Age must be in between 28 
and 65");
```

```
      }
   }
}
```

As shown in the above example, the validation method checks whether the 'Name' field has a value or not. If no value has been supplied, we add a field error for the 'Name' field with a custom error message. Secondly, we check if entered value for 'Age' field is in between 28 and 65 or not, if this condition does not meet we add an error above the validated field.

Configuration Files

Finally, let us put everything together using the **struts.xml** configuration file as follows –

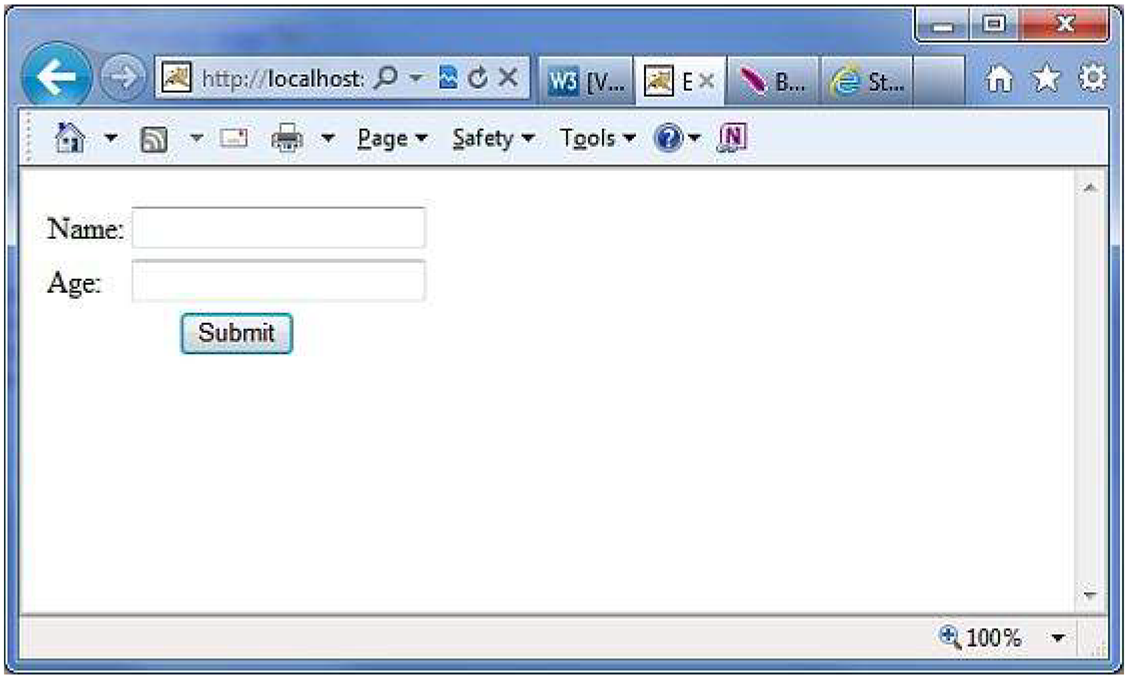
```
<?xml version = "1.0" Encoding = "UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts 
Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">

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

      <action name = "empinfo" 
         class = "com.tutorialspoint.struts2.Employee"
         method = "execute">
         <result name = "input">/index.jsp</result>
         <result name = "success">/success.jsp</result>
      </action>

