Developing Applications Using Java Web Frameworks

Developing Applications Using Java Web Frameworks
Learner's Guide

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2014

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as learning-to-learn, thinking, adaptability, problem solving, positive attitude etc. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/ organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

- Evaluation of instructional process and instructional materials

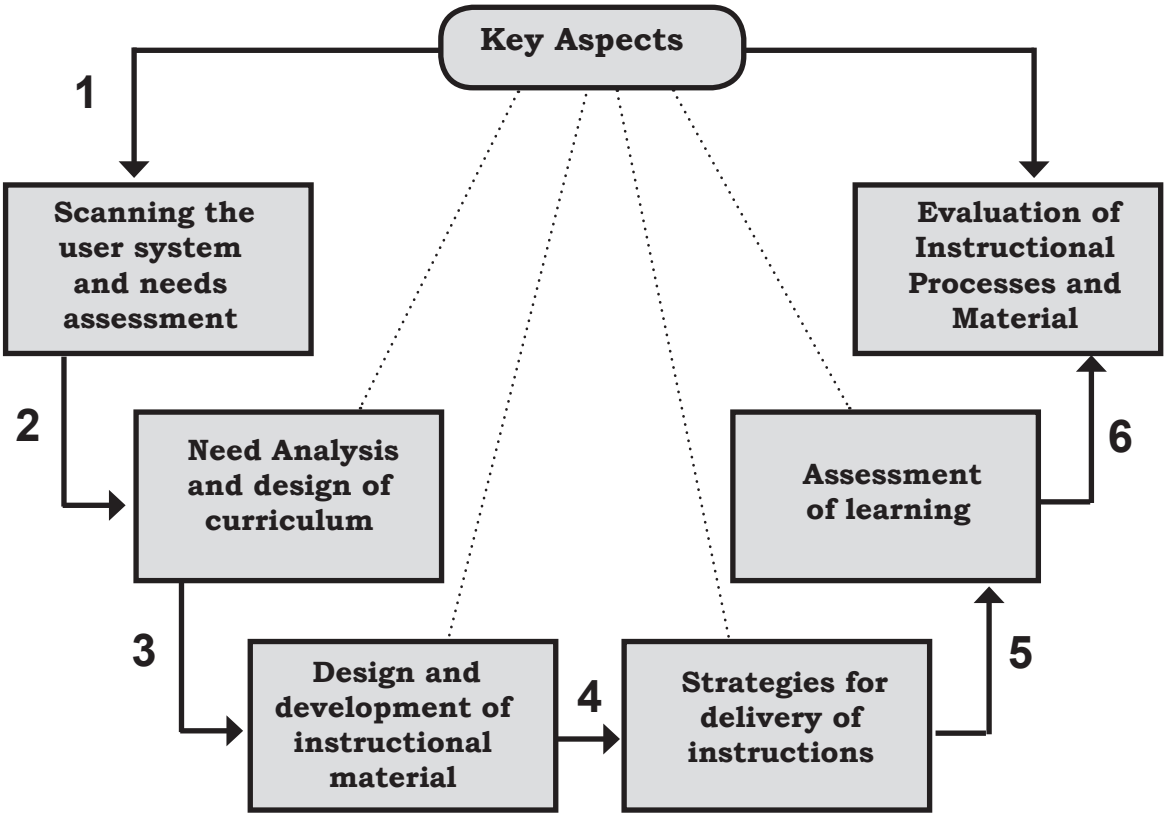
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Industry Recruitment Profile Survey - The Industry Recruitment Profile Survey was conducted across 1581 companies in August/September 2000, representing the Software, Manufacturing, Process Industry, Insurance, Finance & Service Sectors.

Aptech New Products Design Model



Onlinevarsity
your e-way to learning

WRITE-UPS BY

EXPERTS AND **LEARNERS**

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE

FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

Preface

The primary aim of 'Developing Applications Using Java Web Frameworks' book is to teach the students how to develop rich and interactive Web applications based on Model-View-Controller (MVC) architectural pattern. The book teaches the students how to work with the latest MVC frameworks namely, Struts 2.0 and JSF 2.0. Apache has provided an open source MVC framework named Struts, whereas Sun Microsystems has developed JSF framework, which is an MVC component-based framework used for developing Web applications that are similar to desktop applications. Both the frameworks provides a variety of tags, components, and Application Programming Interface (API) methods and classes to develop Web applications according to organizational needs.

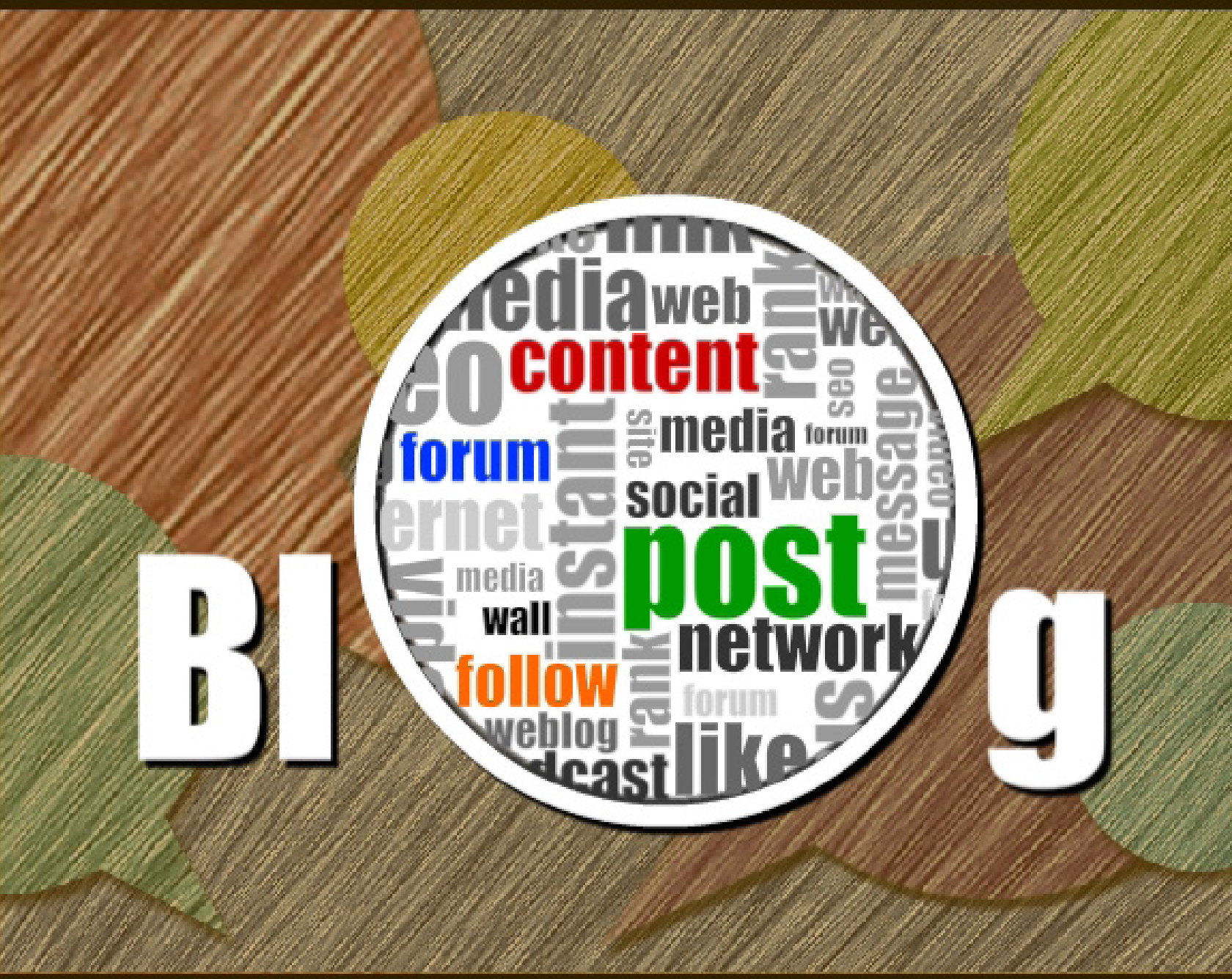
The knowledge and information in this book is the result of the concentrated effort of the Design Team, which is continuously striving to bring to you the latest, the best, and the most relevant subject matter in Information Technology. As a part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends and learner requirements.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Centre at Aptech's corporate office, Mumbai.

Design Team

Onlinevarsity

your e-way to learning



Balanced Learner-Oriented Guide

for enriched learning available @

Table of Contents

Sessions

Onlinevarsity

your e-way to learning

G ROWTH

R ESEARCH

O BSERVATION

U PDATES

P ARTICIPATION

www.onlinevarsity.com

01 Introduction to Struts Framework

Welcome to the Session, **Introduction to Struts Framework**.

This session introduces the Model-View-Controller (MVC) architectural pattern. It briefly explains the Model 1 and Model 2 approaches used for designing Java Web applications. Further, the session introduces the Struts framework based on MVC pattern. It explains the features, components, and architecture of Struts 1 framework in detail. Finally, the session concludes with the explanation of the steps used for developing Java Web applications by using Struts 1 framework.

In this Session, you will learn to:

- Explain Model-View-Controller (MVC) architecture
- Explain the role of components involved in MVC architecture
- Explain JSP Model 1 and JSP Model 2 approaches
- Describe Struts framework
- List the features and components of Struts 1 framework
- Explain the architecture of Struts 1 framework
- Explain the process of developing Web applications using Struts 1 components

Session
01

Introduction to Struts Framework

1.1 Introduction

Internet has made a paradigm shift in how people trade and communicate with each other. It has influenced the world to an extent that had never been seen with any other human invention. There is an essential need felt by everyone to have a presence in the virtual world through the Internet. Powerful Websites have become an absolute necessity to everyone, be it an individual or a big corporation. Initially, the Websites were designed to provide static information to its users. However, with time, the need for more and more interactivity on the Web pages led to the development of dynamic Web pages.

Consider the example of a bank's Website. It provides various interactive and dynamic options based on the customized needs and requirements of the customers. One of the key areas of designing a bank Website is its offerings provided on the various business needs. Thus, designing a Website with reusable software components is a difficult task.

To create complex applications, normally the architect consultants involved in the application designing adopt design patterns. A design pattern is the one which describes a proven solution to a recurring problem in the application development.

One such pattern which is widely acceptable among the Website designing is the Model-View-Controller (MVC). The MVC is an architectural pattern whose main concern is to separate data from the user interface. The separation of the data from the user interface helps the developer because changes to the user interface can be made without affecting the underlying data handling logic. Also, the data can be reorganized without changing the user interface.

1.2 MVC Architectural Pattern

MVC architectural pattern enables the separation of data (Model) from the user interface (View). This is made possible by introducing an intermediate component called the Controller. The Controller defines how the user interfaces react to the user input.

Figure 1.1 shows the components of the MVC architectural pattern.

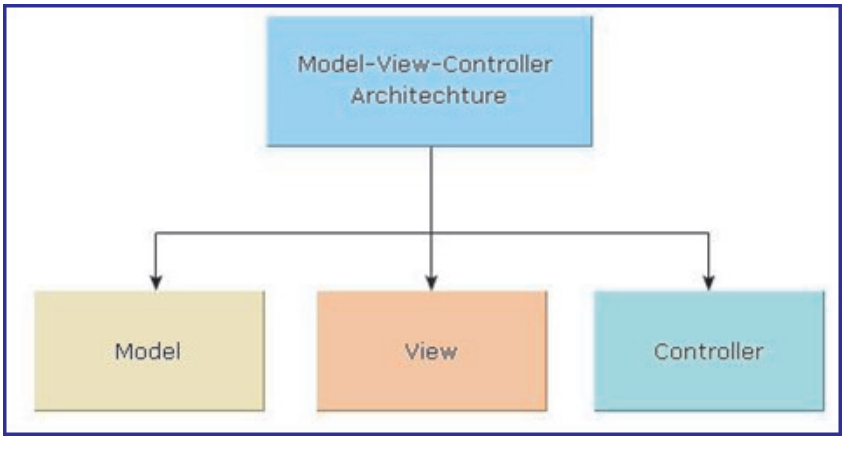


Figure 1.1: MVC Architectural Pattern

The relationship between the three components of MVC architecture is as follows:

- **View and Controller**

  Creation and selection of View is the responsibility of the Controller.

- **Model and View**

  The View is dependent on the Model. If any change is to be made to the Model, then there may be a requirement to make parallel changes in the View also.

- **Model and Controller**

  The Controller is dependent on the Model. If any change is to be made to the Model, then there may be a requirement to make parallel changes in the Controller also.

1.2.1 Role of Components in MVC

The role of the MVC architecture components in developing Web applications is as follows:

- **Model**

  The role of the Model component of MVC architecture is as follows:

  - It encapsulates the domain logic of an application.
  - It manages information and notifies observers whenever that information changes.
  - It contains only data and functionality that are related by a common purpose. If you need to model two groups of unrelated data and functionality, you create two separate models.
  - It makes it very easy to map real-world entities into a model.

- **View**

  The role of the View component of MVC architecture is as follows:

  - It obtains data from the model and presents it to the user. The View represents the output and/or input of the application.
  - It generally has free access to the model, but should not change the state of the model.
  - It reads data from the model using query methods provided by the model.

- **Controller**

  The role of the Controller component of MVC architecture is as follows:

  - It receives and translates input to requests on the Model or View.
  - It provides the means by which a user can interact with the application.
  - It accepts input from the user and instructs the Model and View to perform actions based on that input.

- It maps the end-user action to the application response. For example, if the user clicks the mouse button or chooses a menu item, the Controller is responsible for determining how the application should respond.
- It is responsible for calling methods on the model that changes the state of the model.

1.3 Java Web Development Models

With the introduction of JSP technology, Sun Microsystems provided a road map for developing JSP-based Web applications. It defined two models which provided approaches for designing Java-based Web applications. These are as follows:

Model 1 – It was the primary solution implemented when JSPs were first introduced.

Model 2 – It presented the correct way for building JSP-based Web applications. Model 2 was adopted as base model for MVC-based Web frameworks such as Struts.

1.3.1 JSP Model 1 Architecture

Model 1 architecture is one of the approaches used for developing a Web application. In Model 1 architecture, the JSP page is not only responsible for processing requests and sending back replies to the clients, but also for extracting the HTTP request parameters, invoking the business logic, and handling the HTTP session.

Figure 1.2 shows JSP Model 1 architecture.

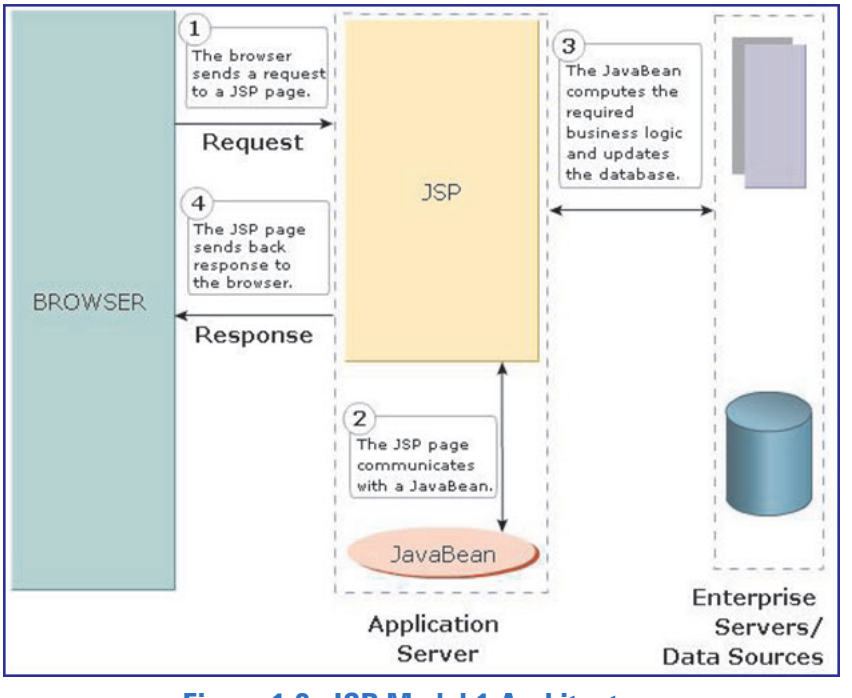


Figure 1.2: JSP Model 1 Architecture

In this model, there is no extra Servlet involved in the process. The client request is sent directly to a JSP page, which may communicate with JavaBeans or other services, but ultimately the JSP page selects the next page for the client to view.

JSP Model 1 Architecture has a page-centric architecture. In this architecture, the application is composed of a series of interrelated JSP pages and these JSP pages handle all aspects of application including presentation, control, and the business logic.

The disadvantages of using JSP Model 1 architecture are as follows:

1. It suffers from navigation problem. If you change the name of a JSP page that is referenced by other pages, you must change the name at many locations in the application.
2. It is difficult to maintain the application based on JSP Model 1 architecture, as it is inflexible. For example, if an application requires a large number of request processing events, then it will lead to a large Java code being embedded within the JSP page. Maintenance of such a page by a designer becomes a complicated task.

1.3.2 JSP Model 2 Architecture

Model 2 architecture is the second approach that was used for developing Web applications. It separates the 'Business Logic' from the 'Presentation Logic'. Besides this, Model 2 has an additional component - a Controller.

Here, a Servlet acts as a Controller. The responsibility of the Controller Servlet is to process the incoming request and instantiate a Model - a Java object or a bean to compute the business logic. It is also responsible for deciding to which JSP page the request should be forwarded.

JSP page is responsible for the View component and retrieves the objects created by the Servlet and extracts dynamic content for insertion within a template for display.

Figure 1.3 shows the JSP Model 2 architecture.

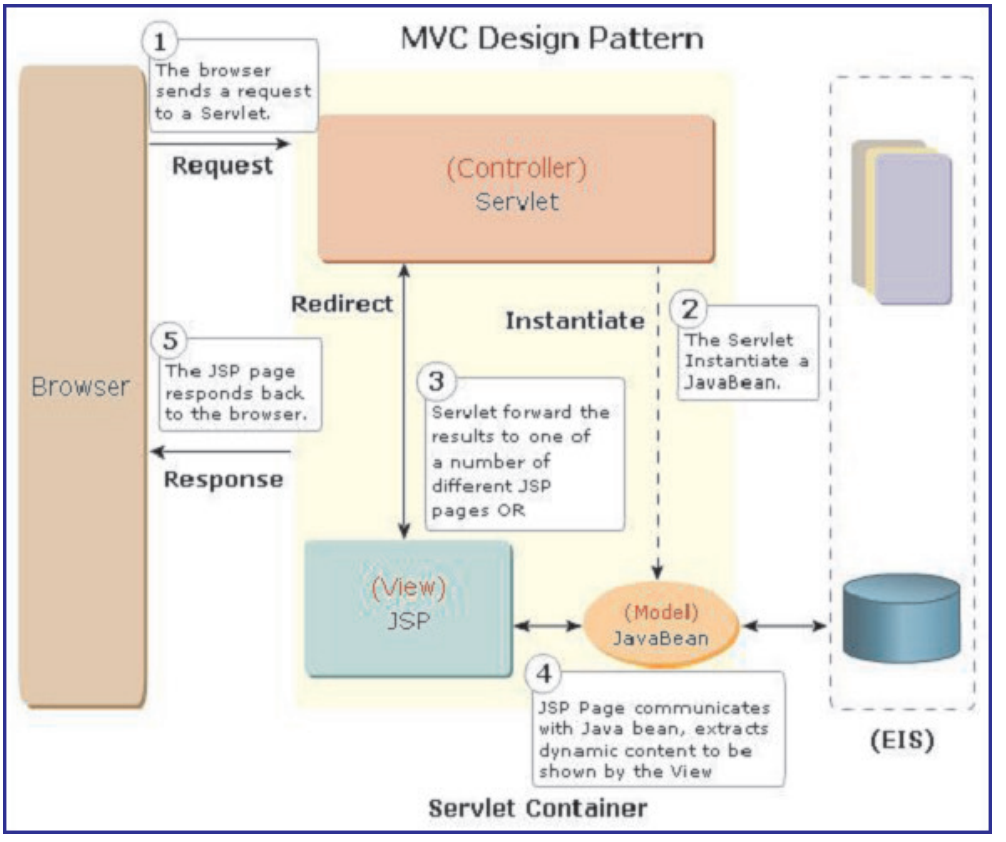


Figure 1.3: JSP Model 2 Architecture

Model 2 has a Servlet-centric architecture. The Servlet acts as controller and selects suitable business logic to handle the request. It also redirects the results to appropriate View. JSP acts as View for response generation. Such, a separation of business and presentation logic makes it possible to accommodate multiple interfaces to a Web application. These may include Web or wireless or GUI.

The advantage of using JSP Model 2 architecture is that Model 2 applications are easier to build, maintain, and extend because Views do not refer to each other directly. Other advantage is that the Model 2 Controller Servlet provides a single point of control for security and logging. It often encapsulates incoming data into a form usable by the back-end MVC Model.

On the other side, the disadvantage of using JSP Model 2 architecture is that it increases the design complexity as it introduces some extra classes/code due to the separation of Model, View, and Controller components.

1.4 Struts

According to Booch G., Rumbaugh J., and Jacobson I., a framework is 'An architectural pattern provides an extensible template for application within a domain. When you specify a framework, you specify the skeleton of its architecture, together with the slots, tabs, knobs, and dials that you expose to users, who want to adapt that framework to their own context.'

Another way of defining a framework is that it provides a structural support in the form of classes that can be extended for reuse and a functional toolkit in the form of software libraries.

The difference between a software library and a framework is that a software library contains functions or routines that an application can invoke, whereas a framework provides basic generic components that are extended by the application to provide certain set of functions.

1.4.1 Role of a Framework

Framework captures the design decisions that are common to an application domain. Thus, the role of a framework is as follows:

- They propagate design reuse over code reuse.
- They can be written in any programming language such as Java and C++ and thus can be executed and reused directly.
- They are created and used for a particular application domain.
- They not only allow building application faster, but the applications have similar structure and are easier to maintain.

Figure 1.4 depicts the role of a framework.

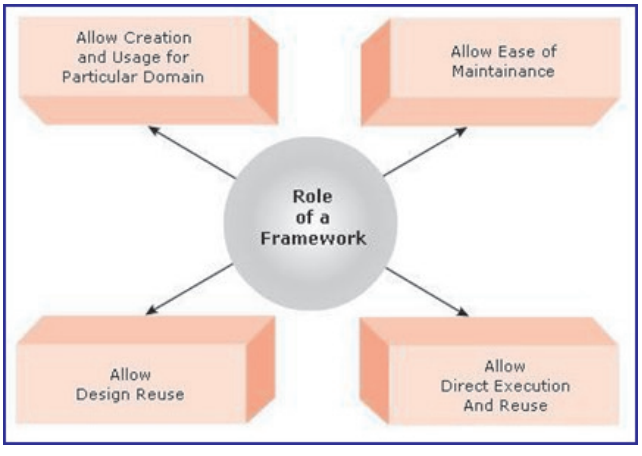


Figure 1.4: Role of a Framework

1.4.2 Characteristics of a Framework

The different characteristics of a framework are:

- A framework consists of various classes and components. Each of these classes or component gives an abstraction of a specific concept.
- A framework also gives a definition of how these classes or components work with each other to provide a solution to a problem.
- Framework components and classes can be reused. This is because framework provides a general behavior that various applications can use it.
- Framework provides an organized pattern of an application.

Figure 1.5 depicts the characteristics of a framework such as Struts.

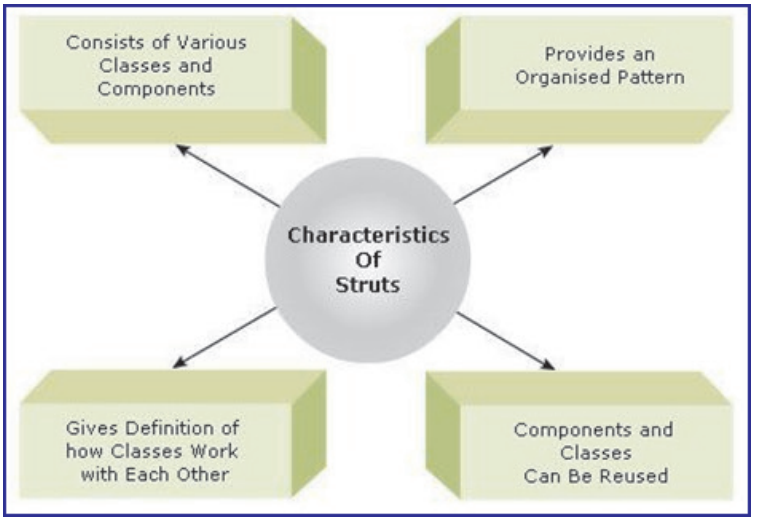


Figure 1.5: Characteristics of Struts Framework

1.4.3 Description of Controller

Frameworks are needed to incorporate the following features in software design process. They are as follows:

- **Modularity**

  Modularity is achieved by encapsulating the application specific detail, which may change and separate it from a stable interface.

- **Extensibility**

  Extensibility in design means the ease with which a new behavior can be added depending upon the application domain.

- **Reusability**

  Design level reusability is enhanced by frameworks by defining generic components that can be reapplied to create new applications.

- **Inversion of Control (IOC)**

  Inversion of Control is the design pattern, which reduces the tight coupling between associating objects by creating separate event handler objects.

1.5 Introduction to Struts

Struts is an open source Java-based framework. The purpose of this framework is to simplify the development of Java Enterprise Edition (Java EE) Web applications based on MVC architecture. Struts extends the Java Servlet API and has become the default standard for building Web applications in Java.

The Struts framework resides in the Web tier and the Struts applications are managed by the Web container residing on the Web server. Thus, the applications can use the services such as, handling requests through HTTP and HTTPS protocol provided by the Web container.

Session
01

Introduction to Struts Framework

The Struts framework was developed by Craig McClanahan and supported by Apache Software Foundation's (ASF) Jakarta group. Struts 1 is the first version of Apache Struts framework. Later in the year 2007, Apache released Struts 2 framework combining the features of WebWork framework to offer new enhancements and refinements in the original Struts framework.

Struts framework basically consists of two entities - first, it is an MVC Web application that can be extended or added by the applications. Second, it provides a set of libraries, as tool set to build Web applications. In other words,

1.5.1 Features of Struts 1 Framework

Struts offer a unified framework based on MVC architecture and a collection of utilities to develop Web applications.

Some of the features of Struts are as follows:

- **Code Extensibility**

  Many Struts values are represented in XML or in property files rather than being hard-coded in a Java program. This means that changes can be made to the file without recompiling the Java code. This approach allows the Web designer and Java programmers to focus on their respective tasks.

- **Support for Localization and Internationalization**

  Struts support localization and internationalization in the form of ResourceBundle.

- **Simpler Request Processing Mechanism**

  Struts provide several utilities, such as `StrutValidatorUtil` class, `MessageResource` class, and so on which simplifies the request processing mechanism.

- **Model – View Communication**

  Struts provide a set of custom JSP tags which allows you to create HTML forms that directly associates with Java Bean component. This association allows initial form field values from Java objects to be displayed and allows redisplay of form with some or all previously entered values unchanged.

- **Input Validation**

  Struts provide built-in capability for validation of form values.

1.5.2 Components of Struts1 Framework

Struts follow the MVC architecture. The major constituents of Struts framework are as follows:

- **Base Framework**

  Struts base framework provides the MVC functionality. Model component comprises Java data object or Enterprise JavaBean (EJB), whereas View component comprises JSP's and custom tags. Struts view layer uses ActionForm objects allowing exchange of data between user and the business layer.

The Struts `ActionServlet` class implements Controller Servlet which has the request and response component. Extension to Controller component is Struts `Action` class which decouples the client request from the business model.

- **JSP Tag Libraries**

  JSP tag libraries helps in programming the view logic in JSP. It allows JSP authors to use html like tags for representing the functionality that is defined by Java classes. Some of the JSP tag libraries in JSP include HTML tags, Bean tags, and JavaServer Pages Standard Tag Library (JSTL).

- **Tiles Plugin**

  Struts framework has Tiles sub-framework. Tiles are a JSP framework which allows the reuse of presentation code which is the html code. With this plugin, the JSP pages can be broken into pieces and then, joined to form a single page.

- **Validator Plugin**

  Struts framework also has Validator sub-framework. It facilitates validation of data on server as well as on the client-side. An XML file contains each of the validation rules which help to add and remove validations as it is not hard-coded into the file.

Figure 1.6 shows the components of Struts 1 framework.

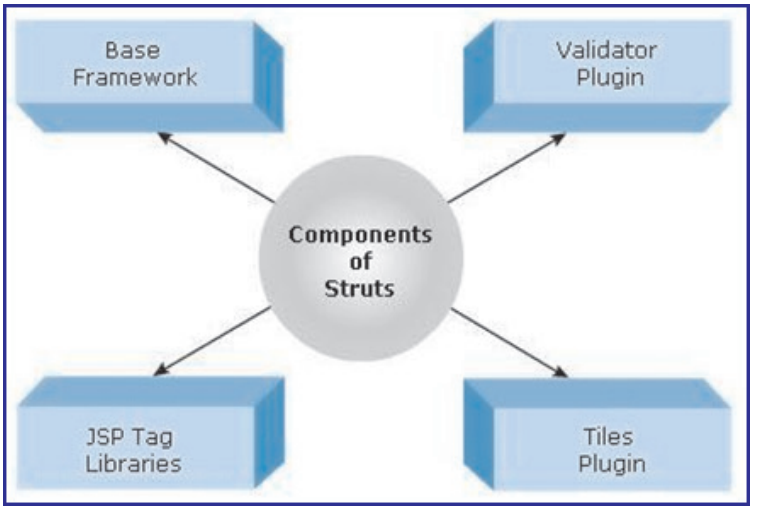


Figure 1.6: Components of Struts Framework

1.6 Struts 1 Architecture

Struts 1 framework defines all the three MVC components, they are as follows:

- **Model Component**

  This component provides the interface to data and/or methods. Thus, it eliminates the need for Controller component to add unnecessary code for accessing or manipulating data. Rather, the Controller just communicates with the Model component that has the data access and manipulation logic.

- **View Component**

  This component is a HTML Form data and also the JSP page which interacts with model data to generate response to the browser. In other words, the View component provides data that the user is going to see.

- **Controller Component**

  This component is a Servlet that receives requests from the browser and control the flow of data among the model and view. In Struts, Controller component is in the form of `ActionServlet` which executes `Action` object. `Action` object interacts with the model and prepares the data for view.

Figure 1.7 shows the Struts 1 MVC components.

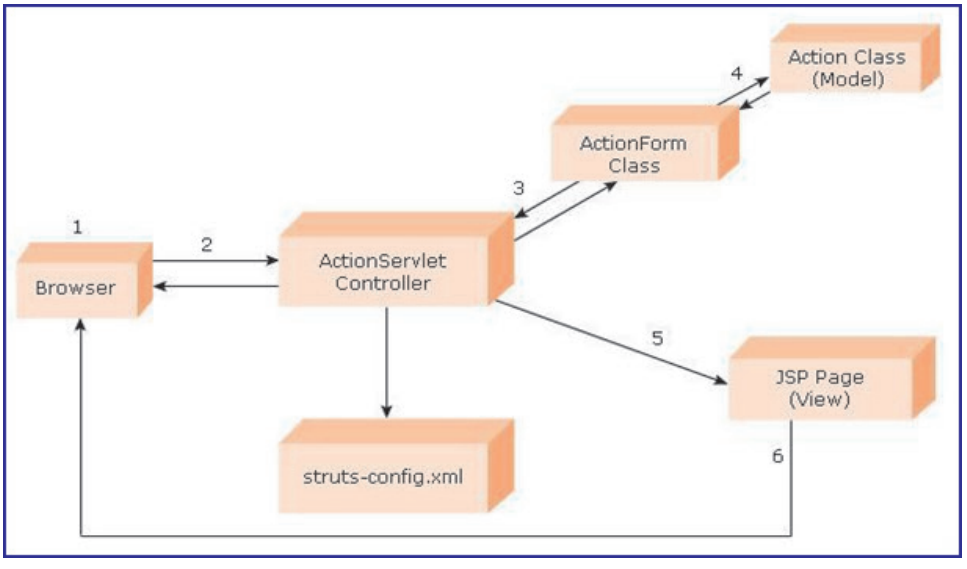


Figure 1.7: Struts 1 MVC Components

1.6.1 Struts Model Component

Struts architecture does not provide any specific constraints for the Model component. The application developer may use technology such as Enterprise Java Bean (EJB), Java Data Object (JDO), or the Data Access Objects (DAO) pattern to implement the Model component.

The model layer may be broken down into three sublayers. Each of these three sublayers need not get implemented as separate classes. However, the layers depict the breakdown from the functional aspect.

External interface sublayer provides an interface for external code to interact with your application. Business Logic sublayer provides the business rules and Data Access layer will communicate with the data base.

For example, in a Struts application that manages customer information, it may be appropriate to have a 'Customer' Model component as Java object that provides a program to access the information about customers. Yet another Model component may be 'preferred product' that populates products of customer's choice.

Figure 1.8 depicts the Struts Model Components.

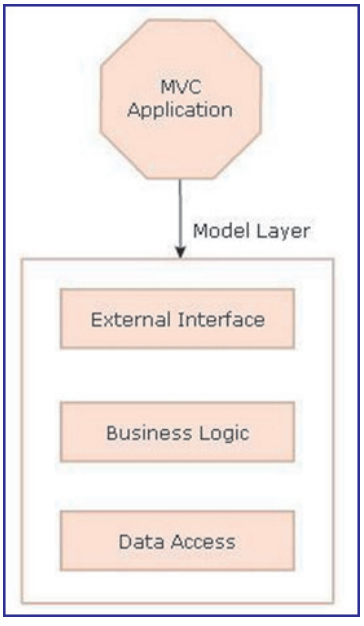


Figure 1.8: Struts Model Components

1.6.2 Struts View Component

For developing the View component in a Web application, Struts provides a rich set of features and functionality. The View layer of a Struts application can have several forms such as HTML/JSP, XML/XSLT depending on the requirement of the application. HTML/JSP is the most widely used presentation technology for developing Web applications.

Struts HTML/JSP View component has the following sub components:

- **JSP Pages**

  They contain static HTML and JSP library tags to generate dynamic HTML page. The JSP contains code for user interaction. The static and the dynamic HTML generated page is sent to the browser. Struts provide JSP tag library which allows creating HTML forms for capturing data and also displaying the data.

- **Form Beans**

  Form beans provide a channel for data transfer between the View and Control layers of Struts application. When a Struts application receives an HTML form, it takes the data from the incoming form and populates the corresponding FormBean. The Controller accesses the data using the getter method of FormBean.

- **JSP Tag Libraries**

  Struts have its own JSP tag libraries which help in development of JSP pages. These custom tags libraries provide a means to create HTML forms whose data will be captured in Form Beans and display the data stored in Form Beans. There are utility tags which help to implement conditional logic, iteration, and many more.

- **Resource Bundles**

  These allow internationalization of Java application by having application content placed into central repository called bundles. Thus, the content is not hard coded in the application and is read from the bundle by the application during execution. The advantage of using resource bundles for storing the application content is that it can be changed without recompiling the application.

Figure 1.9 depicts the Struts View components.

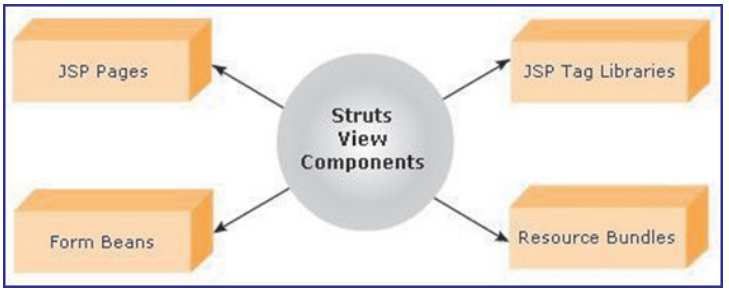


Figure 1.9: Struts View Components

1.6.3 Struts Controller Component

The Controller is an important component of Struts framework. Struts greatly contribute to the MVC design pattern controller component.

The core of Struts framework is Struts Controller Servlet called `ActionServlet`. The `ActionServlet` class handles run-time events in accordance with a set of rules that are provided at deployment time. These rules are contained in a Struts configuration file and specify how the Servlet responds to every event. Changes to the flow of control require changes only in the configuration file.

`ActionServlet` delegates its processing of a request to a `RequestProcessor` class. This `RequestProcessor` class selects the Form Bean associated with the request, populates the Form Bean with relevant data, performs validation, and then selects the respective `Action` class to execute the request. The Struts framework ends with the `Action` class and the application code starts from here.

Figure 1.10 shows the Struts Controller components.

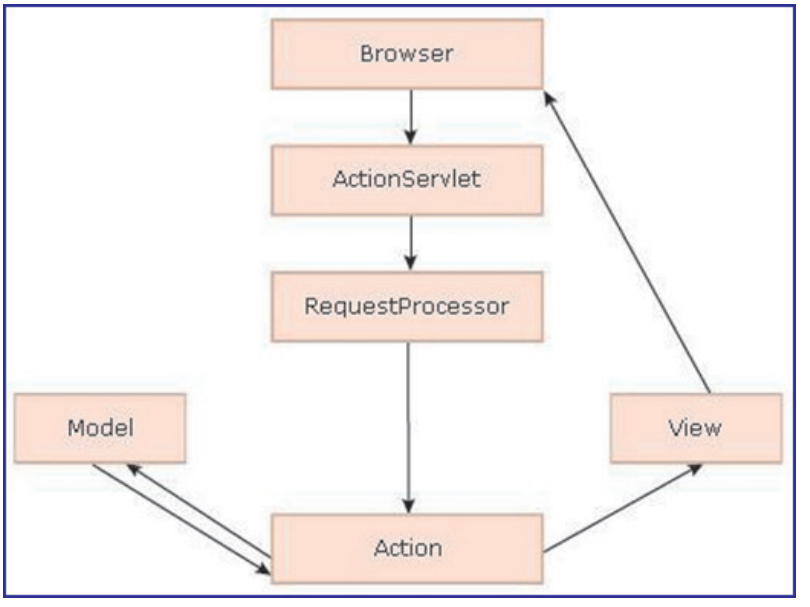


Figure 1.10: Struts Controller Components

Struts follow same control flow as that of the MVC architecture.

The control flow is as follows:

- Initially, the user sends a request to the Controller Servlet through a View.
- The Controller Servlet looks up the requested URI in the XML configuration file and determines the name of the `Action` class that will process the business logic.
- The `Action` class acts on the Model component as per the application's logic.
- On completion of processing the request, the `Action` class returns the results to the `ActionServlet` class.
- Based on the results provided, `ActionServlet class` decides which View to be forwarded with these results.
- The selected View displays the result thus, completing the request–response cycle.

Figure 1.11 shows the Struts 1 application control flow.

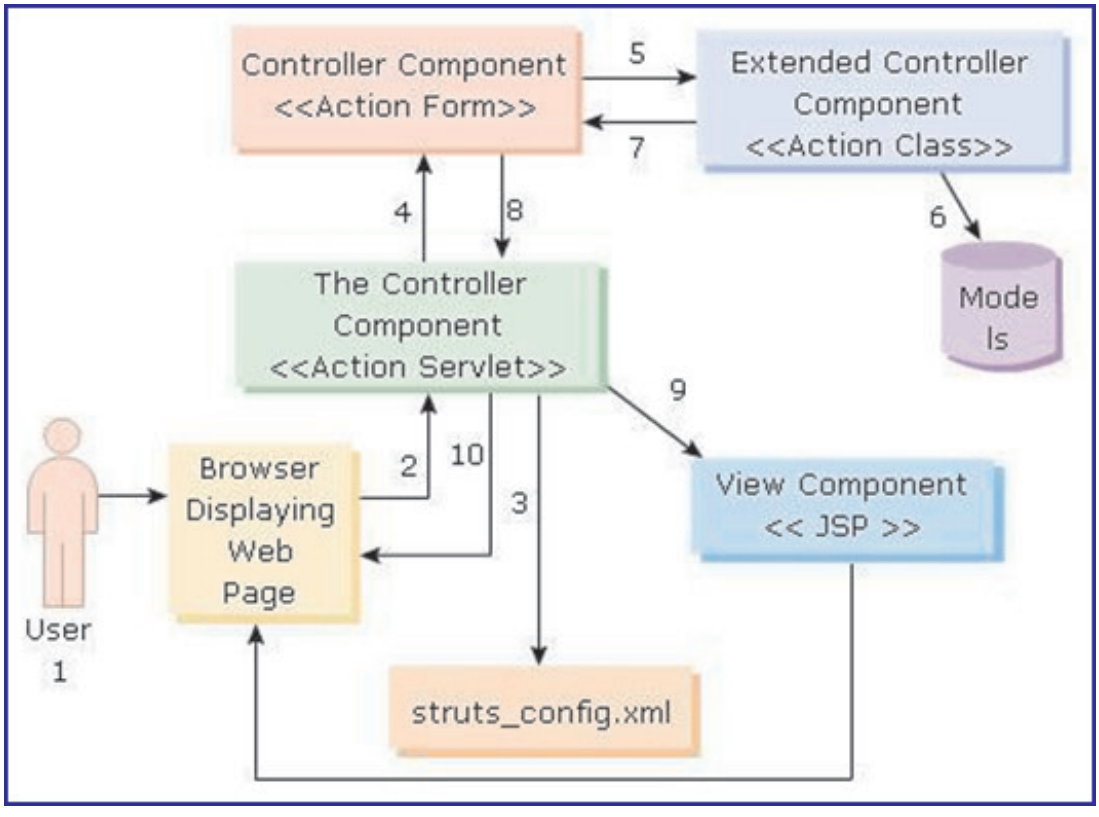


Figure 1.11: Struts 1 Application Control Flow

1.6.4 struts-config.xml File

The `struts-config.xml` configuration file is a link between the View and Model components in the Web. Some of the elements of the `struts-config.xml` file are as follows:

- **struts-config**

  This is the root node of the configuration file.
- **form-beans**

  This is where you map your ActionForm subclass to a name. The form name is used as alias for the other elements in the `struts-config.xml` file.
- **global forwards**

  This section maps a page on your Web application to a name. You can use this name to refer to the actual page.
- **action-mappings**

  This is where you declare form handlers and they are also known as action mappings.
- **controller**

  This section configures Struts controller servlet.

- **plug-in**

  This section tells Struts where to find your properties files, which contain prompts and error messages.

1.7 Developing Struts 1 Web Application

The `org.apache.struts.action.ActionServlet` class is the core class of all Struts based application. In Struts based Web applications, the purpose of the `ActionServlet` class is to receive all incoming HTTP requests for the application and determine which Action class will process the incoming request.

The `ActionServlet` class is not an abstract class and can be used for assembling simple application. However, a specialized `ActionServlet` class may be created for specific application. `ActionServlet` class is derived from `javax.Servlet.http.HttpServlet` class and implements the life cycle methods of the Servlet class such as `doGet()`, `doPost()`, and so on.

`ActionServlet` class is responsible for initializing the Struts framework for a Web application and receiving all the requests. It is a concrete class and may be used as it is in an application or can be extended. Every Struts based Web application will have a deployment descriptor named `web.xml`. A `web.xml` file is an XML file with several elements. It is this file in which the `ActionServlet` class is configured.