   </package>
</struts>
```

Following is the content of **web.xml** file –

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

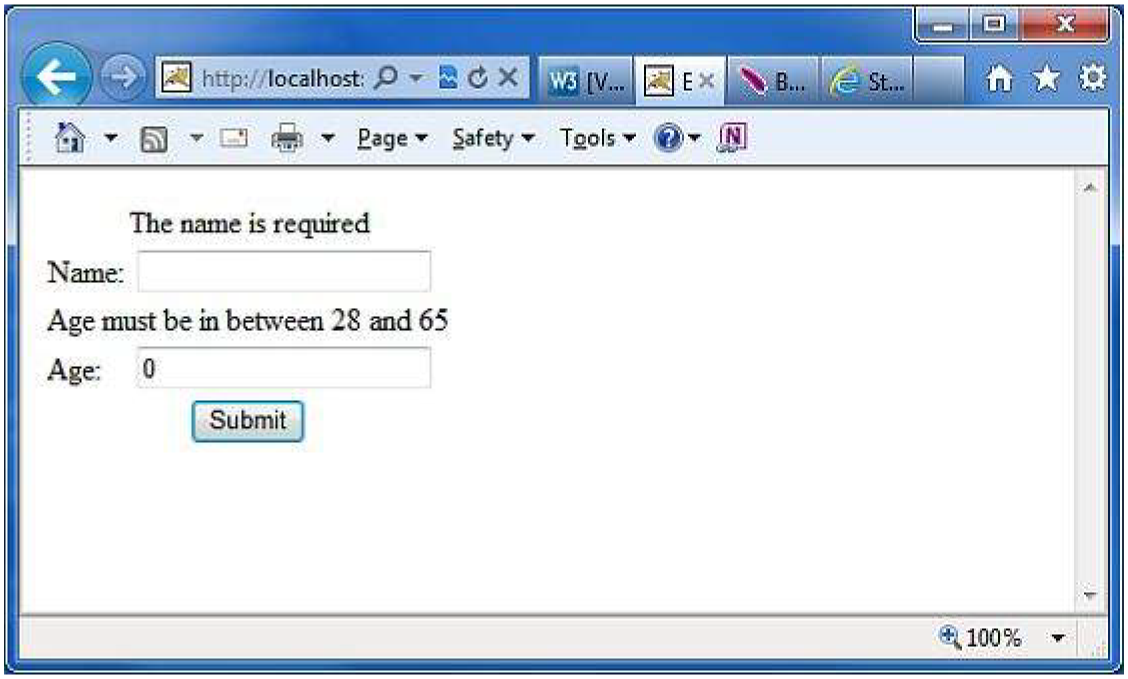
```
<web-app xmlns:xsi = "http://www.w3.org/2001/XMLSchema-
instance"
   xmlns = "http://java.sun.com/xml/ns/javaee"
   xmlns:web = "http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
   xsi:schemaLocation =
"http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   id = "WebApp_ID" version = "3.0">

   <display-name>Struts 2</display-name>

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

   <filter>
      <filter-name>struts2</filter-name>
      <filter-class>
         org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

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/index.jsp**. This will produce the following screen −


Name:

Age:

Now do not enter any required information, just click on **Submit** button. You will see the following result –

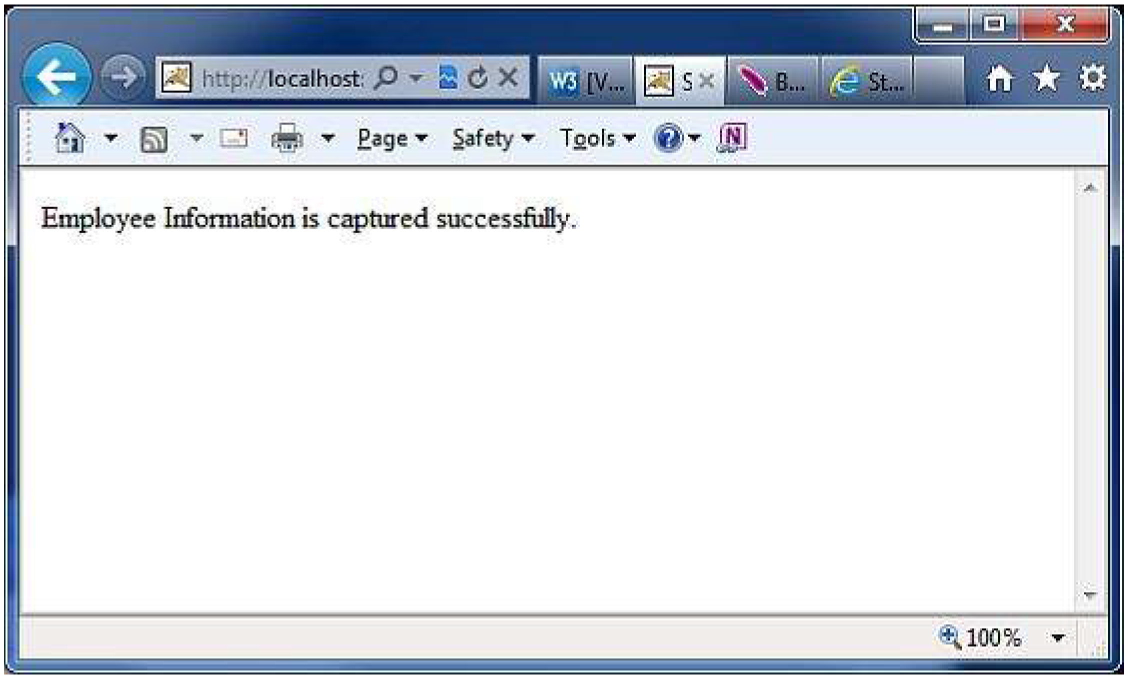

The name is required

Name:

Age must be in between 28 and 65

Age: 0

Enter the required information but enter a wrong From field, let us say name as "test" and age as 30, and finally click on **Submit** button. You will see the following result –



Employee Information is captured successfully.

How this Validation Works?

1. When the user presses the submit button, Struts 2 will automatically execute the validate method

2. If any of the **"if"** statements listed inside the method are true, Struts 2 will call its addFieldError method. If any errors have been added, then Struts 2 will not proceed to call the execute method.

3. Rather the Struts 2 framework will return **input** as the result of calling the action.

Hence, when validation fails and Struts 2 returns **input**, the Struts 2 framework will redisplay the index.jsp file. Since, we used Struts 2 form tags, Struts 2 will automatically add the error messages just above the form filed.

These error messages are the ones we specified in the addFieldError method call. The addFieldError method takes two arguments. The first, is the **form** field name to which the error applies and the second, is the error message to display above that form field.

```
addFieldError("name","The name is required");
```

To handle the return value of **input** we need to add the following result to our action node in **struts.xml.**

```
<result name = "input">/index.jsp</result>
```

XML Based Validation

The second method of doing validation is by placing an xml file next to the action class. Struts2 XML based validation provides more options of validation like email validation, integer range validation, form validation field, expression validation, regex validation, required validation, requiredstring validation, stringlength validation and etc.

The xml file needs to be named **'[action-class]'-validation.xml.** So, in our case we create a file called **Employee-validation.xml** with the following contents −

```
<!DOCTYPE validators PUBLIC
   "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
   "http://www.opensymphony.com/xwork/xwork-validator-
1.0.2.dtd">

<validators>
   <field name = "name">
      <field-validator type = "required">
         <message>
            The name is required.
         </message>
      </field-validator>
   </field>

   <field name = "age">
     <field-validator type = "int">
         <param name = "min">29</param>
         <param name = "max">64</param>
         <message>
            Age must be in between 28 and 65
         </message>
      </field-validator>
   </field>
```

```
</validators>
```