Code Snippet 1 shows how to configure the Servlet name and class in the `web.xml` file.

Code Snippet 1:

```
<web-app>
<!--ActionServlet Configuration -->
<Servlet>
<Servlet-name>actionExample</Servlet-name>
<Servlet-class>org.apache.struts.action.ActionServlet</Servlet-class>
</Servlet>
<Servlet-mapping>
<Servlet-name>actionExample</Servlet-name>
<url-pattern>*.do</url-pattern>
</Servlet-mapping>
</web-app>
```

In Code Snippet 1, the Servlet name is configured as '**actionExample**' and Servlet class is the default `ActionServlet` class.

In `<Servlet-mapping>` description, the tag `<url-pattern>` describes a pattern to resolve URLs. A portion of HTTP request is compared to the `<url-pattern>`. If the pattern matches, then this means that the `ActionServlet` named '**actionExample**' should service all the requests having an extension of `.do`.

1.7.1 Developing Action

The `Action` class is present in `org.apache.struts.action` package.

The `Action` class acts as a bridge between the user's request and the business operation to be performed. It typically performs a single business operation on behalf of the user. Thus, it is responsible for processing a specific HTTP request and generating a corresponding HTTP response.

The `Action` class does not contain business logic, rather it is designed to delegate business logic to the Model component.

It provides an interface to the application's Model layer; thus acting as a bridge between the View and Model layer. It transfers the data from the view layer to the specific business process layer and then, returns the processed data from business layer to the view layer. Every `action` is mapped to a path in the Struts configuration file. When a request is made to the `ActionServlet` class with a given path, the `action` for processing the request is invoked.

Figure 1.12 shows the class diagram of `Action` class.

Figure 1.12: Action Class

To use the `Action` class, developer has to subclass and overwrite the `execute()` method. The business logic of Web application begins with the execute() method. It is same as the service() method of the Servlet. The execute() method has two functions:

1. It performs the business logic for the application.
2. It helps framework to determine where it should next route the request.

When a user performs an action, the `execute()` method of the `Action` class is invoked to process the request based on user's action.

The `ActionServlet` class uses the execute() method to pass the parameterized class to the `ActionForm class`. The execute() method returns an object of type `ActionForward class`. Based on the value returned by `ActionForward class`, the RequestProcessor class determines where to forward the request such as a JSP or another `Action`. If the response has already been completed, then the method simply returns null.

The `execute()` method may throw any business logic exceptions.

Syntax:

```
public ActionForward execute(ActionMapping map, ActionForm form,
javax.Servlet.ServletRequest req, javax.Servlet.ServletResponse
resp) throws java.lang.Exception
```

where,

- map: represents an object of `ActionMapping` consists of the deployment information for an `action` bean.
- form: refers to an object of `ActionForm` class.
- req: refers to the HTTP request object.
- resp: refers to the HTTP response object.
- Exception: raises an exception only when the business code throws an exception.

The Struts Framework defines two implementations for the execute() method. The first execute() implementation is used to define custom `Actions` that are non-HTTP specific. This version is to be overridden in order to service request that are not HTTP specific. The second `execute()` implementation is used to define custom `Actions` that are HTTP specific. This version is to be overridden in order to service HTTP specific requests.

The parameters of `Action.execute()` method are shown in table 1.1.

Parameter	Description
`ActionMapping`	It contains all of the deployment information for a particular `Action` bean.
`ActionForm`	It represents the Form inputs containing the request parameters from the View referencing this `Action` bean.
`ServletRequest or HttpServletRequest`	It is a reference to current request object either Http specific or non-Http specific respectively.
`ServletResponse or HttpServletResponse`	It is a reference to current response object either Http specific or non-Http specific respectively.

Table 1.1: Parameters of Action.execute() Method

To derive an `Action` class, following steps are to be executed:

1. Create a class that extends the `Action` class of org.apache.struts.`action` package.
2. Override the execute() method in order to specify the desired business logic. The business logic is implemented in this method. It returns the `ActionForward` object which is used to route the request to the specified view.
3. To describe the new `action`, add an `<action>` element in the Struts configuration file.

Code Snippet 2 demonstrates the development of Login `Action` class for login Web application.

Code Snippet 2:

```
public class LoginAction extends org.apache.struts.action.Action {
 /* forward name="success" path="" */
 private final static String SUCCESS = "success";
 private final static String FAILURE = "failure";
 /**
  * This is the action called from the Struts framework.
  * @param mapping The ActionMapping used to select this instance.
  * @param form The optional ActionForm bean for this request.
  * @param request The HTTP Request we are processing.
  * @param response The HTTP Response we are processing.
  * @throws java.lang.Exception
  * @return
  */

public ActionForward execute(ActionMapping mapping, ActionForm
form,
   HttpServletRequest request, HttpServletResponse response)
   throws Exception {
  LoginForm loginForm = (LoginForm) form;
```

```
  if (loginForm.getUserName().equals(loginForm.getPassword())) {
  return mapping.findForward(SUCCESS);

 } else {
  return mapping.findForward(FAILURE);
 }
}
}
```

1.7.2 ActionForward Class

An `ActionForward` class of Struts Framework represents a destination where the controller (RequestProcessor) can process the forward request. The execute() method of the `Action` class when executed, returns an object of `ActionForward` class. This object has been mapped to the name of the forward from Struts configuration file. Rather than hard-coding URLs inside the application, a developer can use forward to define logical names for URLs and then, use these logical names to reference the URLs. Referencing URLs by logical names gives the benefit of shielding. When URL changes, there is no need to update each reference to URL. One change to forward definition is sufficient to reflect all places in application.

The properties supported by an `ActionForward` class are shown in table 1.2.

Property	Description
contextRelative	It specifies if URL value for path represents the absolute path or path is relative to the module under consideration.
Name	It is the logical name by which the instance may be looked up in relationship to a particular ActionMapping.
Path	It has the value that interpreted as Module-relative or Context-relative URL to which the control should be forwarded, or an absolute or relative URL to which control should be redirected.
Redirect	It is set to true if the Controller Servlet should call HttpServletResponse.sendRedirect() on the associated path; otherwise should be set to false.

Table 1.2: Properties Supported by an ActionForward Class

Code Snippet 3 declares the configuration of `action` along with the forward in the Struts configuration file.

Code Snippet 3:

```
<action-mappings>
<action input="/login.jsp" name="LoginForm" path="/Login"
scope="session"
 type="com.example.LoginAction">
<forward name="success" path="/success.jsp" />
<forward name="failure" path="/failure.jsp" />
</action>
</action-mappings>
</struts-config>
```

The code in the code snippet defines a URL to be used as forward with local scope. Here, the `action` will be forwarded to the **success.jsp** page, whenever the forward named 'Success' is found.

1.7.3 Developing Model

Model component of Struts framework represents application's business data and the rules that specify how to access and modify it.

As the Struts framework is for Web application development, the purpose of having a separate Model component is to ensure that the model remains independent of the client that is accessing it and thus is reusable. It is very essential for any application to ensure data integrity within the model.

Struts framework does not offer any special feature in order to build the Model component. Instead, users of this framework are free to use many other frameworks and component models available for developing business domain. EJB, JavaBeans, and Java Data Objects are some of the component models that can be used to create Struts Model component.

In a Struts-based application, the Model components are accessed from the subclass of Struts `Action` class that is part of the Struts Controller layer.

The `Action` sub class via its `Action` interface uses its Data Transfer Object to pass and retrieve data from model. It is necessary that no business logic and data access code be placed in `Action` object. Also business specific model code should not refer to Struts code or object. Violation of this would defeat the whole purpose of MVC architecture.

Figure 1.13 depicts the Struts model.

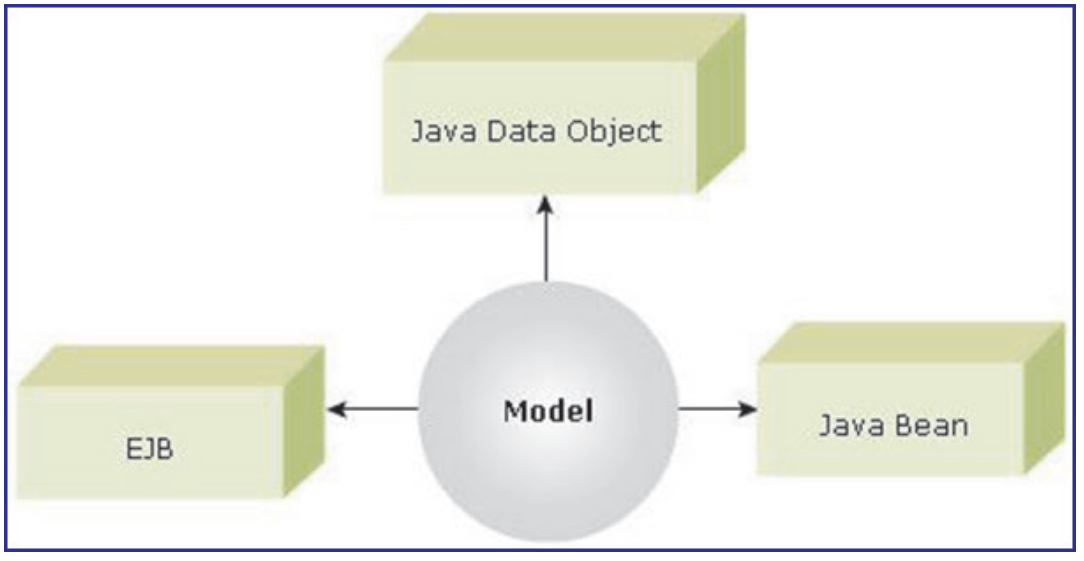


Figure 1.13: Struts Model

1.7.4 Developing ActionForm

All Web applications accept inputs from a user. When a Web application accepts data from user, it must be validated before sending it to the model layer.

Struts framework supports the functionality for retrieving the data entered by Web application user (that is from View layer), storing it temporarily for validation and displaying an error messages for invalid data or sending valid data to `Action` class (that is to Controller layer).

Also, when the model layer returns data for display, the `ActionForm` class is populated. This is then used by JSP page to provide the input fields for an HTML form. This makes the HTML form, a stable entity, as it takes data from the `ActionForm` class and not from Java Beans.

Figure 1.14 shows the class diagram of `ActionForm` class.

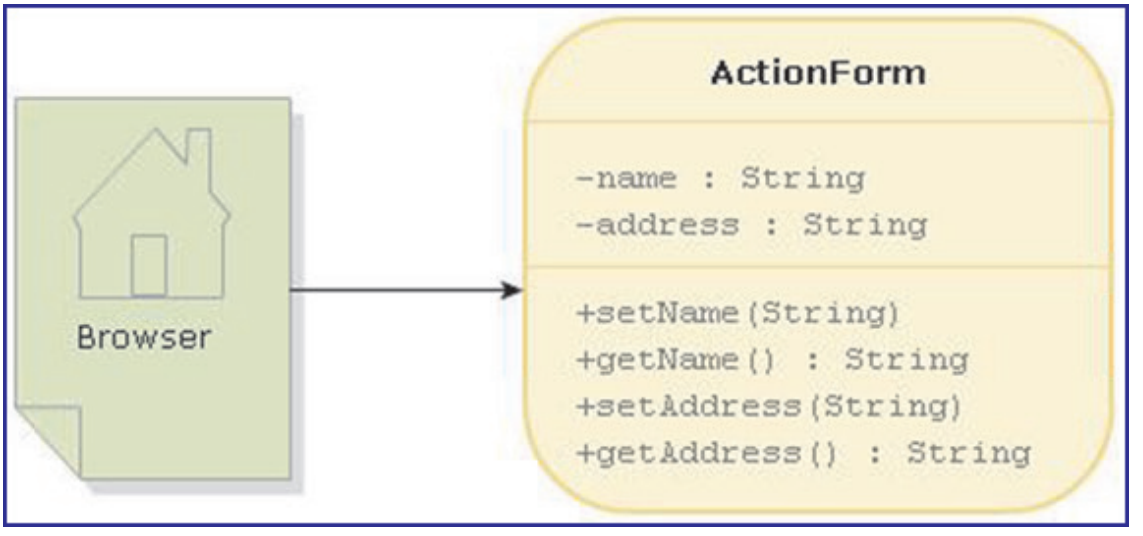


Figure 1.14: ActionForm Class

The `org.apache.struts.action.ActionForm` is the abstract base class supporting this functionality. Actual implementation is in concrete `ActionForm` classes which are Form Beans. Form Beans are basic Java beans with 'get' and 'set' methods defined for each of their properties and are used for supporting the mentioned functionality.

A subclass of this class is created to capture the application specific form data as well as to display the required data on the presentation page.

The abstract `ActionForm` class has two methods which a subclass may override for customized `ActionForm` class. They are as follows:

- `'reset()'`

  The reset() method is used to reset the `ActionForm` attributes to whatever state the application wants. For example, setting the String values to some pre-initialized value or to null. The reset() method is called for each new request, before the `ActionForm` is populated from the request parameters.

- `'validate()'`

  The validate() method is called by the RequestProcessor class for every request for performing data validation on `ActionForm` attributes before data is sent to `Action` class.

Figure 1.15 shows the LoginForm class extending from `ActionForm`.

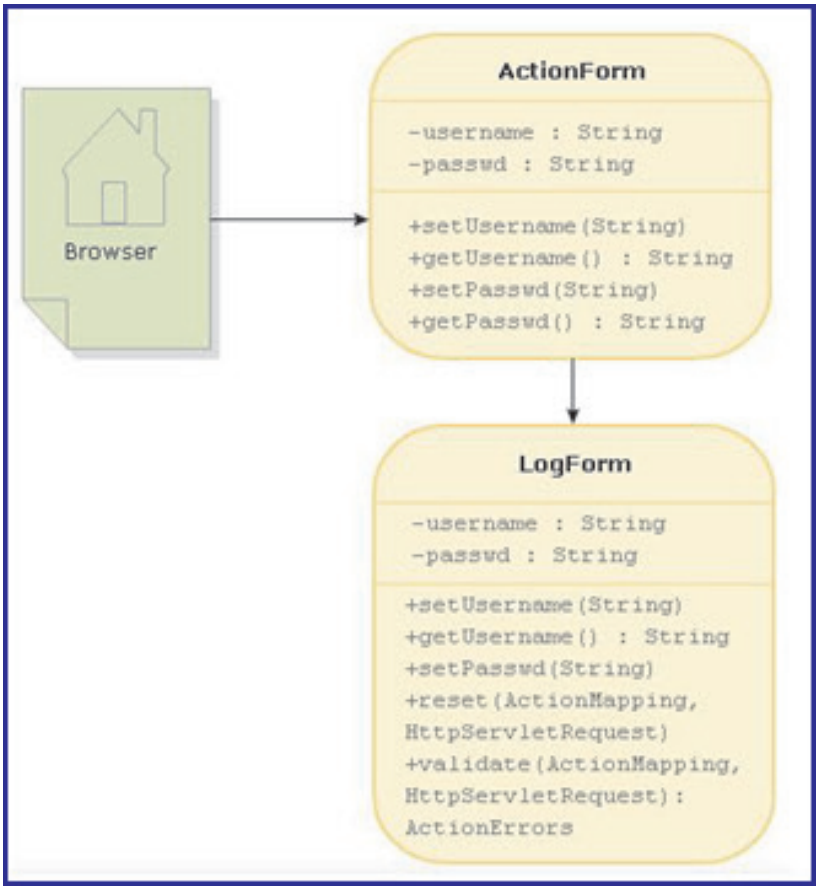


Figure 1.15: Deriving ActionForm Class

Code Snippet 4 demonstrates the development of `LoginForm` class.

Code Snippet 4:

```
public class LoginForm extends org.apache.struts.action.ActionForm
{
 private String userName;
 private String password;
 /**
 *
 */
 public LoginForm() {
```

```
 super();
 // TODO Auto-generated constructor stub
 }

/**
 * This is the action called from the Struts framework.
 * @param mapping The ActionMapping used to select this instance.
 * @param request The HTTP Request we are processing.
 * @return
 */
 public ActionErrors validate(ActionMapping mapping,
HttpServletRequest request) {
 ActionErrors errors = new ActionErrors();
 if (userName == null || userName.length() < 1) {
  errors.add("userName", new ActionMessage("error.userName.
required"));
  // TODO: add 'error.name.required' key to your resources
 }
 if (password == null || password.length() < 1) {
  errors.add("password", new ActionMessage("error.password.
required"));
  // TODO: add 'error.name.required' key to your resources
 }
 return errors;
}

 /**
 * @return the userName
 */
```

```
 System.out.println("Inside getter "+userName);
 return userName;
}

/**
 * @param userName the userName to set
 */
public void setUserName(String userName) {
 System.out.println("Inside setter "+userName);
 this.userName = userName;
}

/**
 * @return the password
 */
public String getPassword() {
 return password;
}

/**
 * @param password the password to set
 */
public void setPassword(String password) {
 this.password = password;
}
}
```

This code snippet creates a LoginForm class which is extended from `ActionForm` class. The LoginForm class declares two properties - the userName and password corresponding to the form fields. A getter and setter method is specified for both the attributes. Next, the validate() method checks if the userName and password have been assigned value or not. It returns an error if these attributes have null value.

1.7.5 JSP Tag Libraries

Struts come packaged with a set of its own custom JSP tag libraries that aids in the development of JSPs. The tag libraries are fundamental building blocks in Struts applications because they provide a convenient mechanism for creating HTML forms whose data will be captured in Form Beans and for displaying data stored in Form Beans.

Additionally, Struts' tag libraries provide several utility tags to accomplish things such as conditional logic, iterating over collections, and so on. With the advent of the JSP Standard Tag Library (JSTL), many of the utility tags have been superseded.

Following is the list of the Struts tag libraries and their purpose:

- **HTML** - Used to generate HTML forms that interact with the Struts APIs. Instead of using HTML tags to create forms, you can use their corresponding HTML Tag Library tags to create forms that tie into the

  Struts framework. The tags in this library can automatically populate form controls with data from Form Beans, thereby saving your time and several lines of code. For example, tags such as button, frame, image, and so on.
- **Bean** - Used to work with Java bean objects in JSPs, such as to access bean values. The Bean Tag Library is a collection of utility tags that provides convenient access for interacting with Web application objects within a JSP. Most of the tags are used to capture references to specific objects and store them in JSP scripting variables so that other tags can access the objects. The remaining tags are used to render objects to the JSP output. For example, tags such as cookie, include, message, and so on.
- **Logic** - Used to cleanly implement simple conditional logic in JSPs. The Logic Tag Library provides a rich set of tags for cleanly implementing simple conditional logic in JSPs. With these tags, you can wrap content that will be processed only when a particular condition is true.

Table 1.3 shows the logic tags.

Tag	Description
present	It is used to test whether a cookie, a request parameter, a request header or a JavaBean's property is present, and executes tag body if it is present.
notPresent	It is used to test whether a cookie, a request parameter, a request header or a JavaBean's property is present, and executes tag body if it is not present.
Equal	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is equal to the specified value, and executes tag body if it is present.
notEqual	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is equal to the specified value, and executes tag body if it is not present.

Tag	Description
greaterThan	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is greater than the specified value, and executes tag body if it is present.
lessThan	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is less than the specified value, and executes tag body if it is present.
greaterEqual	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is greater than or equal to the specified value, and executes tag body if it is present.
lessEqual	It is used to test whether the value of a cookie, a request parameter, a request header or a JavaBean's property is less than or equal to the specified value, and executes tag body if it is present.

Table 1.3: Logic Tags

- **Nested** - Used to allow arbitrary levels of nesting of the HTML, Bean, and Logic tags.

Code Snippet 5 shows the `login.jsp`. The form has one text field to get the user name and one password field to get the password. The form also has one Submit button, which when clicked calls the login `action`. <html:errors/> tag is used to display the error messages to the user.

Code Snippet 5:

```
<%@taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html>
<head>
<title>Login Page</title>
</head>
<body>
<div style="color:red">
<html:errors />
</div>
<html:form action="/Login" >
```

```
User Name :<html:text name="LoginForm" property="userName" />

Password :<html:password name="LoginForm"property="password"
/>

<html:submit value="Login" />

</html:form>

</body>

</html>
```

Here, the action is `/Login`, the input page is login.jsp and the corresponding action class is `LoginAction.java`.

Code Snippet 6 shows the `success.jsp` and `failure.jsp` pages.

Code Snippet 6:

```
<!—success.jsp -->

<html>

 <head>

  <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">

  <title>JSP Page</title>

 </head>

 <body>

  <h1>Login Success. Welcome <bean:write name="LoginForm"
property="userName"></bean:write></h1>

 </body>

</html>
```

```
<!—failure.jsp -->

<html>

 <head>

  <meta http-equiv=”Content-Type” content=”text/html;
charset=UTF-8”>

  <title>JSP Page</title>

 </head>

 <body>

  <div style=”color:red”>


<h1>Invalid user name <bean:write name=”LoginForm”
property=”userName”></bean:write></h1>

  </div>

 </body>

</html>
```

Check Your Progress

1. Which of the following is not component of MVC architecture?

(A)	Controller	(C)	View
(B)	Model	(D)	Mapping XML file

2. Which of the following statements is/are true about the role of framework?

(A)	They propagate code reuse over design reuse.
(B)	They can be written in any programming language such as Java and C++.
(C)	They only allow faster building of application.
(D)	They are not created and used for a particular application domain.

(A)	A and B	(C)	B and C
(B)	A and D	(D)	C and D

3. Match the component with its associated description.

Component		Description	
(A)	Controller	(1)	An .xml file used for configuring Servlet and JSP.
(B)	web.xml	(2)	Handles presentation.
(C)	View	(3)	Handles business logic.
(D)	Model	(4)	Handles request and response.