Above XML file would be kept in your CLASSPATH ideally along with class file. Let us have our Employee action class as follows without having **validate()** method –

```
package com.tutorialspoint.struts2;

import com.opensymphony.xwork2.ActionSupport;

public class Employee extends ActionSupport{
   private String name;
   private int age;

   public String execute() {
       return SUCCESS;
   }

   public String getName() {
       return name;
   }

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

   public int getAge() {
       return age;
   }

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

Rest of the setup will remain as it is i the previous example, now if you will run the application, it will produce same result what we received in previous example.

The advantage of having an xml file to store the configuration allows the separation of the validation from the application code.

You could get a developer to write the code and a business analyst to create the validation xml files. Another thing to note is the validator types that are available by default.

There are plenty more validators that come by default with Struts. Common validators include Date Validator, Regex validator and String Length validator. Check the following link for more detail Struts - XML Based Validators.

JSON is…

- A lightweight text based data-interchange format
- Completely language independent
- Based on a subset of the JavaScript Programming Language
- Easy to understand, manipulate and generate

JSON is NOT…

- Overly Complex
- A “document” format
- A markup language
- A programming language

Why use JSON?

- Straightforward syntax
- Easy to create and manipulate
- Can be natively parsed in JavaScript using **eval()**
- Supported by all major JavaScript frameworks
- Supported by most backend technologies

Much Like XML

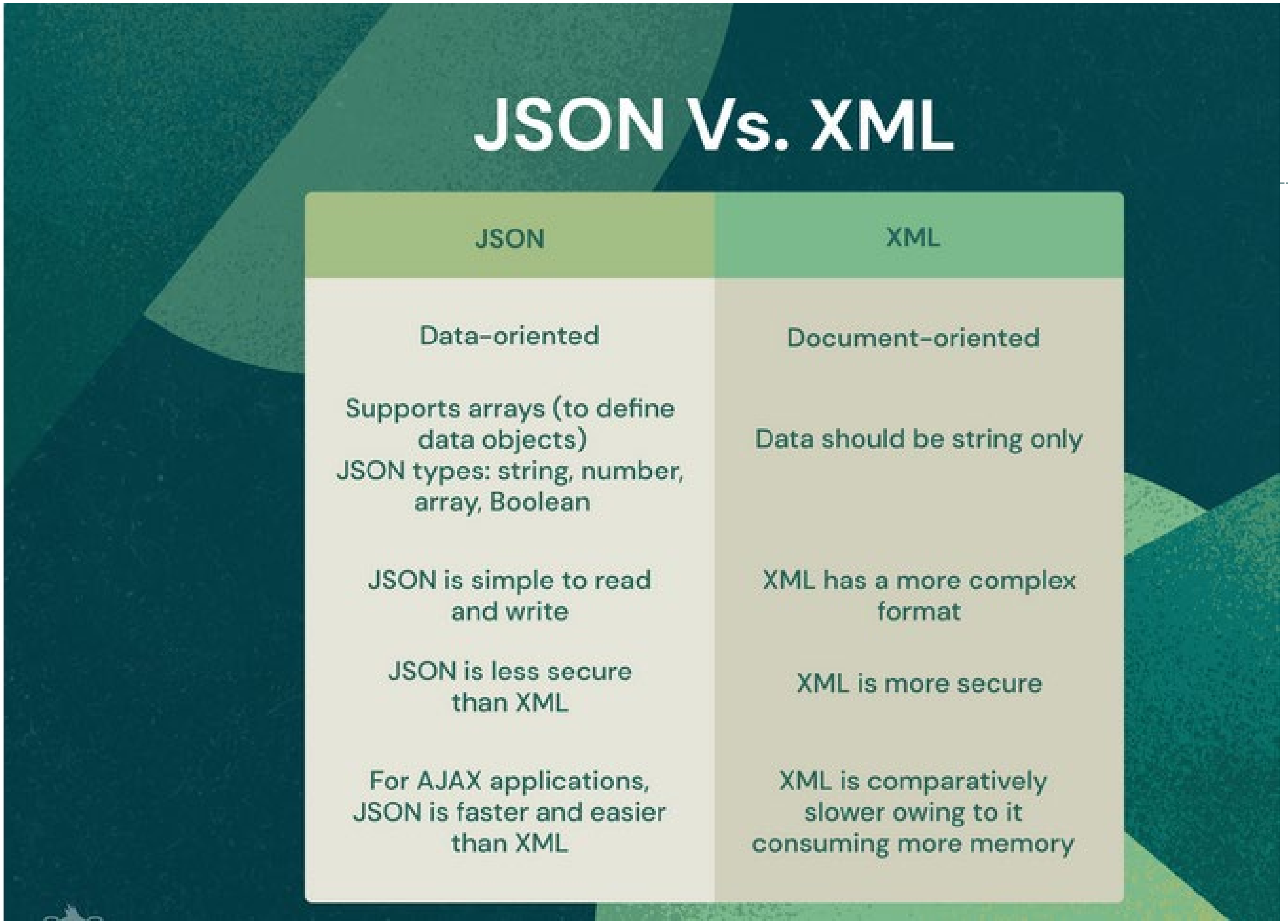
- Plain text formats
- “Self-describing“ (human readable)
- Hierarchical (Values can contain lists of objects or values)

Not Like XML

- Lighter and faster than XML
- JSON uses typed objects. All XML values are type-less strings and must be parsed at runtime.
- Less syntax, no semantics
- Properties are immediately accessible to JavaScript code

Knocks against JSON

- Lack of namespaces
- No inherit validation (XML has DTD and templates, but there is JSONlint)
- Not extensible
- It’s basically just ***not*** XML



JSON Vs. XML

JSON	XML
Data-oriented	Document-oriented
Supports arrays (to define data objects) JSON types: string, number, array, Boolean	Data should be string only
JSON is simple to read and write	XML has a more complex format
JSON is less secure than XML	XML is more secure
For AJAX applications, JSON is faster and easier than XML	XML is comparatively slower owing to it consuming more memory

How & When to use JSON

- Transfer data to and from a server
- Perform asynchronous data calls without requiring a page refresh
- Working with data stores
- Compile and save form or user data for local storage

JSON Object Syntax

- Unordered sets of name/value pairs
- Begins with **{** (left brace)
- Ends with **}** (right brace)
- Each name is followed by **:** (colon)
- Name/value pairs are separated by **,** (comma)

JSON Example

```
var employeeData = {
  "employee_id": 1234567,
  "name": “Freyan",
  "hire_date": "1/1/2021",
  "location": "Norwalk, CT",
  "consultant": false
};
```

Arrays in JSON

- An ordered collection of values
- Begins with [(left bracket)
- Ends with] (right bracket)
- Name/value pairs are separated by , (comma)

JSON Array Example

```
var employeeData = {
  "employee_id": 1236937,
  "name": “khyati",
  "hire_date": "1/1/2022",
  "location": "Norwalk, CT",
  "consultant": false,
   "random_nums": [ 24,65,12,94 ]
};
```

Data Types: Strings

- Sequence of 0 or more Unicode characters
- Wrapped in "double quotes“
- Backslash escapement

Data Types: Numbers

- Integer
- Real
- Scientific
- No octal or hex
- No NaN or Infinity – Use **null** instead.

Data Types: Booleans & Null

- Booleans: true or false
- Null: A value that specifies nothing or no value.

Data Types: Objects & Arrays

- Objects: Unordered key/value pairs wrapped in { }
- Arrays: Ordered values wrapped in [ ]

JSON objects

- JSON Array of Numbers
- [12, 34, 56, 43, 95]

- JSON Array of Booleans
- [true, true, false, false, true]

- JSON Array of Objects
- {"employees":[
-     {"name":"Ram", "email":"ram@gmail.com", "age":23},
-     {"name":"Shyam", "email":"shyam23@gmail.com", "age":28},
-     {"name":"John", "email":"john@gmail.com", "age":33},
-     {"name":"Bob", "email":"bob32@gmail.com", "age":41}
-]}

JSON Multidimensional Array

- [
- ["a", "b", "c"],
- ["m", "n", "o"],
- ["x", "y", "z"]
-]

Java JSON

- The **json.simple** library allows us to read and write JSON data in Java.
- we can encode and decode JSON object in java using json.simple library.
- The org.json.simple package contains important classes for JSON API.
- JSONValue
- JSONObject
- JSONArray
- JsonString
- JsonNumber

Encoding JSON in Java

- import org.json.simple.JSONObject;
- public class JsonExample1{
- public static void main(String args[]){
- JSONObject obj=new JSONObject();
-   obj.put("name","sonoo");
-   obj.put("age",new Integer(27));
-   obj.put("salary",new Double(600000));
-    System.out.print(obj);
- }}

Java JSON Encode using List

- import java.util.ArrayList;
- import java.util.List;
- import org.json.simple.JSONValue;
- public class JsonExample1{
- public static void main(String args[]){
-   List arr = new ArrayList();
-   arr.add("sonoo");
-   arr.add(new Integer(27));
-   arr.add(new Double(600000));
-   String jsonText = JSONValue.toJSONString(arr);
-   System.out.print(jsonText);
- }}

Java JSON Decode