Check Your Progress

(A)	A-3, B-1, C-4, D-2	(C)	A-2, B-1, C-4, D-3
(B)	A-4, B-1, C-2, D-3	(D)	A-4, B-3, C-2, D-2

4. Which of the following are correct Struts View components?

(A)	JSP Tag Libraries
(B)	Resource Bundles
(C)	JSP Tags
(D)	Configuration File

(A)	A, B, and C	(C)	Only C
(B)	A, C, and D	(D)	B, C, and D

Check Your Progress

5. Which of the following code correctly declares a derived `ActionForm` class?

(A)	`public class LogForm extends ActionForm {`
	`private String username = null;`
	`public void setUsername(String username){`
	`this.username = username;`
	`}`
	`public String getUsername(){`
	`return(this.username);`
	`}`

(B)	`public class LogForm extends ActionForm {`
	`private String username = null;`
	`public String getUsername(){`
	`return(this.username);`
	`}`
	`}`

(C)	`public class LogForm extends ActionForm {`
	`private String username = null;`
	`public void setUsername(String username){`
	`this.username = username;`
	`}`
	`}`

(D)	`public class LogForm extends ActionForm {`
	`public void setUsername(String username){`
	`this.username = username;`
	`}`
	`public String getUsername(){`
	`return(this.username);`
	`}`
	`}`

Answer

1.	D
2.	C
3.	B
4.	A
5.	A

Summary

- The MVC is an architectural pattern whose main concern is to separate data from the user interface.
- In Model 1 architecture, the JSP page is not only responsible for processing requests and sending back replies to the clients, but also for extracting the HTTP request parameters, invoking the business logic, and handling the HTTP session.
- Model 2 architecture is an approach used for developing a Web application. It separates the 'Business Logic' from the 'Presentation Logic'. Besides this, Model 2 has an additional component - a Controller.
- The responsibility of the Controller Servlet is to process the incoming request and instantiate a Model - a Java object or a bean to compute the business logic.
- A framework provides a structural support in the form of classes that can be extended for reuse and a functional toolkit in the form of software libraries.
- Struts provides the foundation along with libraries and utilities to develop MVC based applications easily and faster.
- The major constituents of Struts framework are: base framework, JSP tag libraries, tiles plugin, and validator plugin.
- In Struts, Controller component is in the form of ActionServlet which executes Action object. Action object interacts with the Model and prepares the data for View.
- Struts come packaged with a set of its own custom JSP tag libraries that aids in the development of JSPs.

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions *in your* mind?

are here to HELP

Post your queries in ASK to LEARN @

www.onlinevarsity.com

02

Working with Struts 2 Framework

Welcome to the Session, **Working with Struts 2 Framework**.

The session will provide an insight to Apache Struts 2 framework and its features. It explains the architecture and request life cycle of Struts 2 framework. Further, the session explains the working of Struts 2 components such as actions and result Views. It also explains how to configure Struts 2 actions in the configuration file. Finally, the session concludes with the introduction to annotations in Struts 2 framework.

In this Session, you will learn to:

- Explain the features of Struts 2 framework
- Explain the architecture of Struts 2 framework
- Explain the components and request life cycle of Struts 2 framework
- Explain the difference between Struts 1 and Struts 2 framework
- Explain how to implement Struts 2 actions using Action interface and ActionSupport class
- Explain the process of managing data from forms to JavaBean properties
- Explain how to configure Struts 2 components in the configuration file
- Explain Result and Result Types
- Explain the annotations provided by Struts 2 framework
- Explain how to develop a Java Web application using Struts 2 framework

Session
02

Working with Struts 2 Framework

2.1 Introduction

Apache Struts is a Java-based framework that separates the business logic from the presentation layer. Struts helps in easy development of Web applications by providing flexible and extensible application architecture. It is based on MVC design paradigms which separate the application development into three levels.

They are as follows:

- **Model** – Provides the application logic and handles database interactions.
- **View** – Presentation of data through HTML and JSP pages.
- **Controller** – Routes the application flow and information between View and Model.

The first version of Struts 1 provided the several components that make up the control flow of the application. These components are namely, the controller, request handlers, and Action objects. In Struts 1, the standard controller is a Servlet. It is known as `ActionServlet` and is the primary component of the Struts 1 framework. Then, it defines a set of `ActionMappings` which provide match against the requested URL and specifies the class name of the mapped `Action` class. `Action` classes are the main component of Struts 1. They encapsulate business logic and maps the response into a model. Finally, the action returns the response which is rendered as a view using the JSP page. All the components in the application are glued together in the framework through the Struts configuration file known as `struts-config.xml`.

In this, the requested URL is mapped to a unit of work known as action. The task of action is to perform the functionality for the requested URL by accessing the HTTP session, HTTP request, and form parameters. It invokes the business logic and maps the response into a Model. Finally, the action returns the response which is rendered as a View using the JSP page.

The main limitation of Struts 1 is that it encourages a fix approach to MVC approach when designing Web applications. It provides a standard component template defined for the applications. Thus, resulting in a rigid approach.

The next generation of Apache Struts 2 came with the launching of Struts 2 which made the Web application development easier. It is the second generation of Web application framework based on WebWork framework created by OpenSymphony. After working independently for several years, OpenSymphony joined with Apache to create Struts 2 framework. Thus, Struts 2 is not just the next version of Struts 1, but it is a thorough revision of the Struts architecture containing features of WebWork and Struts 1.

Struts 2 = WebWork + Struts 1

Struts 2 is a flexible framework for creating enterprise-ready Java Web applications based on cleaner implementation of MVC. The framework is designed to simplify the entire development cycle, including building, deploying, and maintaining applications over time.

2.1.1 Features of Struts 2

Struts 2 is an advanced version of Struts framework. It provides many new features that were not in Struts 1.

Struts 2 achieved easy development of Web application by reducing XML configurations and introducing annotation-based configuration. In Struts 2, Action classes are Plain Old Java Objects (POJO) objects that are used for simplifying the development and testing of the code. Struts 2 also introduced the concept of Interceptors that reduces the coupling in the application.

Some of the important features of Struts 2 framework are as follows:

- **Simplified Design**

  Struts 2 framework classes are based on interfaces and most of its core interfaces are HTTP independent and framework neutral. This enables the users to test Struts application very easily without referring the low-level HTTP Request and HTTP Response objects in the application.

- **Configurable MVC Components**

  In Struts 2 framework, the component information is configured in `struts.xml` file. Therefore, any change in the information can be made by simply changing it in the xml file.

- **POJO-based Actions and Forms**

  Struts 2 Action classes are POJOs which are simple java class. The action class acts as a model in the Web application. They are not required to implement any interface or inherit any class. Therefore, testing of the code is highly simplified. Even, to capture user input from the form, developers can use JavaBeans instead of ActionForms provided in Struts 1.

- **Enhanced Tags**

  Struts 2 tags enable to add stylesheet-driven markup capabilities. Here, the user can create consistent pages with fewer codes. Markup tags can be changed by changing an underlying stylesheet in Struts 2. In addition, it is very easy for the developer to extend the tags and write one's own functionality. Struts 2 provide various types of tags such as Data tags, UI tags, Control tags, and so on which simplify the application development.

- **Object Graph Navigation Language (OGNL) and ValueStack**

  Struts 2 uses an elegant expression language called Object Graph Navigation Language (OGNL) which is more powerful and flexible than Java Standard Tag Library (JSTL) expression language used by Struts 1 framework. OGNL supports Asynchronous JavaScript and XML (AJAX) implementation in User Interface (UI). OGNL allows the developer to refer and manipulate the data present on the ValueStack. ValueStack is simply a storage area that holds application data associated with the processing of a request.

- **Easy Integration**

  Struts 2 is an extensible framework can be easily integrated with any other frameworks. A framework must provide an abstraction layer for integrating with other frameworks. The frameworks that can be integrated with the Struts 2 applications include Hibernate, Spring, Tiles, and so on.

- **Theme and Template Support**

  Struts 2 provides three types of themes: xhtml, simple, and css_xhtml. The default theme of struts 2 is xhtml. These themes and templates can be used for common look and feel across the application. Using these themes and templates, a user can change the overall look of a Website, simply by changing the theme files.

- **Annotation Support**

  Struts 2 uses annotations. The advantage of using annotations is that it eliminates the manual task of configuring the components of the application in the XML configuration file. Annotations allow the developers to add meta-data information into the Java source code declaring Actions. This meta-data information is translated into components during runtime.

- **Minimal Configurations**

  For the Struts 2 application, minimal configuration is required as most of the settings will take the default values. Configuration is required only if there are any modifications in the default settings.

- **AJAX Support**

  Struts 2 integrates AJAX support into the framework by creating AJAX tags.

- **View Technologies**

  Struts 2 provide a great support for presenting the results on the Web pages. The multiple view technologies include JSP, FreeMarker, Velocity, and XSLT.

2.2 Architecture of Struts 2

Action-based framework was introduced with the advent of Struts 1 Web application development. In action-based MVC framework, a Servlet act as a controller and provides a centralized point of control for client page requests. However, Struts 2 is based on a pull-MVC framework. It means that the data that is to be displayed is pulled from Action class, which acts as a model in the Web application.

Struts 2 also provides powerful APIs for configuring the Validators, Interceptors, and so on that reduces the processing on the action class.

2.2.1 Struts 2: MVC Pattern

Struts 2 follows the MVC design pattern which helps to define the coding pattern or style. The MVC pattern helps the developer to separate the different areas of functionality that applies to the domain of Web applications.

Figure 2.1 shows the core components of Struts 2 MVC pattern.

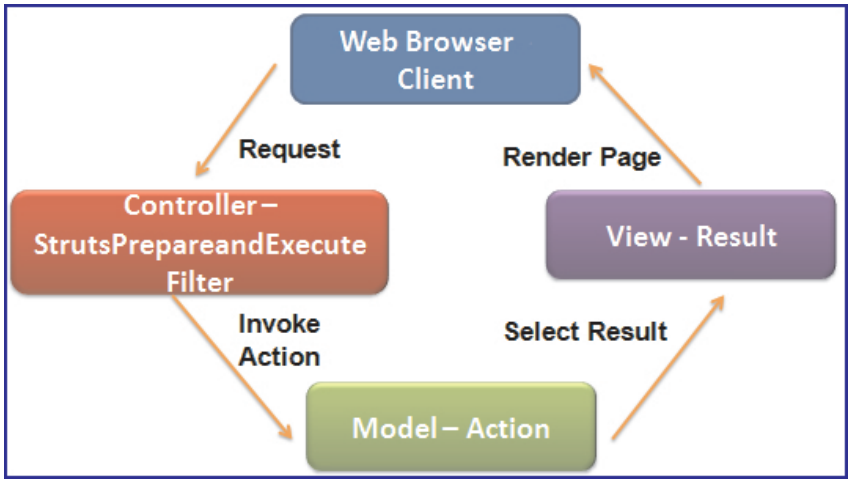


Figure 2.1: Struts 2 MVC Pattern – Core Components

- **Controller**

  The Controller controls all the processing done by the application. In other words, it manages the interaction between the View and the Controller. A Web application routes the request to appropriate set of actions present within the application. This mapping of request to appropriate set of actions is the main task of the Controller. Thus, it can be said that the controller is the first component that will process the HTTP request coming from the client.

  In Struts 2, `StrutsPrepareAndExecuteFilter` plays the role of a Controller. It is a Servlet filter that checks each incoming requests and determines the Struts 2 action to handle the request. The framework is required to be informed about the mapping between the request URL and the corresponding action. This is specified either through XML based configuration files or annotations.

- **Model**

  A Model is a business object and has no user interface. The model in MVC design pattern in Struts 2 framework is implemented by the action component. The model is the core of an application, as it contains both the data model and the business logic.

  Model can be defined as the internal state of an application. For example, in the login authentication process of an application, both the business logic and the data from the database are involved. The business logic will accept the data, that is, the username and password and verify it against the data stored in the database. Both this business logic and the data will combine and will result in either successful or unsuccessful authentication state. Neither the data nor the business logic alone can produce the result.

Thus, it can be said that Struts 2 action serves two roles, that is, it encapsulates the business logic and is the locus of data transfer.

A Web application can have a number of actions mapping to various requests. Thus, after receiving a request the Controller checks the mapping to determine the correct action that will handle the request.

Once the correct action is found, the Controller invokes the action and hands over control of the request processing to the action. Once the action class completes the request processing, it forwards the result to the Struts 2 View component.

- **View**

It is the presentation component of the MVC design pattern. View component returns the result page to the Web browser. The JSP page is used to commonly display the results. In other words, the view translates the state of the application to a visual presentation with which the user can interact. The action has to choose which result should be displayed. Usually, it is the outcome of the action processing such as success or error.

2.2.2 Struts 2 Framework

Figure 2.2 shows the high-level diagram of Struts 2 framework.

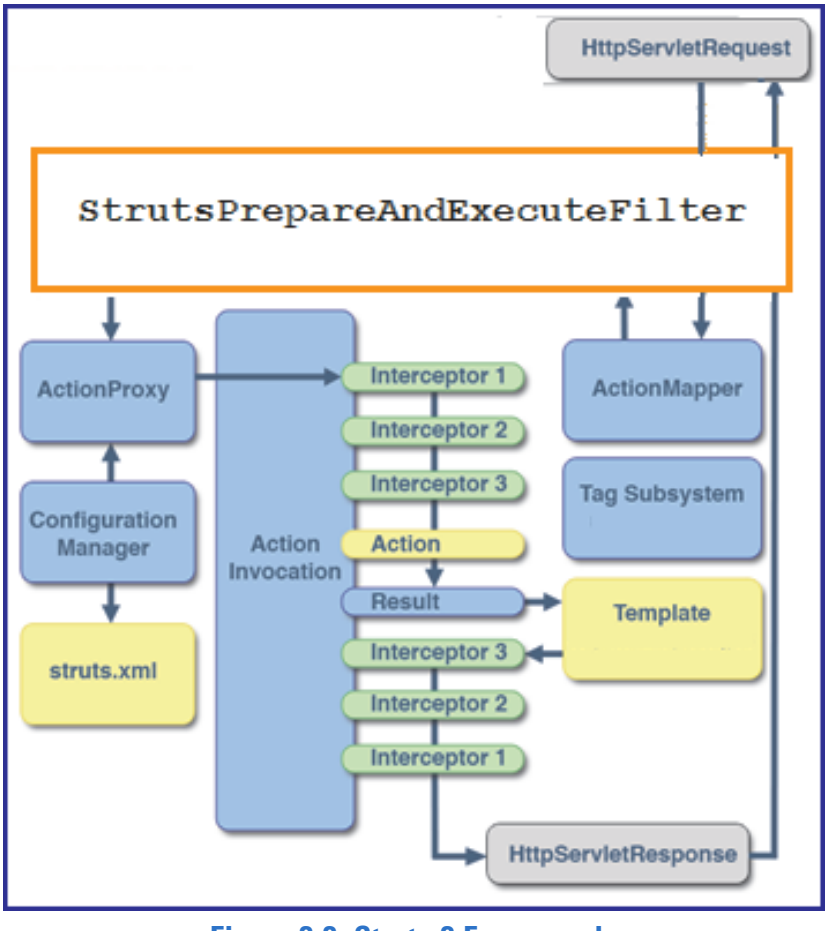


Figure 2.2: Struts 2 Framework

The various components of the Struts 2 framework are as follows:

- **Request Initiation (HttpServletRequest)**

  A request in a form of a URL initiates from and ends in a user's Web browser.

  A request initiated by a user is passed through a standard filter chain which makes the actual decision by invoking the required `StrutsPrepareAndExecuteFilter`.

- **StrutsPrepareAndExecuteFilter**

  From Struts 2.1 onwards, `org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter` plays a role of the Controller. This important object is a Servlet filter that inspects each incoming request to decide which action should handle the request.

  When called, the Controller consults the **ActionMapper** to determine whether the request should invoke an action. The framework handles all of the Controller work. A developer only need to inform the framework which request URLs maps to which actions. This information is passed to the framework using an XML-based configuration, `struts.xml` or Java-based annotations.

  It is important to note that earlier version of Struts framework, that is Struts 1.0 provided `FilterDispatcher` as a Controller.

- **Action Mapper**

  It is an interface which provides a mapping between HTTP requests and action invocation requests and vice-versa. When `HttpServletRequest` comes, the `ActionMapper` tries to match an appropriate action invocation request. If no action invocation request matches, it returns null and if the ActionMapper determines that an action should be invoked, the `StrutsPrepareAndExecuteFilter` delegates control to the **ActionProxy**.

- **Action Proxy**

  The ActionProxy reads the framework configuration files manager, such as the `struts.xml` file. It holds the configuration and context information to process the request and the execution results after the request has been processed. Then, the ActionProxy creates an instance of `ActionInvocation` class and delegates the control.

- **Action Invocation**

  `ActionInvocation` is responsible for command pattern implementation. In a request-response model, command pattern is used to execute lose coupling. The request is routed to the invoker and invoker sends it to the encapsulated command object. The request is then sent to the suitable method of the receiver to initiate proper action. The receiver object is then generated and attached to the command. Thereafter, the invoker object is formed and assigned to the command object to initiate an action.

The configured Interceptors are invoked one by one in sequence and then the Action is invoked. Once the Action returns, it looks for a proper result associated with the Action result code mapped in the `struts.xml` file.

Then, the result is executed involving the rendering of JSP or templates.

- **Interceptors**

Interceptors allow the developers to develop the code specification that can be executed before or after the execution of an action.

Using interceptors, the developer can perform the following:

- Provide pre-processing logic before invoking the action.
- Interact with the action and set the request parameter on the action.
- Provide post-processing logic after invoking the action.
- Modify the results.
- Catch exceptions to perform alternate processing or return different results.

Interceptors have access to the action being executed, the environment variables, and the execution properties. In Struts 2, the core functionalities such as handling exception, type conversion, file upload, page preparation, and so on are implemented using pluggable Interceptors. As pluggable Interceptors are being used, developers can choose the required ones depending on the action being developed.

Interceptors are invoked both before and after the execution of the action and the results are rendered back to the user. The Struts 2 framework first finds which `Action` class to invoke for this request and determines the interceptors related with the action mapping.

Then, an instance of `ActionInvocation` is generated and its `invoke()` method is called. `ActionInvocation` holds the information about the action and the associated interceptors. It knows the sequence in which the interceptors should be invoked. `ActionInvocation` now invokes the `intercept()` method of the first interceptor in the stack and so on. After the implementation of all the interceptors, the action class will be invoked. Then, a result string will be returned and the consequent view will be rendered.

At the end, all the applied Interceptors will be executed again in reverse order. Finally, the response will be returned back to the user through the filters configured in `web.xml` file.

2.2.3 Request-Response Cycle in Struts 2 Architecture

In Struts 2 Web application there are different components that interact with each other. The components represent the user who initiated the request, the environmental components, classes, and the elements of the Struts 2 framework.

Figure 2.3 describes the request-response life cycle of Struts 2.

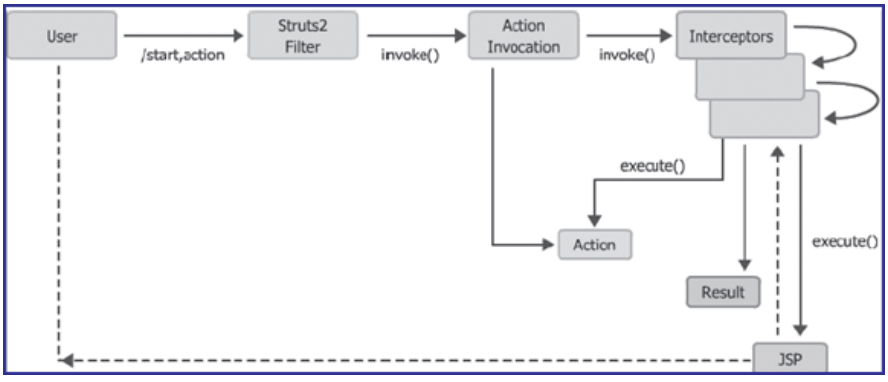


Figure 2.3: Struts 2 Request-Response Walkthrough

The steps taken for processing a request-response cycle are as follows:

- **Step 1: User sends the request**

  The request-response cycle starts and ends from the user's Web browser. A request is in the form of an URL. This request URL represents an Action.

  All requests for entire Web contexts are forwarded to a Servlet container on the Web server. When a request is received by the Servlet container, the container passes the request to the Servlet controller or the filter for processing. At the core of the Struts 2 framework are the filter classes which provide access to the infrastructure required for request processing.

  The request processing life cycle of Struts 2 follows the given order:

  - Displays static content through user interface provided by the View.
  - Determines the action configuration, that is, determines which action maps to the URL from the incoming request.
  - Creates the action context as it converts the request to a protocol independent format for the actions to use.
  - Creates the action proxy class containing the configuration and context information for processing of the request. It also contains the execution results after the requests have been processed.
  - Performs cleanup of the `ActionContext` object and ensures no memory leak.

- **Step 2: StrutsPrepareAndExecuteFilter determines the appropriate action**

  The `StrutsPrepareAndExecuteFilter` accepts the request and then consult `ActionMapper` to determine the suitable Action. If `ActionMapper` discovers an action to be invoked, then `StrutsPrepareAndExecuteFilter` delegates the control to `ActionProxy`.

For the processing to start, the `execute()` method on the action class is invoked. The method is invoked after the `ActionProxy` class is created and configured. `ActionProxy` class reads the `struts.xml` configuration file.

The `ActionProxy` class stores the configuration and context information for request processing and execution results after the request has been processed. `ActionProxy` class then creates an instance of the `ActionInvocation` class and delegates the control. The `ActionInvocation` object manages the execution environment and contains the conversational state of the action being processed. This class is the core of the `ActionProxy` class. Actions, interceptors, and results are the three environmental components.

- **Step 3: Interceptors are applied**

  `ActionInvocation` class implements the command pattern and invokes the interceptors as configured. Interceptors add processing logic depending on the method being invoked in the action class. The developer is required to write code around each and every action. The `ActionInvocation` class creates an instance of the action class.

  Struts 2 create a new instance for each and every request that is received. This instance behaves as POJO. The interceptors help to specify the 'request processing life cycle' for an action.

- **Step 4: Action is executed**

  Next, the action method is executed to carry out the database related operations like storing or retrieving data from the database. After processing of the request by the `execute()` method, a String result is returned which is mapped to an implementation of the `Result` interface. `ActionInvocation` class is responsible for looking up the results mapped in `struts.xml` configuration file.

- **Step 5: Result is displayed**

  The Interceptors are executed in reverse order and the response is sent to the Filter. The `Result` interface is similar to an action class and contains a single method that generates a response for the user. The result is sent to the Servlet container which sends the output to the user browser.

2.2.4 Configuring web.xml

The developer configures the Struts 2 Web application in `web.xml` file present in the `WEB-INF` folder of the application. Struts 2 Web configuration uses `<filters>` in place of `<Servlet>`. The URI pattern should be specified as `'/*'` so as to ensure that all kind of request patterns are served by `org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter`.

Figure 2.4 shows the implementation of the Controller Servlet filter in `web.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
. . .
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
org.apache.struts2.dispatcher.ng.filter.StrutsPrepare
                                    AndExecuteFilter
</filter-class>
</filter>
<filter-mapping>
          <filter-name>struts2</filter-name>
           <url-pattern>/*</url-pattern>
</filter-mapping>

<session-config>
      <session-timeout> 30 </session-timeout>
</session-config>

<welcome-file-list>
      <welcome-file>example/index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Figure 2.4: Configuring Controller in web.xml File

2.2.5 Differences between Struts 1 and Struts 2

Struts 1 framework and Struts 2 framework differ in many ways. Table 2.1 describes the basic differences between Struts 1 and Struts 2.

Feature	Struts 1	Struts 2
Controller	Struts 1 uses a Servlet Controller such as `ActionServlet` class.	Struts 2 uses a Filter to act as a Controller.
Form Input	In Struts 1 an HTML form mapped to an `ActionForm` object is used to capture inputs.	In Struts 2 Action properties are used for capturing input and thus eliminating the need for creating another class for obtaining input. The HTML form maps directly to a POJO class in Struts 2.
Model	In Struts 1 the `Action` interface is implemented. It acts as an adapter between the contents of an HTTP request and the corresponding business logic that executes to process the request.	Struts 2 action classes do not require to implement the `Action` interface. The `Action` object is a POJO object that can be easily tested.
Thread Model	In Struts 1, only one instance of the user-defined `Action` class is created for handling all incoming requests and so it is required to be thread-safe.	Struts 2 creates instance of user-defined `Action` class for each incoming request.

Feature	Struts 1	Struts 2
Tag Library	Struts 1 provides several tag libraries such as HTML, Bean, and Logic.	Struts 2 provide a single tag library.
Expression Language	Struts 1 provided Java Standard Tag Library (JSTL) Expression Language (EL).	Struts 2 uses OGNL as an expression language.
Type Conversion	Struts 1 uses JSP mechanism of binding object into the page contexts for access.	Struts 2 uses `ValueStack` for the taglibs to access values.
Control of Action Execution	Struts 1 has a separate life cycle for each of the module. All actions share same life cycle in the module.	Struts 2 create separate life cycle for each action using the Interceptor Stack.

Table 2.1: Differences between Struts 1 and Struts 2

2.3 Introduction to Struts 2 Action

The action classes are the core part of the Struts 2 framework, as they act as controller in the MVC pattern. The main functions of Struts 2 action classes includes responding to a user request, executing business logic, and then returning a result to the user based on configuration file, `struts.xml` to render the view page.

Struts 2 actions also serve in two other important capacities:

- Transferring the data from the request to the view, irrespective of the type of result.
- Assisting the framework to determine which result should render the view that will be returned in response to the request.

Struts 2 actions are not singleton, that is, an instance of action is created on each request. Therefore, unlike Struts 1 actions, Struts 2 actions are not required to be thread safe. Struts 2 classes are simple and independent of other API. Thus, testing of Struts 2 action classes is much easier.

2.3.1 Using Action Classes

As Struts 2 POJO-based actions, so it is not necessary for the developer to implement any interface or class. The only important point is the class must implement an `execute()` method. The code specification inside the `execute()` method should only hold the logic of the work associated with the request. The `execute()` method does not accept any parameters, but it returns a String value. The String value specifies the result page which has to be returned as a View to the client.

Code Snippet 1 shows how a POJO class that can be implemented as action in Struts 2.

Code Snippet 1:

```
public class MessageAction{
      //execute() method containing business logic
      public String execute() {
           return "success";
      }
}
```

Figure 2.5 displays the request processing mechanism of actions in Struts 2.

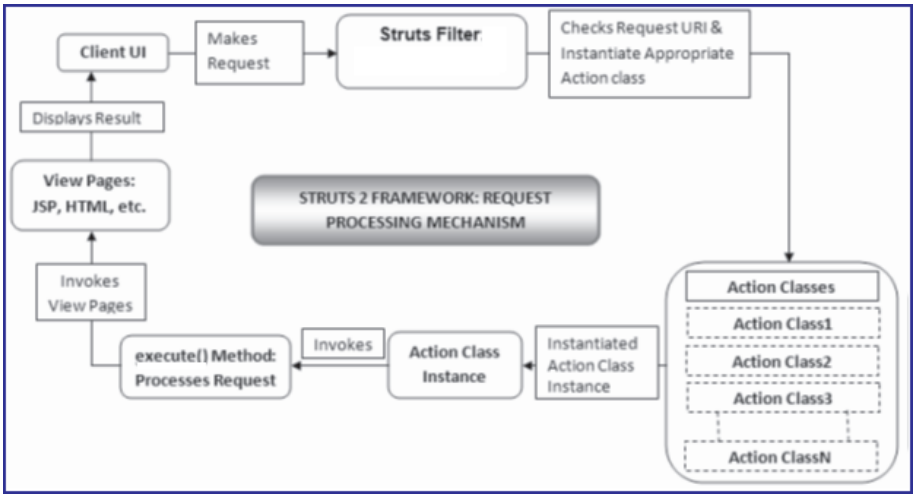


Figure 2.5: Request Processing Mechanism of Action Class

Struts 2 framework also provides helper interfaces and classes which can be used for adding additional functionalities to an action class.

Following are the two helpers:

- By implementing `Action` interface. The `Action` interface is provided in the package `com.opensymphony.xwork2` in the Struts 2 framework.
- By extending `ActionSupport` class defined in the `com.opensymphony.xwork2` package.
- **Action Interface**

  The `Action` interface is a helper interface which exposes the `execute()` method to the action class implementing it. The `Action` interface also declares five constants that can be returned from the action class.

Figure 2.6 shows the `Action` interface.

```
public Interface Action {
 //States that the action execution was successful
   public static final String SUCCESS = "success";
//states that the action execution was successful
//but do not show a view.
   public static final String NONE = "none";
//States that the action execution was a failure
   public static final String ERROR = "error";
//States that the action execution requires
//more input in order to succeed
   public static final String INPUT = "input";
//States that the action could not be executed as
//the user was not logged in
   public static final String LOGIN = "login";
// Business logic
   public String execute() throws Exception;
}
```

Figure 2.6: Action Interface

The `Action` interface enforces the implementation of the default `execute()` method. It provides common String values as CONSTANTS which can be returned from the `execute()` method. The constants are used internally by the framework and are as follows:

- `static String ERROR="error"` - This constant indicates that the execution of the action was a failure.
- `static String INPUT="input"` - This constant indicates that the action needs more input in order to execute.
- `static String LOGIN="login"` - This constant indicates that the action could not execute, as the user is not logged in.
- `static String NONE="none"` - This constant indicates that the execution of the action is successful, but the result View is not displayed.
- `static String SUCCESS="success"` - This constant indicates that the execution of the action is successful.

Code Snippet 2 shows the implementation of the `Action` interface.

Code Snippet 2:

```
public class MessageAction implements Action{
  public String execute(){
    return SUCCESS;
  }
}
```

Code Snippet 2 returns the constant value '`SUCCESS`' instead of String value from the `execute()` method.

ActionSupport Class

The `ActionSupport` class adds useful utilities to the class that extends it. The class implements the `Action` interface and some more useful interfaces. The developer extends the user defined action class from the `ActionSupport` class as it provides a default implementation for the `execute()` method. The default implementation of the `execute()` method in the `ActionSupport` class returns `Action.SUCCESS` String object.

In addition to the `Action` interface, the `ActionSupport` class also implements other interfaces.

They are as follows:

- `Validateable` and `ValidationAware` interfaces that provide programmatic, annotation-based, and declarative XML-based validation.
- `TextProvider` and `LocaleProvider` interfaces that provide support for localization and internationalization.
- `Serializable` interface that helps to create classes for easy transfer of binary data over a communication channel.

Figure 2.7 shows the `ActionSupport` class declaration semantics.

```
public ActionSupport implements Action,
Validateable, ValidationAware, TextProvider,
LocaleProvide, Serializable {
. . .
    public String execute()throws Exception {
            return SUCCESS;
    }
}
```

Figure 2.7: ActionSupport Class

2.3.2 Actions and Form Data

The Struts 2 framework follows the JavaBean paradigm. This means to access a form field's data, a getter/setter method is required. Thus, the Struts 2 framework, automatically moves the request parameters from the form to the JavaBeans properties that have matching names.

The data from the form fields is stored locally in the `Action` class by using the class instance variables or properties. The getter and setter methods help to retrieve and assign the data values to the properties respectively. The `execute()` method references the data using these properties.

Figure 2.8 displays the mapping of the form field with the JavaBeans properties.

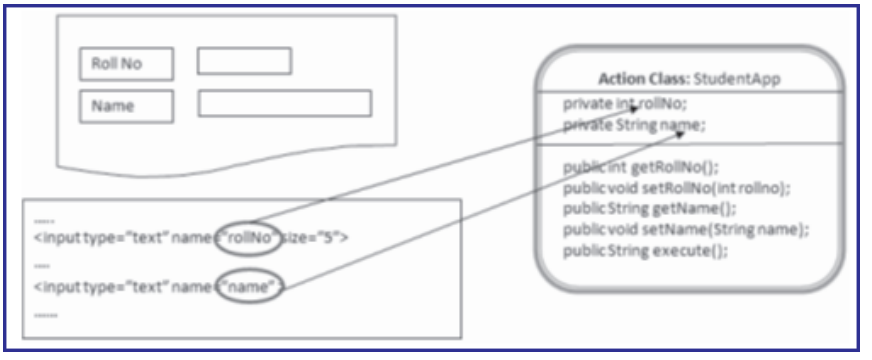


Figure 2.8: Action and Form Fields

As shown in figure 2.8, the **`rollNo`** and **`name`** parameters from the form are automatically assigned to the **`rollNo`** and name property of the JavaBean in the `Action` class using OGNL. In Struts 2, each request string or form value is a name-value pair. Note, that setter method does not always need to be a `String` value. Struts 2 is capable of converting `String` data type to the required data type. In other words, Struts 2 provides automatic type conversion between string and other data types using OGNL expression language.

2.3.3 Actions Return Value

After the completion of action, the `execute()` method returns a control string. This string helps the Struts 2 filter to decide the result/view page that should be rendered. The mapping of the resultant string and the view page to be rendered is defined in the configuration file named `struts.xml`.

For example, in the `struts.xml` file, **success** is the name of one of the result components that point to a JSP page that will be rendered as view.

2.3.4 Configuring Actions

The `struts.xml` is the configuration file used in Struts 2. It initializes the MVC resources and acts as glue to the resources. The resources include Interceptors, Action classes, and Results. Results are for preparing views commonly done by using JSP pages.

The elements that can be configured in the `struts.xml` file are packages, namespaces, includes, actions, results, interceptors, and exceptions.

Code Snippet 3 shows the implementation of `struts.xml` file.

Code Snippet 3:

```
<!DOCTYPE struts PUBLIC

. . .

<struts>

<include file="example.xml"/>

  <!-- Configuration for the default package. -->

  <package name="default" namespace="/" extends="struts-default"
method="execute">

   <action name="StudentApp" class="com.demo.myApp.StudentApp">

    <result name="success">/SuccessView.jsp</result>

    <result name="error">/ErrorView.jsp </result>

   </action>

  </package>

</struts>
```

In the given code, `struts.xml` file uses the following default elements:

- `<struts>` - This tag is the outermost tag that contains the Struts 2 specific configuration.
- `<package>` - This tag is used to group together configuration information that share common attributes such as Interceptor stacks or URL namespace. Thus, packages help to group together the application's components based on commonality of function or domain. Packages group the following components into a logical configuration unit:
  - Actions
  - Result Types
  - Interceptors
  - Interceptor stacks

  The attributes of the package element are as follows:

  - `name` – It indicates the name of the package.
  - `namespace` - It helps in separating different packages into different namespace and hence, helps in avoiding action mapping confliction.

    This attribute indicates the location of the user-defined action class.

It is used to generate the URL namespace to which the actions of these packages are mapped. In Code Snippet 3, the namespace is set to `/`.

Hence, to access the **StudentApp** action, the Web browser will need to set the request to /**StudentApp.action** within the Web application's context. This implies that if the web-app is called **studApp** and it is running on a server called `www.demoserver.com`, the action can be accessed by the given URL, `http://www.demoserver.com/studApp/StudentApp.action`

- `extends` – This attribute indicates the name of the parent package. It specifies the package name whose components will be inherited by the current package that is being defined. This attribute is similar to the `extends` keyword in Java. In Code Snippet 3, the `extends` attribute contains the value '`struts-default`'.

  By extending the `struts-default` package, the action will by default inherit the commonly needed Struts 2 components ranging from complete Interceptor stacks to all the common result types.

- `abstract` – The abstract attribute if set to true, indicates that this package will only be used to define inheritable components, besides actions. Thus, this attribute is used to create a base package without the action configuration.

- `<action>` - This tag maps an identifier with an action class. The framework uses action's name to process the request. This action name is mapped with the request URL. The attributes of the `<action>` element are as follows:

  - `name` – It indicates the name of the action within the Web application. The action's name is concatenated with the package's namespace to form the request URL.

    For example, the URL for the **StudentApp** application will be formed as: `http://www.demoserver/studApp/StudentApp.action`.

  - `class` – It indicates the Java class that will be instantiated for the Request.

  - `method` – It indicates the method name to be invoked on a request. This attribute is optional. If it is not specified, then the filter assumes the default value as '`execute()`' and invokes the method.

- `<result>` - It specifies the name of the result page. Every action element can have one or more result elements. Each result page has a view page associated with it which the action class invokes.

  The attributes of the `<result>` element are as follows:

  - `name` – It indicates the result name. It is an optional attribute. If it is not specified, the filter assumes the default value to be **success**.

  - `type` – It indicates the kind of result. This attribute is an optional attribute. If it is not specified, the filter assumes the default value to be `Dispatcher`. The `Dispatcher` forwards the Web browser to the appropriate View page.

- `<include>` - This tag is used to modularize a Struts 2 application. It allows the

inclusion of other configuration files. It is a child tag to the Struts 2 tag. This tag consists of only one attribute, named, `file`. The `file` attribute allows specifying the name of the file to be included. The included file should have a structure identical to the `struts.xml` configuration file.

2.4 Result and ResultTypes

After processing of the action, the result is sent in two parts, the result type and the result. Result defines the next action after the processing of the action is complete. The action returns a String value such as 'success' or 'error', as a result which is used to select the result element.

Normally, JSP technology is used for displaying the results, however, Struts 2 also provides other technologies for rendering results such as Velocity Templates, FreeMarker Templates, and EXtensible Stylesheet Language Transformations (XSLT).

The result type specifies the implementation details for the type of information that is returned to the user. They are pre-configured or provided as a plugin. The default result type is configured as a dispatcher which uses JSP to render the response to the user.

The framework provides several implementations of the `com.opensymphony.xwork2.Result` interface which can be ready to define the Result Types in the application.

Table 2.2 describes the configured Result Types provided by Struts 2 framework.

Result Type	Description
Dispatcher	This is the default result type rendered using JSP.
Chain	Chains actions with each other.
HttpHeader	Returns a configured HTTP header response.
Redirect	Redirects the user to a configured URL.
RedirectAction	Redirects the user to a configured action.
Stream	Streams raw data to the browser and is useful for downloadable content and images.
XSL	Renders XML to the browser, which is transformed using XSLT.
PlainText	Returns the content as plain text.
FreeMarker	The page is rendered as FreeMarker.

Table 2.2: Result Types

2.4.1 Configuring Result

The developer can set the result types in the `struts.xml` configuration file. They can either set the most often used result type as a default for a particular package or can set the specific result type with the `Result` element within the configuration file.

Code Snippet 4 shows the setting of default result type in `struts.xml`.

Code Snippet 4:

```
<result-types>
 <result-type name="dispatcher" default="true" class="org.apache.
struts2.dispatcher.ServletDispatcherResult" />
</result-types>
```

Code Snippet 4 declares `dispatcher` as the default result type for the package. This means each request will be forwarded to a Web resource such as JSP to render the result to the client.

Also, note that the child packages inherited from the parent package will also inherit the result types from it. The child packages can even set their own default result type.

Code Snippet 5 shows the setting of result type with the specific result.

Code Snippet 5:

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

In Code Snippet 5, the `type` attribute is optional. If not specified, then the framework will use the default result type, which is '`dispatcher`'. Similarly, if the `name` attribute is not specified, then the framework understands it as the default result, that is, '`success`'.

2.4.2 User-defined Results

Consider a scenario where the application needs to perform the Create, Read, Update, and Delete (CRUD) operations. Then, depending on the operation performed, the appropriate page needs to be rendered in the result.

The `execute()` method of the action class defines the logical name of the action. Apart from the pre-defined CONSTANTS result names, the action class can also return a bunch of logical result strings which can be mapped to appropriate result pages in the configuration file.

Code Snippet 6 shows the `execute()` method with different logical results.

Code Snippet 6:

```
public class CRUDAction
{
 public String execute() {
 if(createOper())
 { return "create"; }
```

```
  else if (deleteOper())
  { return "delete"; }
  else if (readOper())
 { return "read"; }
  else if (writeOper())
  { return "write"; }
  else
  return "error";
  }
}
```

Code Snippet 6 evaluates the invoked methods and return the appropriate result value from the `execute()` method. Based on the return result value, the appropriate View is displayed to the user.

Code Snippet 7 shows the mapping of the results with their corresponding view pages configured in the `struts.xml` file.

Code Snippet 7:

```
<struts>
  <action name="CRUDaction" class="actions.CRUDAction">
    <result name = "create">/create.jsp</result>
    <result name = "delete">/delete.jsp</result>
    <result name = "read">/read.jsp</result>
    <result name = "update">update.jsp</result>
   </action>
</struts>
```

2.5 Different Approaches for Developing Struts Components

Struts 2 was developed for Java SE 5 and hence, annotations, which is an important feature of JDK 5, is also available in Struts 2 for configuration. There are many ways components in which annotations can be used in Struts 2. Use of the Zero Configuration terminology means departure from XML based configuration to an annotation-based configuration. The requirement for `struts.xml` is eliminated with the introduction of the annotation.

Annotations allow the developers to add meta-data information into the Java source code. Struts 2 uses annotations. Annotations are added on Java classes that implement the `Action` interface. The advantage of using annotations is that it is elegant and eliminates the clutter that is present in the XML configuration file.

To enable zero configuration, a developer has to modify the configuration in the `web.xml` configuration file. In the configuration file, the developer has to specify to Struts 2 packages that have actions and are using annotations. This is specified in the `<init-param>` element of the filter configuration. The value in the `<param-name>` element is set to `actionPackages`. The `<param-value>` element contains the comma-delimited list of package names. Each of these packages and their sub packages will be checked for classes that implement `Action` or `ActionSupport` class or whose name ends in Action. The annotation configuration which is added to the action class is used during runtime configuration.

To use Struts 2 annotations, we need to add `struts2-convention-plugin` library in the classpath and in `web.xml`. Struts 2 filter configuration provide the Java package that contains action classes.

Code Snippet 8 displays the configuration setting of the `web.xml` configuration file.

Code Snippet 8:

```
...
<filter>
  <filter-name> struts 2 </filter-name>
  <filter-class>org.apache.struts2.dispatcher.ng.filter.
StrutsPrepareAndExecuteFilter</filter-class>
 <init-param>
   <param-name>actionPackages</param-name>
    <param-value>com.book.session2.actions</param-value>
 </init-param>
</filter>
...
```

In the given code, the `<param-name>` and the `<param-value>` elements have been used to specify the location of the packages containing the action classes using annotations.

2.5.1 Action Annotations

These annotations are available when the framework scans the classpath for `Action` classes. Some of the `Action` annotations are:

- **@Action**

  This annotation is used to mark the action class. It maps to some URL called as action path. The action path is made up of package, class, and method name of an action.

  Figure 2.9 shows the anatomy of mapping a URL to Struts 2 action.

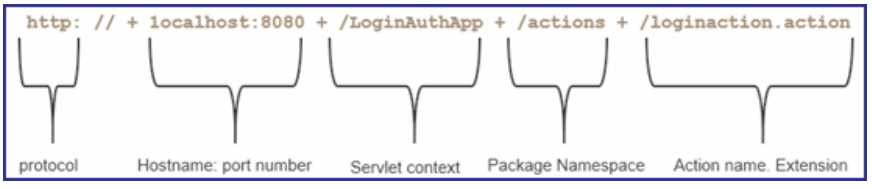


Figure 2.9: Mapping of a URL to Struts 2 Action

Code Snippet 9 shows the assigning of `@Action` annotation.