```
import org.json.simple.JSONObject;
import org.json.simple.JSONValue;
public class JsonDecodeExample1 {
public static void main(String[] args) {
    String s="{\"name\":\"sonoo\",\"salary\":600000.0,\"age\":27}";
    Object obj=JSONValue.parse(s);
    JSONObject jsonObject = (JSONObject) obj;

    String name = (String) jsonObject.get("name");
    double salary = (Double) jsonObject.get("salary");
    long age = (Long) jsonObject.get("age");
    System.out.println(name+" "+salary+" "+age);
}
}
```

ValueStack

- A valueStack is simply a stack that contains application specific objects such as action objects and other model object.
- At the execution time, action is placed on the top of the stack.
- We can put objects in the valuestack, query it and delete it.

Methods of ValueStack interface

- **public String findString(String expr)** finds the string by evaluating the given expression.
- **public Object findValue(String expr)** finds the value by evaluating the specified expression.
- **public Object findValue(String expr, Class c)** finds the value by evaluating the specified expression.
- **public Object peek()** It returns the object located on the top of the stack.
- **public Object pop()** It returns the object located on the top of the stack and removes it.
- **public void push(Object o)** It puts the object on the top of the stack.
- **public void set(String key, Object value)** It sets the object on the stack with the given key. It can be get by calling the findValue(key) method.
- **public int size()** It returns the number of objects from the stack.

OGNL

- The **Object Graph Navigation Language** (OGNL) is an expression language.
- The struts framework sets the **ValueStack** as the root object of OGNL. Notice that action object is pushed into the ValueStack. We can direct access the action property.
- <s:property value="username"/>
- Here, username is the property key.
- The struts framework places other objects in ActionContext also e.g. map representing the **request**, **session**, **application** scopes.
- To get these values i.e. not the action property, we need to use # notation. For example to get the data from session scope, we need to use #session as given in the following example:
- <s:property name="#session.username"/>
- (or)
- <s:property name="#session['username']"/>

The Value Stack

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

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.

The value stack can be accessed via the tags provided for JSP, Velocity or Freemarker. There are various tags which we will study in separate chapters, are used to get and set struts 2.0 value stack. You can get valueStack object inside your action as follows −

ActionContext.getContext().getValueStack()

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

It is important to understand that the Action object is always available in the value stack. So, therefore if your Action object has properties **“x”** and **“y”** there are readily available for you to use.

Objects in the ActionContext are referred using the pound symbol, however, the objects in the value stack can be directly referenced.

For example, if **employee** is a property of an action class, then it can be referenced as follows −

<s:property value = "name"/>

instead of

<s:property value = "#name"/>

If you have an attribute in session called "login" you can retrieve it as follows −

<s:property value = "#session.login"/>

OGNL also supports dealing with collections - namely Map, List and Set. For example to display a dropdown list of colors, you could do −

<s:select name = "color" list = "{'red','yellow','green'}" />

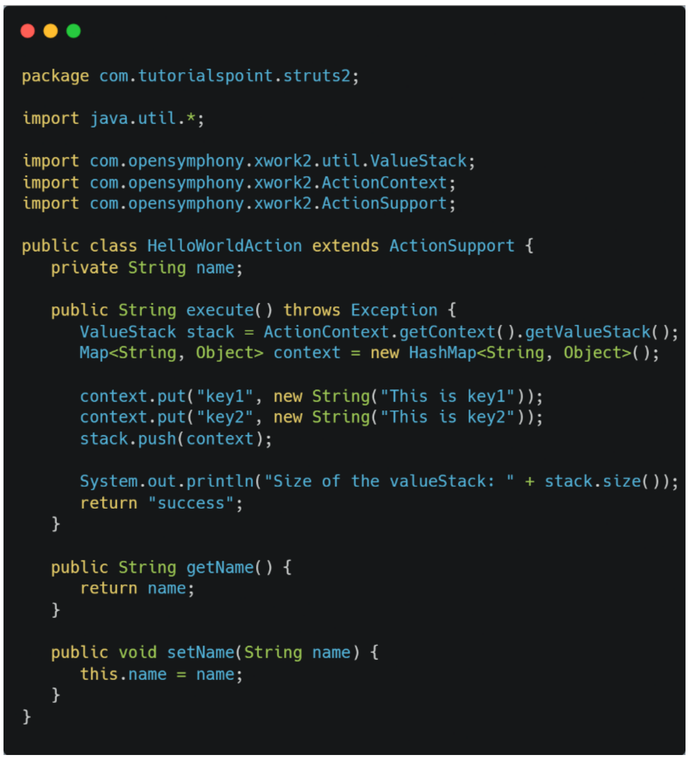
The OGNL expression is clever to interpret the "red","yellow","green" as colours and build a list based on that.

The OGNL expressions will be used extensively in the next chapters when we will study different tags. So rather than looking at them in isolation, let us look at it using some examples in the Form Tags / Control Tags / Data Tags and Ajax Tags section.

ValueStack/OGNL Example

Create Action

Let us consider the following action class where we are accessing valueStack and then setting few keys which we will access using OGNL in our view, i.e., JSP page.

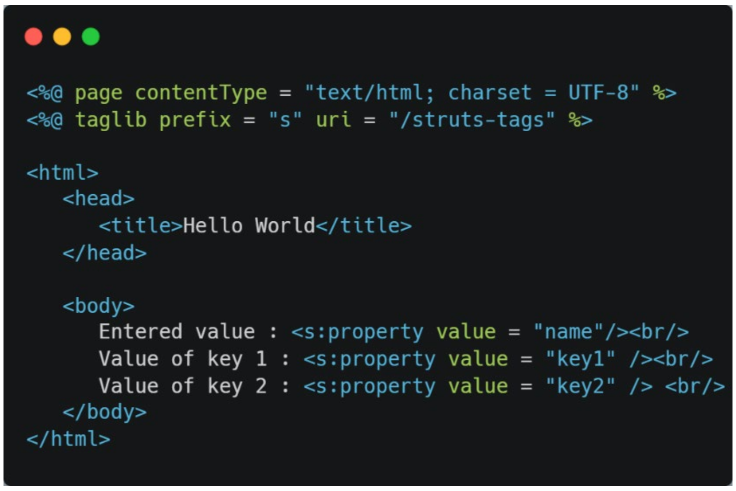