Code Snippet 9:

```
. . .
public class LoginAction extends ActionSupport{
{
 private String name;
 private String pwd;

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

```
public String getPwd() {
   return pwd;
 }
 public void setPwd(String pwd) {
   this.pwd = pwd;
  }
 // Applying action annotation
 @Action(value="/login")
 public String execute() throws Exception {
   if("admin".equals(getName()) && "admin".equals(getPwd()))
       return "SUCCESS";
   else
       return "ERROR";
  }
}
```

Struts 2 will automatically create action for classes name ending with `Action`. The action name is determined by removing the `Action` suffix and converting first letter to lowercase. So, if class name is `LoginAction`, then action will be '`login`'.

If the class is not annotated with `@Result` annotation to provide the result, then result pages are looked into `WEB-INF/content` directory of the project directory. The name of the result page should be `{action}-{return_string}.jsp`. So, if `LoginAction` action class is returning '`SUCCESS`', the request will be forwarded to `WEB-INF/content/login-success.jsp` page.

Thus, to avoid default naming conventions, the developers should apply proper annotation to the class with proper configuration.

Similarly, the developer can also map multiple URLs to a single `Action` class, by specifying `Action` annotation inside the `Actions` annotation. For example, `@Actions ({@Action("/action1"), @Action("/action2")})`.

- **@Namespace**

This annotation helps to set the path of the action URL. It is placed before the action class and it applies to all actions defined in the class, even if the fully qualified URLs are not specified.

Code Snippet 10 shows the use of @Namespace annotation on the action class.

Code Snippet 10:

```
...
package com.book.session2.actions;
import org.apache.struts2.convention.annotation.Namespace;
@Namespace("/custom")
public class HelloWorld extends ActionSupport {
 @Action("/hello/url")
 public String execute() {
   return SUCCESS;
 }
  @Action("url")
  public String sayHello() {
      return SUCCESS;
  }
}
...
```

In the given code, the action will be invoked using the URLs: **/hello/url** and **/custom/url**. These are two different URLs.

If no namespace annotation is used, the namespace will be generated from the package name. This is done by dropping the part of the package name that appears in the **actionPackages** configuration value. In the example, **com.book.session2.actions** was the value present in **actionPackages** and the action being configured is **com.book.session2.actions**, then the **namespace is /actions/HelloWorld**.

- **@Result and @Results**

This annotation is used for configuring a result for the return value. Result annotation can be declared at global or local level. If it is added at the class level, then it is a global annotation and will be shared across all actions defined within the action class. If it is defined at the method level, then it is local. The annotation accepts four parameters for configuration.

They are as follows:

- **name**: The string value returned from the methods that processes the request.
- **value**: A value that the result type uses. For example, for JSP this is the name of the template to render.
- **type**: The class of the result type. Since the class name is specified, it need not be enclosed within quotes.
- **parameters**: An array of string parameters.

The `@Results` annotation is used for configuring multiple results. You place multiple `@Result` annotations within the `@Results` annotation.

Code Snippet 11 shows the use of `@Result` annotation.

Code Snippet 11:

```
...
package com.book.session2.actions;
@Result(name="success", value="/jsp/success.jsp",type=
ServletDispatcherResult.class)
public class HelloWorld extends ActionSupport {
 public String execute() {
  return SUCCESS;
 }
}
...
```

The `Result` annotation maps the result code with the result page. Here the result code 'success' is mapped to the result '`/jsp/successPage.jsp`'.

2.6 Creating a Web Application Using Struts 2

Struts 2 application is based on MVC design pattern. To create a Web application in Struts 2, you need to identify the Model, View, and Controller.

The steps to be followed are:

- Create views which will provide user interface for your applications.
- Create an action class; define the input properties and its getter/setter methods within the action class and define the `execute()` method to perform the action implementation.

- Establish relationship between the Views and Controllers using configuration file, `struts.xml`, or using annotations.
- Modify the `web.xml` file to enable the Web application with Struts 2 filter.
- Build and run the Struts 2 Web application.

2.6.1 Create Views

In the Web application, first you need to describe the Views that represent the presentation layer of the application. To create view you need to use JSP/HTML technology along with Struts 2 tag library. In the demo application, there are three views named **index.jsp**, **AcceptDetails.jsp**, and **DisplayDetails.jsp**. These view names are not standard, whereas **index.jsp** is the standard view name used by Struts 2 application.

Code Snippet 12 shows the implementation of **index.jsp** page.

Code Snippet 12:

```
...
<%@taglib prefix="s" uri="/struts-tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
  <%@taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Welcome Page</title>
</head>
<body>
<h1>Welcome to Struts 2 Example!</h1>
Click to enter <s:a href="example/AcceptDetails.jsp"> Details </s:a>
</body>
</html>
...
```

In the given code, the **index.jsp** page uses Struts 2-specific HTML tag. This is the first page of the application where the user sees a Welcome Message and a link. The `<s:a href>` tag is used to emulate an HTML link. On clicking this link the user is taken to another JSP page.

Code Snippet 13 shows the implementation of the **AcceptDetails.jsp** page.

Code Snippet 13:

```
...
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>User Details Page</title>
</head>
<body>
<h1>User Details Page </h1>
    <s:form action="WelcomeMessage">
    <s:textfield name="FirstName" label="First Name"/>
    <s:textfield key="LastName" label="Last Name"/>
    <s:submit/>
</s:form>
</body>
</html>
...
```

The `<s:textfield/>` emulates an HTML text field. This text field allows the user to enter text. The `<s:submit/>` tag emulates an HTML submit button. On clicking the submit button, the `<s:form/>` tag is created and populated with the values from the `<s:textfield/>` tag.

Code Snippet 14 shows the implementation of the **DisplayDetails.jsp** page.

Code Snippet 14:

```
...
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>User Profile Page</title>
</head>
<body>
    <h1>User Details:</h1>
    <h3><s:property value="message"/></h3>
</body>
</html>
...
```

In the given code, the **AcceptDetails.jsp** page accepts the user details. When the user submits the form, the **firstname** and **lastname** entered in the **AcceptDetails.jsp** page. The view is displayed only after the successful submission of the form by the user.

In this code, the `<s:property value="message"/>` tag is used to automatically find the message property and print the value in the message property.

2.6.2 Creating the Action Class

The action class will embed the business logic. The data sent from **AcceptDetails.jsp** page is processed by action class. The action class extends the `ActionSupport` class.

Code Snippet 15 shows the implementation of the action class.

Code Snippet 15:

```
...
public class WelcomeMessage extends ActionSupport {
 /*
  * Declaring properties
  */
private String firstName;
private String lastName;
private String message;
/**
* Returns the user first name
* @return first name
*/
   public String getFirstName() {
      return firstName;
   }
/**
* Sets the user first name
* @param firstName
*/
   public void setFirstName(String firstName) {
         this.firstName = firstName;
    }
/**
* Returns the user last name
```

```
* @return last name
*/
   public String getLastName() {
     return lastName;
   }
/**
* Sets the user last name
* @param lastName
*/
    public void setLastName(String lastName) {
        this.lastName = lastName;
   }
/**
* Defines the message property containing result value
* @return result message
*/
      public String getMessage() {
        return message;
      }
/**
* Defines the message property containing result value
* @return result message
*/
     public String getMessage() {
         return message;
     }
/**
* Sets the result message value
* @param message
```

```
*/
    public void setMessage(String message) {
      this.message = message;
    }
/**
* Defining execute method for processing action
*/
    public String execute() throws Exception {
         java.util.Date dt = new java.util.Date();
        //Defining temporary variable for storing the result message
          String tmpmessage;
          tmpmessage = "Hello " + getFirstName() + " " + getLastName();
          tmpmessage = tmpmessage + " You have logged in at " +
                                                    dt.toString();

          //Setting the result message to the message variable
             setMessage(tmpmessage);
             return SUCCESS;
      }
}
...
```

In the given code, all the input fields submitted in the **AcceptDetails.jsp** page are declared as properties in the **`WelcomeMessage`** action class. The getter/setter methods for these properties have been declared. The OGNL expression language maps the input fields of the **AcceptDetails.jsp** page with the properties in the **`WelcomeMessage`** action class. Thus, for this reason the input field names of the **AcceptDetails.jsp** page are the same as the names of the properties in the **`WelcomeMessage`** action class.

2.6.3 Configuring Actions in struts.xml File

Code Snippet 16 shows the implementation of the `struts.xml` file.

Code Snippet 16:

```
. . .
<package name="example" namespace="/example" extends="struts-
default">
<action name="WelcomeMessage" class="example.WelcomeMessage">
<result name="success">/example/DisplayDetails.jsp</result>
</action>
</package>
</struts>
. . .
```

The `<action />` element maps an identifier to handle an action class. The mapping is used by the action's name and framework to determine how to process the request when a request is matched.

The attributes of action element are `name` and `class`. The `name` attribute identifies the name of the action which matches a part from the URL. The `class` attribute identifies the action class to be executed with the full package description. The sub-element of **action** is `result`.

The `<result />` element displays the desired view. The `execute()` method of the action class returns a String value. This string value is used to select a `<result />` element. For example, if the `execute()` method of the action class named **WelcomeMessage.java** returns **SUCCESS**, then this value is matched with the result named, **success**. After matching the value, the **DisplayDetails.jsp** page is invoked to display the result. The `execute()` method can return **ERROR** or **INPUT** for errors or conversion issues, respectively. These values should then be matched with the particular result name present in the `<result />` element and a JSP page should be rendered.

Figure 2.10 displays the application startup page.

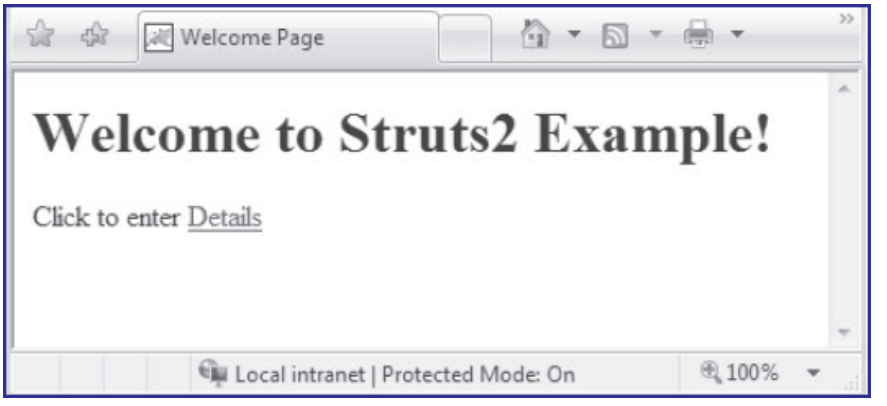


Figure 2.10: Application Startup Page

Figure 2.11 displays the input page for entering the user details.

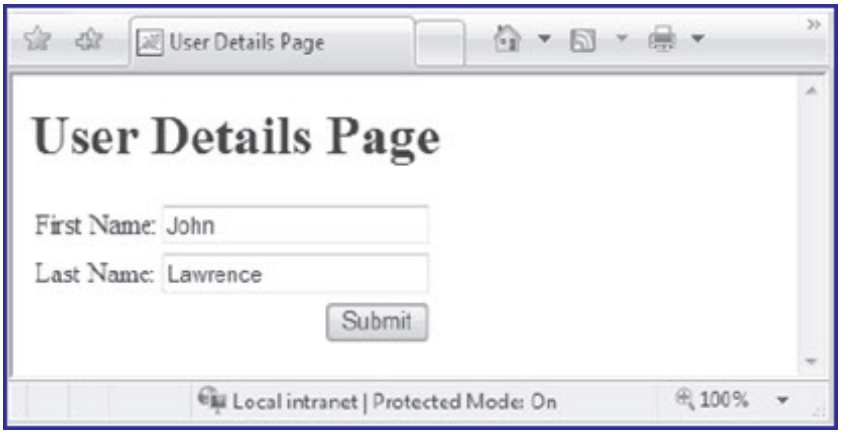


Figure 2.11: User Details Page

Figure 2.12 displays the Welcome page.

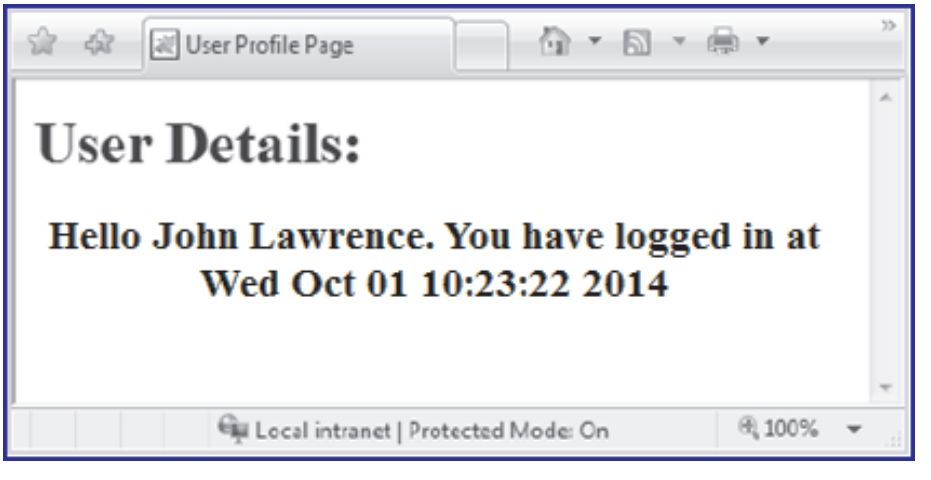


Figure 2.12: Welcome Page

Check Your Progress

1. Which of the following statement(s) regarding Struts 2 framework are true?

(A)	Struts 2 framework is an extension of Struts and JSF framework.
(B)	Struts 2 framework supports the development of persistence logic in the enterprise applications.
(C)	In Struts 2 framework, the component information is configured in `struts.xml` file.
(D)	Struts 2 is an extensible framework can be easily integrated with any other frameworks.
(E)	Struts 2 is based on a pull-MVC framework.

(A)	A, C, E	(C)	A, C, D
(B)	B, C, D	(D)	A, D, E

2. Which among the following components acts as a Controller in Struts 2 framework?

(A)	`Configuration Manager`	(C)	`ActionMapper`
(B)	`StrutsPrepareAndExecuteFilter`	(D)	`Interceptors`

3. Which is the correct syntax of the `execute()` method declared in the `Action` class?

(A)	`public void execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws IOException`
(B)	`public ActionForward execute(ActionMapping mapping, ActionForm form, ServletRequest request, ServletResponse response) throws Exception`
(C)	`public String execute() throws Exception`
(D)	`public String execute(ActionForm form, ServletRequest request, ServletResponse response) throws ServletException, IOException`

Check Your Progress

4. In the request life cycle of Struts 2, which component invokes the Interceptors to apply the common functionalities?

(A)	`ActionServlet`	(C)	`ActionInvocation`
(B)	`ActionMapping`	(D)	`FilterDispatcher`

5. Which of the following result type redirects the user to a configured action?

(A)	Redirect	(C)	Dispatcher
(B)	Chain	(D)	RedirectAction

Answer

1.	A
2.	B
3.	C
4.	C
5.	D

Summary

- Apache Struts is a Java-based framework that separates the business logic from the presentation layer.
- Some of the important features of Struts 2 framework are namely, simplified design, MVC components, POJO-based actions, enhanced tags, OGNL and ValueStack, easy integration, template support, annotations, AJAX support, and View technologies.
- In Struts 2, StrutsPrepareAndExecuteFilter plays the role of a Controller. A Model is a business object and has no user interface. View component returns the result page to the Web browser.
- The various components of the Struts 2 framework are namely, StrutsPrepareAndExecuteFilter, Action Mapper, Action Proxy, Action Invocation, and Interceptors.
- The StrutsPrepareAndExecuteFilter accepts the request and then consult ActionMapper to determine the suitable Action.
- The main functions of Struts 2 action includes responding to a user request, executing business logic, and then returning a result to the user based on configuration file (struts.xml) to render the view page.
- Struts 2 framework also provides helper interfaces and classes which can be used for adding additional functionalities to an action class. They are Action interface and Action-Support class.
- The data from the form fields is stored locally in the Action class by using the class instance variables or properties.

Onlinevarsity

your e-way to learning

Technowise

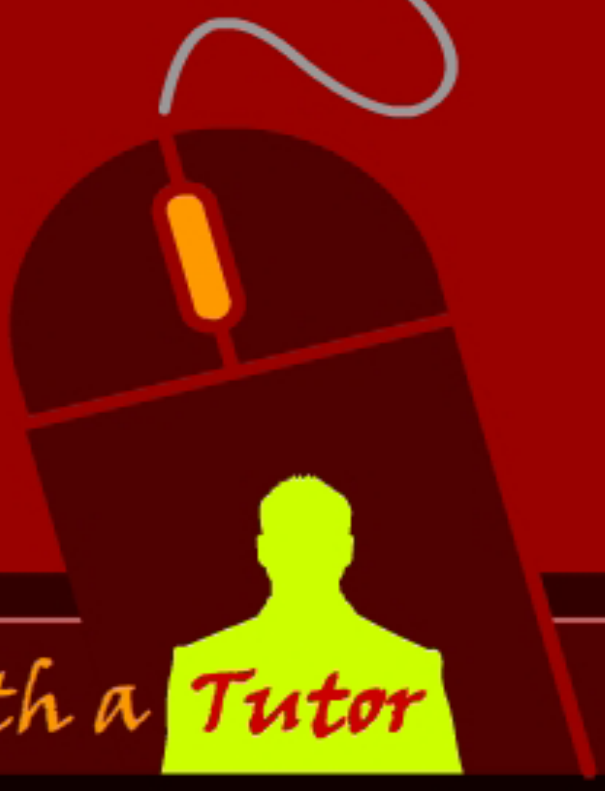
Login to

www.onlinevarsity.com

Tutor Chat

Onlinevarsity
your e-way to learning

are just a click away



To chat with a Tutor

login to www.onlinevarsity.com

03 Struts 2 - Interceptors and Tags

Welcome to the Session, **Struts 2 - Interceptors and Tags**.

This session describes the concept of Interceptors in Struts 2 framework. It explains different types of Interceptors and their configuration in Struts 2. Further, the session explains the stacking of Interceptors in the configuration file and also explains how to develop custom Interceptors for the Struts 2 Web application. The session describes different types of Struts 2 Tags used in developing Views namely, Data tags, Control tags, and User Interface (UI) tags.

In this Session, you will learn to:

- Describe the purpose of Interceptors in Struts 2 framework
- Explain different types of built-in Interceptors
- Explain how to configure Interceptors in Struts 2 application
- Describe Interceptors Stacks
- Explain the process of creating custom Interceptors in Struts 2 framework
- List the different types of tags used in Struts 2 View page
- Explain various Data tags
- Explain various Control tags
- Explain various User Interface (UI) tags

Session
03

Struts 2 - Interceptors and Tags

3.1 Introduction

Interceptor is an important feature introduced in the Struts 2 framework. Interceptors sit between controller and action. They intercepts the request and response, processes the request before and after invoking action. In other words, Interceptors provide a mechanism to supply pre-processing and post-processing functionalities required by action classes. These functionalities include double-submit guards, type conversion, object population, validation, file upload, page preparation, and so on.

Interceptor is an object which intercepts an action dynamically. It allows the developers to write a code which can be executed by the controller before and after invoking actions. Each and every Interceptor object is pluggable, so user can decide exactly which features an action need to support.

The main aim of having Interceptors is to separate the core functionality that may be applicable to multiple actions.

3.1.1 Interceptors

An Interceptor or a stack of Interceptors is configured for an action. These Interceptors are executed, before the execution of the mapped action, to provide all the pre-processing functionalities to the request. Again, these Interceptors are executed after the action is executed to provide additional processing.

There are a set of built-in Interceptors provided by the Struts 2 framework. These built-in Interceptors can be used to provide the required functionalities to action. The Struts 2 framework also allows the developers to develop custom Interceptors for defining the specific functionalities for the Web application.

A custom Interceptor class can be created by implementing the `com.opensymphony.xwork2.interceptor.Interceptor` interface or by extending the `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class. Each of the Interceptor class has a method, `String intercept(ActionInvocation invocation)` that allows processing of the request before or after the processing of the request by `ActionInvocation`. This method returns a String value.

3.1.2 Request and Interceptors

Every request that is received by the Struts 2 framework passes through each Interceptor. Once the request is received, the Interceptors can do the following:

- Ignore the request
- Act on the request data
- Short-circuit the request and prevent the `Action` class method to be executed

In Struts 2, the default Interceptors are configured in the `struts-default.xml` file. It contains interceptors that do all the required pre-processing for the request. The `struts-default`.xml is included in the application's configuration by default, so all of the predefined interceptors are provided in the Struts 2 framework.

3.2 Working of Interceptors

The `ActionInvocation` class is responsible for the execution of the interceptors. `ActionInvocation` class encapsulates the processing details associated with the execution of a particular action. On receiving a request, the framework decides to which action the URL maps. An instance of this action is added to the newly created instance of `ActionInvocation`. Next, the framework consults the configuration files or the Annotations to decide the name and the sequence of firing of the Interceptor.

Figure 3.1 shows the invocation of interceptors by the `ActionInvocation` class in the Struts 2 framework.

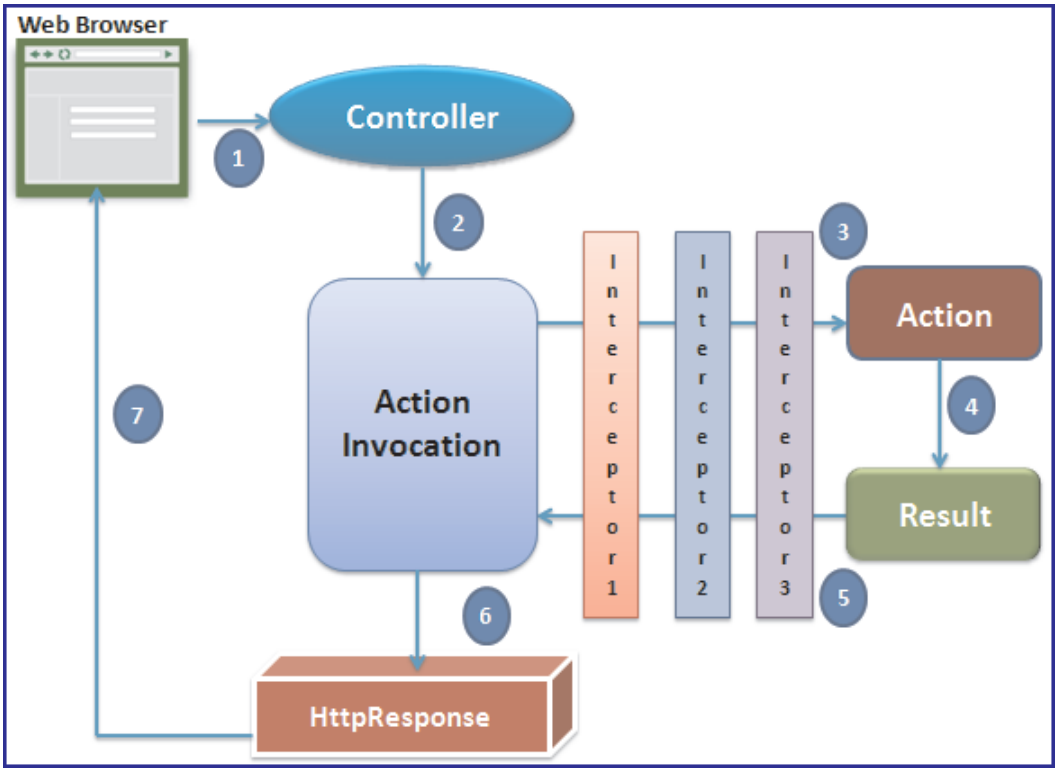


Figure 3.1: Invocation of Interceptors by ActionInvocation

`ActionInvocation` class stores the references of these Interceptors, besides storing other important information such as Servlet request objects and so on.

Once the `ActionInvocation` class contains all the required information, it exposes the `invoke()` method to the framework. The framework invokes this method and the `ActionInvocation` class starts the invocation process by executing the first Interceptor in the stack.

The `ActionInvocation` class is responsible for tracking the stage of the invocation process and passing control to the appropriate Interceptor in the stack. Once the control reaches the Interceptor, the `intercept()` method is invoked.

All the Interceptors are invoked one after the other by recursively invoking the `invoke()` method of the `ActionInvocation` class. The `ActionInvocation` class checks the states and executes whichever Interceptor comes next. After all the Interceptors have been invoked, the `invoke()` method will cause the action to be executed.

The Interceptor performs a three-stage conditional style. The three stages are listed as follows:

- Perform some pre-processing.
- Pass control to successive Interceptors, and finally to action by executing the `invoke()` method.
- Perform some post-processing.

In the pre-processing phase, the Interceptor is used to prepare, filter, and manipulate any of the important data available to it. If an Interceptor determines that the request processing should not continue, it can return a string rather than invoke the `invoke()` method of the `ActionInvocation` class. After a string is returned by the `invoke()` command, Interceptors can also change any of the objects or data available to them as part of pre-processing.

3.2.1 Configuring Interceptors

The Interceptors can be specified using any one of the following approaches:

- By declaring all the Interceptors in the `struts.xml` configuration file. The Interceptor classes are defined using a name-class pair in the configuration file.
- By declaring all the Interceptors that are used for the Web application in an XML file and including that XML file in `struts.xml` configuration file.

All the Interceptors that are required to perform the pre-processing functionalities of the request for a given action should be defined in the action mapping for that specific action. For example, if you want to use two Interceptor for an action class, then they need to be declared.

Code Snippet 1 shows the declaration of the Interceptors.

Code Snippet 1:

```
...
<include file="struts-default.xml">
<package name="default" extends="struts-default">
<interceptors>
<interceptor name="timer1" class="timer1_class">
<interceptor name="timer2" class="timer2_class">
</interceptors>
...
<action name="Login" class="example.Login">
<interceptor-ref name="timer1" />
<interceptor-ref name="timer2" />
<result name="success">/example/DisplayDetails.jsp</result>
<result name="error">/example/error.jsp</result>
</action>
</package>
...
```

In the code, every Interceptor class is provided a name which is used for reference. The `<interceptor-ref name="...">` element is used to declare the list of Interceptors that will intercept the action request. Interceptors are executed in the same order in which they are declared in the action mapping.

In the code, the Interceptor classes will be executed to provide all pre-processing functionalities before the execution of the **Login** action class.

3.2.2 Default Interceptor Configuration - struts-default.xml

The `struts-deafult.xml` file is the base configuration file with default settings of the components in the Struts 2 framework. This file is automatically included into `struts.xml` file to provide default configuration settings to the Web applications.

The `struts-default.xml` file contains the definitions of all Interceptors and Interceptor stacks. To use the Interceptors bundled with the framework, user can include `struts-default.xml` in the `struts.xml` configuration file by extending the package from the struts-default package. These Interceptors classes are present in the `struts-default.xml` file which is contained in the `struts2-core.jar`.

Figure 3.2 shows the list of interceptors defined in the `struts-default.xml` file.

```
<interceptors>
    <interceptor name="alias" class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
    <interceptor name="autowiring" class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor"/>
    <interceptor name="chain" class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
    <interceptor name="conversionError" class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"/>
    <interceptor name="cookie" class="org.apache.struts2.interceptor.CookieInterceptor"/>
    <interceptor name="cookieProvider" class="org.apache.struts2.interceptor.CookieProviderInterceptor"/>
    <interceptor name="clearSession" class="org.apache.struts2.interceptor.ClearSessionInterceptor" />
    <interceptor name="createSession" class="org.apache.struts2.interceptor.CreateSessionInterceptor" />
    <interceptor name="debugging" class="org.apache.struts2.interceptor.debugging.DebuggingInterceptor" />
    <interceptor name="execAndWait" class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor"/>
    <interceptor name="exception" class="com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor"/>
    <interceptor name="fileUpload" class="org.apache.struts2.interceptor.FileUploadInterceptor"/>
    <interceptor name="i18n" class="com.opensymphony.xwork2.interceptor.I18nInterceptor"/>
    <interceptor name="logger" class="com.opensymphony.xwork2.interceptor.LoggingInterceptor"/>
    <interceptor name="modelDriven" class="com.opensymphony.xwork2.interceptor.ModelDrivenInterceptor"/>
    <interceptor name="scopedModelDriven" class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor"/>
    <interceptor name="params" class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>
    <interceptor name="actionMappingParams" class="org.apache.struts2.interceptor.ActionMappingParametersInteceptor"/>
    <interceptor name="prepare" class="com.opensymphony.xwork2.interceptor.PrepareInterceptor"/>
    <interceptor name="staticParams" class="com.opensymphony.xwork2.interceptor.StaticParametersInterceptor"/>
    <interceptor name="scope" class="org.apache.struts2.interceptor.ScopeInterceptor"/>
    <interceptor name="servletConfig" class="org.apache.struts2.interceptor.ServletConfigInterceptor"/>
    <interceptor name="timer" class="com.opensymphony.xwork2.interceptor.TimerInterceptor"/>
    <interceptor name="token" class="org.apache.struts2.interceptor.TokenInterceptor"/>
    <interceptor name="tokenSession" class="org.apache.struts2.interceptor.TokenSessionStoreInterceptor"/>
    <interceptor name="validation" class="org.apache.struts2.interceptor.validation.AnnotationValidationInterceptor"/>
    <interceptor name="workflow" class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>
    <interceptor name="store" class="org.apache.struts2.interceptor.MessageStoreInterceptor" />
    <interceptor name="checkbox" class="org.apache.struts2.interceptor.CheckboxInterceptor" />
    <interceptor name="datetime" class="org.apache.struts2.interceptor.DateTextFieldInterceptor" />
    <interceptor name="profiling" class="org.apache.struts2.interceptor.ProfilingActivationInterceptor" />
    <interceptor name="roles" class="org.apache.struts2.interceptor.RolesInterceptor" />
    <interceptor name="annotationWorkflow" class="com.opensymphony.xwork2.interceptor.annotations.AnnotationWorkflowInterceptor" />
    <interceptor name="multiselect" class="org.apache.struts2.interceptor.MultiselectInterceptor" />
    <interceptor name="deprecation" class="org.apache.struts2.interceptor.DeprecationInterceptor" />
```

Figure 3.2: Interceptors Defined in struts-default.xml

Interceptors are declared using the `interceptor` element. All interceptor elements are nested within the `interceptors` element and are associated with a name and a qualified class name.

Following points are required to be taken care of when you are using framework Interceptors:

- **`struts-default.xml`** file should be included in the `struts.xml` file.
- Action mappings should be enclosed within the `<package>` element which extends `struts-default` package.

3.3 Struts 2 Framework Interceptors

Each Interceptor implements logic for a specific function which is common to all Web applications. Thus, it can be said that the Interceptor class acts as a reusable component which is used in different Web applications.

These Interceptors can be used either by extending the `struts-default` package or defining these Interceptors in the package using the `<interceptor>` element.

Some of the common Interceptors are discussed as follows:

- **Chaining Interceptor**

  The Chaining class extends `AbstractInterceptor` class. The function of this Interceptor is to copy all the objects present in the ValueStack of the currently executing action class to the ValueStack of the next action class to be executed in action chaining. Thus, the main requirement in action chaining is copying the parameters from ValueStack of one action to the ValueStack of the next action class. In other words, one action class needs to access the parameters of the previously executed action class. You can define a collection of include and exclude elements to control which parameter is to be copied and how it is to be copied.

  Table 3.1 lists the methods available in Chaining Interceptor class.

Methods	Description
`Collection getExcludes()`	Returns a collection of excluded parameters.
`Collection getIncludes()`	Returns a collection of included parameters.
`String intercept (Action-Invocation action)`	Implements the code to handle interception.
`Avoid setExcludes (Collection excludes)`	Sets a collection of excluded parameters.
`void setExcludes (Collection excludes)`	Sets a collection of included parameters.

Table 3.1: Chaining Interceptor Class – Methods

Code Snippet 2 shows the implementation of the Chaining Interceptors.

Code Snippet 2:

```
...
<package>
<action name="Login" class="example.Login">
 <interceptor-ref name="basicStack" />
  <result name="success" type="chain">action1</result>
  <result name="error">/example/error.jsp</result>
</action>

<action name="action1" class="example.action1">
 <interceptor-ref name="chain" />
<interceptor-ref name="basicStack" />
 <result name="success" >/example/DisplayDetails.jsp</result>
</action>
</package>
...
```

- **Checkbox Interceptor**

  The `CheckboxInterceptor` class implements the `Interceptor` interface. The class has a field named uncheckedValue of String data type and a method `void setUncheckedValue (String unchecked)` to set value to this field. It is included in the default Interceptor stack such as `basicStack`, `defaultStack`, and `paramsPrepareParamsStack` which is defined in the `struts-default.xml` file.

- **Conversion Error Interceptor**

  The `ConversionErrorInterceptor` class extends the XWork 2 conversion error interceptor. This Interceptor is used when the action implements the `ValidationAware` interface or the `ActionSupport` class. The value of all the fields is stored so that during subsequent requests, the values will be available for display. For example, consider that the value in a field is **abc** and it cannot be changed to `int` data type. Hence, you need to display an error message. While displaying the error message for this field, you will also display to the user the original value that was entered, that is **abc**, along with the error.

  The method, `ActionContext.getConversionErrors()` will return a map containing all the conversion errors.

It is included in the default Interceptor stack such as `basicStack` and `defaultStack` which is defined in the `struts-default.xml` file.

Code Snippet 3 shows the implementation of the `ConversionError` Interceptor.

Code Snippet 3:

```
...
<action name="Login" class="example.Login">
 <interceptor-ref name="params" />
 <interceptor-ref name="conversionError" />
<result name="success">/example/DisplayDetails.jsp</result>
</action>
...
```

- **Create Session Interceptor**

The `CreateSession` class extends the `AbstractInterceptor` Interceptor class. This Interceptor class is used for creating an `HttpSession`. The class defines the logic to be executed when the Interceptor invokes the `intercept()` method.

Code Snippet 4 shows the implementation of the `CreateSessionInterceptor` class.

Code Snippet 4:

```
...
<action name="Login" class="example.Login">
<interceptor-ref name="createSession" />
<interceptor-ref name="defaultStack" />
<result name="success">/example/DisplayDetails.jsp</result>
</action>
...
```

- **Debugging Interceptor**

The `DebuggingInterceptor` class implements the `Interceptor` interface. It provides developer with different debugging screens. The devMode should be enabled in the `struts.properties` file for the Interceptor to intercept the request. The debug request parameter is removed from the list before the action is executed.

Values for the debug parameters are as follows:

- `xml` – dumps in an XML document the parameters, value stack, context, and session.

- `console` – displays an OGNL console to test OGNL expressions against the value stack.
- `command` – is used with an OGNL console as it returns a string result after testing the OGNL expression.

The methods present in this Interceptor class are as follows:

- `String getParameter(String key)`
- `String intercept(ActionInvocation action)`
- `void printContext()`
- `void setDevMode(String mode)`

A request string such as `http://localhost:8080/Test/Login.action?debug=xml`, will display parameters, context, session, and ValueStack in XML format.

Servlet Config Interceptor

The `ServletConfigInterceptor` class allows the developer to inject various objects from the Servlet environment in the `Action` class. It allows to set the properties of the injected objects through setter methods provided by the interface.

The interfaces found in the `org.apache.struts2.Interceptor` package can be implemented by the `Action` class. These are as follows:

- `ServletContextAware`—Sets the `ServletContext`.
- `ServletRequestAware`—Sets the `HttpServletRequest`.
- `ServletResponseAware`—Sets the `HttpServletResponse`.
- `ParameterAware`—Sets a map of the request parameters.
- `RequestAware`—Sets a map of the request attributes.
- `SessionAware`—Sets a map of the session attributes.
- `ApplicationAware`—Sets a map of application scope properties.
- `PrincipalAware`—Sets the `Principal` object used for applying security.

Exception Interceptors

The `ExceptionMapping` Interceptor class extends the `AbstractInterceptor` class. It provides the functionality of exception handling by displaying a page, describing the real problem. The Interceptor enables the mapping of the exception to a result code and does not throw an exception. The encountered exception is wrapped within an `ExceptionHandler` and pushed on the stack.

Besides the `intercept()` method, the class has getter/setter methods for its fields such as `logCategory`, `logEnabled`, and `logLevel`.

To handle exceptions, exceptions need to be configured in the `struts.xml` file. The `exception` Interceptor should be added in the Interceptor stack and should be the first Interceptor in the Interceptor stack. This is required because any exception raised by the action class or any Interceptors can be accessed and caught.

Code Snippet 5 shows the implementation of the Exception Interceptor in the `struts.xml` configuration file.

Code Snippet 5:

```
...
<global-exception-mappings>
 <exception-mapping exception="java.lang.Exception"
result="exception" />
</global-exception-mappings>
<action name="Login" class="example.Login ">
<interceptor-ref name="exception" />
<interceptor-ref name="prepare" />
<interceptor-ref name="debugging" />
<interceptor-ref name="params" />
<interceptor-ref name="defaultStack" />
<result name="success">/example/DisplayDetails.jsp</result>
<result name="exception">/example/exception.jsp</result>
</action>
...
```

- **Parameters Interceptor**

The `ParametersInterceptor` class sets the values of all the parameters on the ValueStack. The `ActionContext.getParameters()` method obtains all the parameters from the ValueStack. The `ValueStack.setValue(String, Object)` method is used for setting the value in the ValueStack. The values are applied to the actions present in the ValueStack after the form requests are submitted.

The parameter names are given as OGNL statements. As the OGNL is an expression language, it simplifies the accessibility of data stored in the `ActionContext`. The Struts framework sets the ValueStack as the root object of OGNL. The action object is pushed into the ValueStack. If the value in the parameter map contains an assignment operator, multiple expressions, or references of objects from context, then it is not allowed by the Interceptor. If the action class implements the `ParameterNameAware` interface, the `acceptableParameterName (String name)` method is used to decide whether the parameter with the given name is acceptable by the action or not.

On invocation of the interceptor, the value in three flags are set.

Following are those flags:

- `XWorkMethodAccessor.DENY_METHOD_EXECUTION` - This flag restricts the invocation of methods when it is set ON.
- `InstantiatingNullHandler.CREATE_NULL_OBJECTS` - This flag automatically creates null reference when it is set ON.
- `XWorkConverter.REPORT_CONVERSION_ERRORS` - This flag reports errors when conversion of data type takes place provided it is set ON.

Other Interceptor

Some of the other Interceptors provided by Struts 2 framework are listed in table 3.2.

Interceptor	Description
Alias Interceptor	Aliases a named parameter to a different parameter name.
Execute and Wait Interceptor	Displays an intermediary waiting page to the user while the action is executed in the background.
Message Store Interceptor	Stores and retrieves error messages, field errors, and action errors in the session for actions. These are used for actions implementing `ValidationAware` interface.
Roles Interceptor	Allows the execution of action provided the user belongs to one of the configured roles.
Validation Interceptor	Provides validation support for actions by checking the action against all the validation rules declared in the Validation framework configuration files such as `Action-validation.xml`.
Scope Interceptor	Looks for the specified parameters and pulls these parameters from the given scope.
Timer Interceptor	Logs the amount of time elapsed between the execution of action and the execution time of the Interceptors in the Interceptor stack of the action.
Model Driven Interceptor	This interceptor pushes the Model result on the ValueStack. However, to do so, the `Action` class must implement `ModelDriven` interface.

Table 3.2: Other Interceptors

3.3.1 Interceptor Stacks

Sometimes, user may need the same set of Interceptors for different actions in the same applications or for different applications. Thus, these set of Interceptors can be grouped into an Interceptor stack and the stack name can be referred in the action mapping. This will save time as user do not have to repeatedly write the same list of Interceptors in every action mapping. The `<interceptor-stack>` element is used to define the stack of Interceptors.

Similar to defining Interceptors, different Interceptor stacks are also defined in the `struts-default.xml` file. These Interceptor stacks can consist of individual Interceptors as well as other Interceptor stacks. Thus, it can be said that Interceptor stacks consist of a group of Interceptors that are commonly used and required.

Table 3.3 lists some of the Interceptor stacks.

Interceptor Stack	Description
`validationWorkflowStack`	Adds to the basic stack features, the validation and workflow features.
`fileUploadStack`	Adds to the basic stack feature, the file uploading feature.
`chainStack`	Adds to the basic stack feature, the chaining feature.
`defaultStack`	Provides a complete stack, including debugging and profiling.
`executeAndWaitStack`	Provides an execute and wait stack by displaying a waiting page to the user. This is useful when file is being uploaded.
`i18nStack`	Handles the settings for the specified locale for the current action request.
`jsonValidationWorkflowStack`	Serializes validation and action errors into JSON.

Table 3.3: Interceptor Stacks

Code Snippet 6 shows the declaration of Interceptor stack.

Code Snippet 6:

```
...
<include file="struts-default.xml">
<package name="default" extends="struts-default">
<interceptors>
<interceptor name="timer1" class="timer1_class">
<interceptor name="timer2" class="timer2_class">

<interceptor-stack name="custom_stack">
<interceptor-ref name="timer1" />
<interceptor-ref name="timer2" />
</interceptor-stack>
</interceptors>
...
<action name="Login" class="example.Login ">
<interceptor-ref name="custom_stack" />
<result name="success">/example/DisplayDetails.jsp</result>
<result name="error">/example/error.jsp</result>
</action>
</package>
...
```

The `<interceptor-stack name="..." >` element is used to define Interceptor stack.

When no Interceptor has been declared for an action mapping and the package extends the struts-default package, the default Interceptor stack named `defaultStack` is used. The `defaultStack` is defined in the `struts-default.xml` using `<default-interceptor-ref name="defaultStack" />` element.

3.3.2 Custom Interceptor

The developer can create a custom Interceptor class by extending the class from the Interceptor class. The class should define the following three methods: `init()`, `destroy()`, and `intercept()`. The `init()` and `destroy()` methods are used for initializing the interceptor and cleaning of the interceptor respectively. They are called once when the Struts 2 framework is initializing and when the framework is shutting down respectively. Interceptors are used across requests and needs to be thread-safe specially the `intercept()` method.

The description of these methods are as follows:

- `void destroy()` - This method is used to clean up resources allocated by the interceptor.
- `void init()` - This method is called at the time of intercept creation but before the request processing using intercept, and to initialize any resource needed by the Interceptor.
- `String intercept(ActionInvocation invocation)` - This method allows the Interceptor to intercept processing and to do some processing before and/or after the processing `ActionInvocation`.

Code Snippet 7 shows a simple example on creating an interceptor for the Struts 2 Web application.

Code Snippet 7:

```
public class MyFirstInterceptor implements Interceptor{
  @Override
  public void destroy() {
  }
  @Override
  public void init() {
  }
```

```
@Override
 public String intercept(ActionInvocation actionInvocation) throws
Exception {
   String startInterceptor=" Start Interceptor 1";
   System.out.println(startInterceptor);
   String result=actionInvocation.invoke();
   String endInterceptor=" End Interceptor 1";
   System.out.println(endInterceptor);
   return result;
  }
}
```

Code Snippet 7 defines three methods. The `init()` method will be invoked only once and will initialize the interceptor. The `intercept()` method defines the processing logic to be invoked at each request. Here, it returns `String` value, so the result View page will be invoked. The `intercept()` method can also return `invoke()` method which will result in the invocation of next interceptor configured in the action. Finally, the `destroy()` method is invoked. The `destroy()` method is invoked only once and destroys the interceptor.

Code Snippet 8 creates the `Action` class.

Code Snippet 8:

```
public class MyAction extends ActionSupport{
  public String execute()
  {
   System.out.println("In Action");
   return SUCCESS;
  }
}
```

Code Snippet 8 shows the action class implementing the `execute()` method.

Code Snippet 9 shows the configuration of custom interceptor in `struts.xml` file.

Code Snippet 9:

```
. . .
<struts>
 . . .
 <package name="default" extends="struts-default" namespace="/">
 <interceptors>
    <interceptor name="myfirstInterceptor"
     class="com.example.MyFirstInterceptor" />
    . . .
 </interceptors>
<action name="Action1" class="com.example.MyAction">
   <interceptor-ref name="defaultStack"/>
   <interceptor-ref name="myfirstInterceptor"/>
   <result name="success">Welcome.jsp</result>
   <result name="input">login.jsp</result>
 </action>
 </package>
</struts>
```

Code Snippet 9 uses the `interceptor-ref` element as the sub-element of action. It specifies that the built-in interceptors defined in the `defaultStack` and `myfirstInteceptor` are applied for the mapped action.

Code Snippet 10 shows the `index.jsp` page that call the appropriate action.

Code Snippet 10:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Performed Action</title>
</head>
<body bgColor="lightBlue">
 <s:form action="Action1">
  <s:submit value="For calling Interceptor" align="center"/>
 </s:form>
</body>
</html>
```

Code Snippet 10 uses the `<s:form>` tag to create the form. The `action` attribute is assigned with the URL, `Action1`, which is a URL of the `MyAction` class.