```java
package com.tutorialspoint.struts2;

import java.util.*;

import com.opensymphony.xwork2.util.ValueStack;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;

public class HelloWorldAction extends ActionSupport {
   private String name;

   public String execute() throws Exception {
      ValueStack stack = ActionContext.getContext().getValueStack();
      Map<String, Object> context = new HashMap<String, Object>();

      context.put("key1", new String("This is key1"));
      context.put("key2", new String("This is key2"));
      stack.push(context);

      System.out.println("Size of the valueStack: " + stack.size());
      return "success";
   }

   public String getName() {
      return name;
   }

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

Actually, Struts 2 adds your action to the top of the valueStack when executed. So, the usual way to put stuff on the Value Stack is to add getters/setters for the values to your Action class and then use <s:property> tag to access the values. But I'm showing you how exactly ActionContext and ValueStack work in struts.

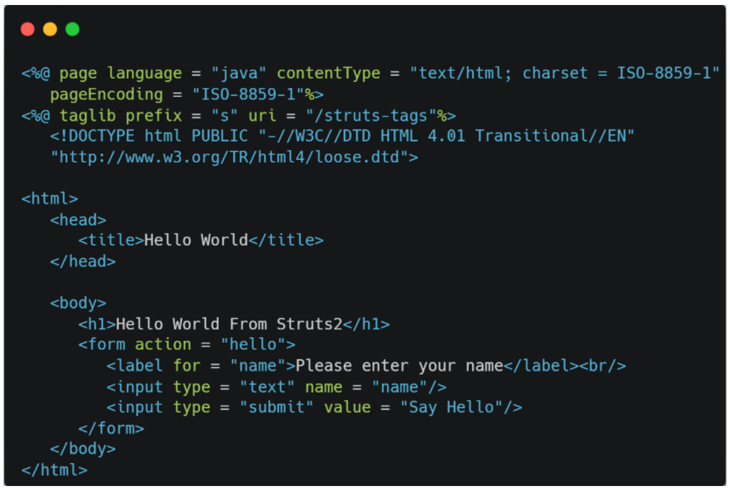
Create Views

Let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project. This view will be displayed in case action returns success –



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

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

   <body>
      Entered value : <s:property value = "name"/><br/>
      Value of key 1 : <s:property value = "key1" /><br/>
      Value of key 2 : <s:property value = "key2" /> <br/>
   </body>
</html>
```

We also need to create **index.jsp** in the WebContent folder whose content is as follows –



```
<%@ 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>Hello World</title>
   </head>

   <body>
      <h1>Hello World From Struts2</h1>
      <form action = "hello">
         <label for = "name">Please enter your name</label><br/>
         <input type = "text" name = "name"/>
         <input type = "submit" value = "Say Hello"/>
      </form>
   </body>
</html>
```

Configuration Files

Following is the content of **struts.xml** file –

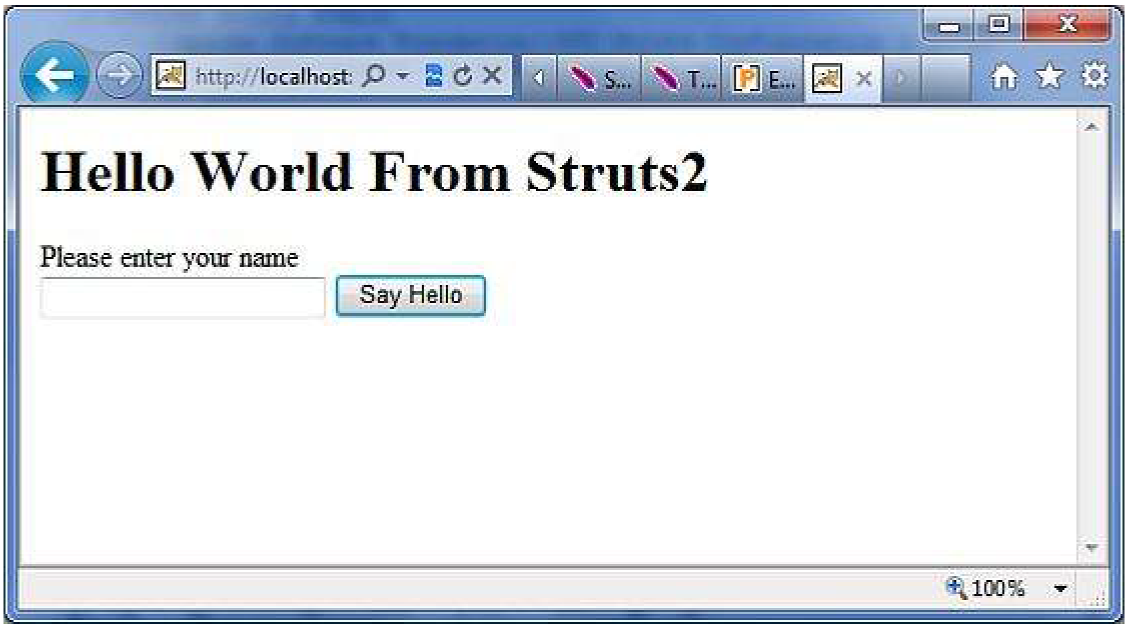
```
<?xml version = "1.0" Encoding = "UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">

<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">/HelloWorld.jsp</result>
      </action>

   </package>
</struts>
```

Following is the content of **web.xml** file –

```
<?xml version = "1.0" Encoding = "UTF-8"?>
<web-app xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xmlns = "http://java.sun.com/xml/ns/javaee"
   xmlns:web = "http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   id = "WebApp_ID" version = "3.0">

   <display-name>Struts 2</display-name>

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

   <filter>
      <filter-name>struts2</filter-name>
      <filter-class>
         org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

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/index.jsp**. This will produce the following screen



Hello World From Struts2

Please enter your name

Now enter any word in the given text box and click "Say Hello" button to execute the defined action. Now, if you will check the log generated, you will find the following text at the bottom −

Size of the valueStack: 3

This will display the following screen, which will display whatever value you will enter and value of key1 and key2 which we had put on ValueStack