The output of the application is as follows:

```
Start Interceptor 1
Start Interceptor 2
In Action
End Interceptor 2
End Interceptor 1
```

3.4 Struts 2 Tag Library

Struts 2 framework uses a set of tags for data reference. Struts 2 Tags are mostly used to create links to other Web resources in the local application. Before using the Struts 2 tags, the developer must have the property taglib declaration at the top of the page. The statement, `<%@ taglib prefix="s" uri="/struts-tags" %>` assigns the 's' prefix by which the Struts 2 tags will be identified.

The Struts 2 Tags is divided into two types: Struts Generic Tags and UI Tags.

3.4.1 Struts Generic Tags

The Struts generic tags control the execution flow when pages are rendered and also they extract data in and out of the ValueStack.

Struts Generic Tags are divided into **Control Tags** and **Data Tags**.

3.4.2 Control Tags

The control tags are used to control the flow of page execution. The most commonly used control tags are `if`, `elseIf`, `else`, `append`, `merge`, and `iterator`.

- **Iterator Tag**

  The `iterator` tag is used to loop over collection of objects. It can iterate over any Collection object, Map, Enumeration or Iterator, and array object.

  Table 3.4 lists the attributes of `iterator` tag.

Attribute	Description
Value	Specifies the object which is to be looped.
Status	It is optional, and if specified, then an `IteratorStatus` object is placed in the action context.

Table 3.4: Iterator Tag Attributes

- **If and else Tag**

  The `if` and `else` tag is similar to if-else control structure provided in languages. Table 3.5 lists the attributes of `if` and `else` tag.

Attribute	Description
Test	It contains the Boolean expression which is evaluated and tested. It returns either true or false.

Table 3.5: If and else Tag Attributes

Code Snippet 11 shows the Action class for accepting country.

Code Snippet 11:

```
public class CountryAction extends ActionSupport {
private String country;
public String getCountry() {
  return country;
}
public void setCountry(String country) {
  this.country = country;
}
@Override
public String execute() throws Exception {
  return SUCCESS;
}
}
```

Code Snippet 12 shows the `index.jsp` page which displays message to the user based on the condition.

Code Snippet 12:

```
<%@taglib uri="/struts-tags" prefix="s"%>
<html>
 . . .
<body>
 <h2>If-else-if tag example</h2>
 <hr>
```

```
<form action="countryAction" method="post">
    Select Country : <select name="country">
      <option value="France">France</option>
      <option value="Nepal">Nepal</option>
      <option value="Russia">Russia</option>
      <option value="China">China</option>
      <option value="USA">USA</option>
    </select><br> <input type="submit">
  </form>
  <hr>
  <s:if test="country!=null">
    <s:if test="country=='Russia'">
      <s:property value="country" /> is the selected
country.
    </s:if>
    <s:elseif test="country=='USA'">
      <s:property value="country" /> is the selected
country.
    </s:elseif>
    <s:elseif test="country=='China' or country=='Nepal'">
      <s:property value="country" /> is the selected
country.
</s:elseif>
    <s:else>
   <s:property value="country" /> is not the selected
country.
  </s:else>
  </s:if>
  <hr>
  <a href="index.jsp">Select Country Again</a>
</body>
</html>
```

Code Snippet 12 displays the form with the selection list. Then, depending on the selected option, a message is displayed to the user, based on the evaluation of the condition.

3.4.3 Data Tags

Data manipulation or creation is done with the help of Data tags. The most commonly used Data tags are action tag, include tag, bean tag, date tag, param tag, property tag, push tag, set tag, text tag, and url tag.

- **Action Tag**

  Action tag helps the users to call actions directly from a JSP page by specifying the action name and an optional namespace. The `executeResult` parameter must be specified in a program, otherwise any result processor defined for this action in `struts.xml` will be ignored.

  Code Snippet 13 shows the implementation of the `Action` class for `Action` tag.

Code Snippet 13:

```
/*
 * Action class
 */
public class ActionTagAction extends ActionSupport {


public String execute() throws Exception {
  return "done";
}


public String doDefault() throws Exception {
  ServletActionContext.getRequest().
   setAttribute("stringByAction", "This is a String put in by the
action's doDefault()");
   return "done";
 }
}
```

```
<!—struts.xml -->

 ....

 <action name=”actionTagAction1”  class=”example.testing.
ActionTagAction”>

  <result name=”done”>success.jsp</result>

 </action>

 <action name=”actionTagAction2” class=”example.testing.
ActionTagAction” method=”default”>

  <result name=”done”>success.jsp</result>

 </action>

 ....


<!-- actiontag.jsp page -->

<div>The following action tag will execute result and include it in
this page</div>

<br />

<s:action name=”actionTagAction” executeResult=”true” />

<br />

<div>The following action tag will do the same as mentioned
earlier, but invokes method specialMethod in action</div>

<br />

<s:action name=”actionTagAction!specialMethod”
executeResult=”true” />

<br />

<div>The following action tag will not execute result, but put a
String in request scope under an id “stringByAction” which will be
retrieved using property tag</div>

<s:action name=”actionTagAction!default”
executeResult=”false” />

<s:property value=”#attr.stringByAction” />
```

Code Snippet 13 creates action with two methods namely, `execute()` and `doDefault()`. Then, the mapping of the action is done in the `struts.xml` and the `actiontag.jsp` page that invokes methods from the action.

Include Tag

Include tag is used to include a JSP file in another JSP page.

Code Snippet 14 shows the use of `include` tag.

Code Snippet 14:

```
...
<-- First Syntax -->
<include value="AptechJsp.jsp">
<-- Second Syntax -->
<include value="AptechJsp.jsp">
<param name="param1" value="value2">
<param name="param2" value="value2">
  </include>
<-- Third Syntax -->
<include value="AptechJsp.jsp">
<param name="param1">value1</param>
<param name="param2">value2</param>
</include>
...
```

Code Snippet 14 shows the use of `<include>` tag on the page. The `value` attribute specifies the name of the Servlet and jsp page to be included.

Bean Tag

Bean tag represents a class that applies to JavaBeans specification. Bean tag contains a number of param elements to set any mutator methods on the class.

Code Snippet 15 shows an example for `bean` tag.

Code Snippet 15:

```
...
<bean name="org.apache.struts2.util.number" var="number">
<param name="first" value="100">
<param name="last" value="125">
</bean>
...
```

- **Date Tag**

Date tag helps to modify a date. User can use a custom format or can use the predefined format in the properties file.

Code Snippet 16 shows an example for `date` tag.

Code Snippet 16:

```
<s:date name="person.birthday" format="dd/MM/yyyy" />
<s:date name="person.birthday" format="%{getText('some.i18n.
key')}" />
<s:date name="person.birthday" nice="true" />
<s:date name="person.birthday" />
```

- **Param Tag**

Param tag is used to parameterize other tags. Param tag has two parameters. First, `name` (String) shows the name of the parameter and `value` (Object) shows the value of the parameter.

Code Snippet 17 shows an example for `param` tag.

Code Snippet 17:

```
<pre>
<ui:component>
<ui:param name="key"  value="[0]"/>
<ui:param name="value" value="[1]"/>
<ui:param name="context" value="[2]"/>
</ui:component>
</pre>
```

- **Property Tag**

Property tag gets the property of a value, which will default to the top of the stack if none is specified.

Code Snippet 18 shows the use of `property` tag.

Code Snippet 18:

```
<s:push value="myBean">
 <!-- Example 1: -->
 <s:property value="myBeanProperty" />

 <!-- Example 2: -->TextUtils
 <s:property value="myBeanProperty" default="a default value"
/>
</s:push>
```

- **Push Tag**

Push tag pushes value on stack for easier usage.

Code Snippet 19 shows the use of `push` tag.

Code Snippet 19:

```
...
<push value="employee">
<property value="userName1">
<property value="userName2">
</push>
...
```

- **Set Tag**

Set tag allocates a value to a variable in a more definite scope.

Code Snippet 20 shows an example for `set` tag.

Code Snippet 20:

```
...
<set name="Aptech" value="environment.name">
<property value="Aptech">
```

- **url Tag**

  url tag is used to create a URL.

  Code Snippet 21 shows an example for `url` tag.

Code Snippet 21:

```
<-- Example 1 -->
<s:url value="example.action">
  <s:param name="id" value="%{selected}" />
</s:url>
<-- Example 2 -->
<s:url action="myexample">
  <s:param name="id" value="%{selected}" />
</s:url>
<-- Example 3-->
<s:url includeParams="get">
  <s:param name="id" value="%{'22'}" />
</s:url>
```

3.4.4 UI Tags

Struts UI tags displays the data on the HTML page and uses the data from ValueStack or from Data tags. Struts UI tags are divided into three types: Form tags, Non-Form tags, and Ajax tags.

- **Form Tag**

  The most commonly used Form tags are checkbox, checkboxlist, combobox, doubleselect, head, file, form, hidden, label, and password. These tags provides user interface for the Struts Web applications.

  Form tags are categorized into three types, namely, Simple UI tags, Group UI tags, and Select UI tags.

Code Snippet 22 shows the use of `Form` tags.

Code Snippet 22:

```
<%@ taglib prefix="s" uri="/struts-tags"%>
. . .
<body>
 <s:div>Email Form</s:div>
 <s:text name="Please fill in the form :" />
 <s:form action="hello" method="post" enctype="multipart/form-data">
 <s:hidden name="secret" value="secretvalue"/>
 <s:textfield key="email.from" name="from" />
 <s:password key="email.password" name="password" />
 <s:textfield key="email.to" name="to" />
 <s:textfield key="email.subject" name="subject" />
 <s:textarea key="email.body" name="email.body" />
 <s:label for="attachment" value="Attachment"/>
 <s:file name="attachment" accept="text/html,text/plain" />
 <s:token />
 <s:submit key="submit" />
 </s:form>
</body>
</html>
```

Non-Form Tag

The most commonly used Non-Form tags are actionerror, actionmessage, component, div, and fielderror.

- **Actionerror** - Actionerror is used to send the error feedback message to user.
- **Actionmessage** - Actionmessage is used to send information feedback message to use.
- **Component** - Component tag renders custom UI widget using the specified templates.

- **Div** - Div tag generates an HTML div that loads its content using an ajax call, through the jQuery framework.
- **Fielderror** - Fielderror tag renders field errors if they exists.

Ajax Tag

Ajax tag is the new tag implemented in Struts 2 framework. In Struts 2, DOJO framework is used for the Ajax tag implementation. Commonly used Ajax tags are autocompleter, bind, head, div, submit, tree, and treenode.

To use Ajax tags, you add the tag library in the JSP page as, `<%@ taglib prefix="sx" uri="/struts-dojo-tags" %>` to JSP page.

The head tag included on the page, can be configured for performance or debugging purposes.

Some of the Ajax tags are as follows:

- **autocompleter**

  The autocomplete tag is a combo box that can autocomplete text entered on the input box. If an action is used to populate the autocompleter, the output of the action must be a well formed JSON string. The demonstration for `AutoCompleteAction` class and the `autocomplete.jsp` is shown in Code Snippet 23.

Code Snippet 23:

```
//Create the ActionClass - AutoCompleteAction.java


public class AutoCompleteAction implements Action
{
 public ArrayList<String> country = new
ArrayList<String>();
 public String countryName;


  public String execute()
  {
    populateCountry();
    return SUCCESS;
  }
```

```
public void populateCountry()
   {
      country.add("Australia");
      country.add("England");
      country.add("India");
      country.add("West Indies");
      country.add("New Zealand");
      country.add("Pakistan");
      country.add("Bangladesh");
      country.add("South Africa");
      country.add("Sri Lanka");
      country.add("Zimbabwe");
   }
...
}
<!-- JSP Page autoComplete.jsp -->
...
<%@taglib uri="/struts-dojo-tags" prefix="sx"%>
<%@taglib uri="/struts-tags" prefix="s"%>
...
<body>
<h3>Auto complete Dropdown | Textbox</h3>
 <s:form action="displayCountry">
 <sx:autocompleter name="countryName" list="country"
showDownArrow="false" label="Countries"/>
<s:submit />
</s:form>
</body>
</html>
```

Code Snippet 23 populates an autocompleter box from the names of the countries specified in the action.

- **bind**

The bind tag generates event listeners for multiple events on multiple sources, making an asynchronous request to the specified `href`, and updating multiple targets.

Code Snippet 24 shows the use for `bind` tag.

Code Snippet 24:

```
<img id="loadingImage" src="images/loadingAnimation.gif"
style="display:none"/>

<s:div id="parentDiv">

 <s:form action="actionName">

  <s:submit id="btn" />

  <sx:bind src="btn" events="onclick" targets="parentDiv"
showLoadingText="false" indicator="loadingImage"/>

 </s:form>

</s:div>
```

In Code Snippet 24, the bug in IE6/IE7 does not allow to use the targets attribute with a parent Div, because div's content are overwritten with the tag's `loadingText` resulting in an "undefined" message in the content, instead of the result of the request.

One possible alternative is to set `showLoadingText="false"` and set the indicator attribute to an element showing the desired loading text or image (outside the div).

- **Div**

Div tag generates an HTML div that loads its content using an XMLHttpRequest call, through the Dojo framework. When the '`updateFreq`' is set, the built in timer will start automatically and reload the div content with the value of '`updateFreq`' as the refresh period (in milliseconds).

- **Submit**

Submit tag renders a submit button that can submit a form asynchronously. The **submit** tag have three different types of rendering:

  - `input`: renders as html `<input type="submit"...>`
  - `image`: renders as html `<input type="image"...>`
  - `button`: renders as html `<button type="submit"...>`

- **tree and treenode**

The `tree` and `treenode` tag render a tree node within a tree widget with AJAX support. The tree and `treenode` of the two combinations are used depending on the

requirement like the tree is needed to be constructed dynamically or statically.

`tree`: This is a tree widget with AJAX support. Normally, this tag uses the '`id`' attribute. The '`id`' attribute is required, if the '`selectedNotifyTopic`' or the '`href`' attribute is going to be used.

`treenode`: This is a tree node which renders a tree node within a tree widget with AJAX support.

Check Your Progress

1. _____ class is responsible for the execution of the action including the sequential invocation of the Interceptor stack.

(A)	ActionInvocation	(C)	Action
(B)	ActionContext	(D)	ActionSupport

2. _____ method implements the code to handle interception.

(A)	`String intercept (ActionInvocation action)`	(C)	`Collection getIncludes()`
(B)	`Collection getExcludes()`	(D)	`void setExcludes(Collection excludes)`

3. _____ displays the data on the HTML page and uses the data from value stack.

(A)	Data Tags	(C)	Control tags
(B)	UI Tags	(D)	Non-Form tags

4. _____ generates event listeners for multiple events on multiple sources, making an asynchronous request to the specified href, and updating multiple targets.

(A)	bind	(C)	push
(B)	uri	(D)	property

5. _____ gets the property of a value, which will default to the top of the stack if none is specified.

(A)	set	(C)	div
(B)	form	(D)	property

Answer

1.	A
2.	A
3.	B
4.	A
5.	D

Summary

- Interceptor is an important feature introduced in the Struts 2 framework. Interceptors sit between controller and action.
- There are a set of built-in Interceptors provided by the Struts 2 framework. These built-in Interceptors can be used to provide the required functionalities to action.
- A custom Interceptor class can be created by implementing the com.opensymphony.xwork2.interceptor.Interceptor interface or by extending the com.opensymphony.xwork2.interceptor.AbstractInterceptor class.
- ActionInvocation class is responsible for the execution of the action including the sequential invocation of the Interceptor stack.
- The Interceptors can be specified using any one of the following approaches:
  - By declaring all the Interceptors in the struts.xml configuration file.
  - By declaring all the Interceptors that are used for the Web application in an XML file.
- The struts-default.xml file contains the definitions of all Interceptors and Interceptor stacks.
- Struts 2 framework uses a set of tags for data reference. Struts generic tags controls the execution flow when pages are rendered and it extracts data.
- Struts Control tags control the flow of page execution and data manipulation is done with the help of Data tags.
- Struts UI tags displays the data on the HTML page and uses the data from value stack.

04

Struts 2 - OGNL, Validation, and Internationalization

Welcome to the Session, **Struts 2 - OGNL, Validation, and Internationalization**.

In this session, you will learn about Object Graph Navigation Language (OGNL). The session explains the relation between ValueStack and OGNL. It explains the use of OGNL as Expression Language (EL) and Data Type Converter. Further, the session explains how to perform validation using Struts 2 Validator framework. Finally, the session explains how to make the Struts 2 application global using internationalization.

In this Session, you will learn to:

- Explain the use of ValueStack
- Explain OGNL expression language
- Explain the flow of data in and out of the Struts 2 framework
- Explain the use of converters and their types
- Explain Validator framework in Struts 2
- Explain different types of validators present in the Validator framework
- Explain the Implementation of validation framework in Struts 2
- Explain internationalization
- Explain the use of i18N interceptor and resource bundle in internationalization
- Explain how to create the Struts 2 Web application with internationalization

Session
04

Struts 2 - OGNL, Validation, and Internationalization

4.1 Introduction

ValueStack is a storage area that holds the data associated with the processing of a request. It is a stack of objects in the Struts 2 framework. All the core components interact with it to provide access to context information as well as elements of the execution environment.

The ValueStack is made up of the following objects:

- **Temporary Objects**: Requires temporary storage during request processing. For example, the iteration value of a variable when an array is being looped in a JSP page.
- **Model Object**: Represents a place on the stack before the action is being executed.
- **Action Object**: Represents the action that is being currently executed.
- **Named Object**: Represents any object that is identified by an identifier. It can be developer created or existing such as `#application`, `#session`, `#request`, `#attr`, and `#parameters`. Each of these existing named objects corresponds to an equivalent HTTP scoped object collection.

Normally, in a stack, you push or pop objects. However, with ValueStack you search for or evaluate a particular expression using Object Graph Navigation Language (OGNL) syntax. OGNL is a powerful expression language that allows to assign and manipulate the data present on the ValueStack.

The developer can access the ValueStack by using the JSP tags. A ValueStack returns the value of a property in the order it stacks the property. In other words, when the ValueStack is queried for a property value, it returns the value provided the property exists in the stack. If the queried property does not exist at the top of the stack then it queries the next element and continues until the last element in the stack is reached. If two properties have the same name, then ValueStack returns the value of the highest object in the ValueStack.

4.1.1 ValueStack and OGNL

OGNL allows the developer to refer and manipulate the data present on the ValueStack. It helps in data transfer and type conversion between data forms and action class. It provides a mechanism to navigate object graphs using dot notation and evaluate expression. The expression can also invoke methods on the objects that are being retrieved.

For ValueStack, OGNL expression is tested at each level and a result is returned after evaluation of the expression. If the expression cannot be evaluated, the next level is tested. A null value is returned when all the levels have been tested and result cannot be evaluated.

4.2 OGNL

OGNL is integrated into the Struts 2 framework to provide data transfer and type conversion. It is an expression and binding language used for getting and setting the properties of the Java objects.

In other words, it acts as a glue between the frameworks string-based input and output and the Java-based internal processing. It is a binding language between the GUI elements and the model objects.

OGNL can be used as follows:

- A binding language between GUI elements such as text fields, check boxes, radio button, and so on to model objects.
- A Type Converter to convert values from one type to another. For example, from String to numeric.
- A data source language to map between table columns and a Swing TableModel.
- A binding language between Web components and the underlying model objects.

It does this by moving data from the request parameter into the action's JavaBean properties and back from action's Java bean properties out into the rendering HTML pages.

OGNL is an open source framework from Apache Commons project. Using OGNL, you can set and get properties from Java Beans. Struts 2 introduced a unique feature of automatic type conversion between form data values and JavaBean properties. This feature of Struts 2 is implemented by OGNL Expression Language (EL).

OGNL EL can refer many of the built-in objects present in a JSP page. It is the default expression language that is used to refer to data from various sections of the framework in a consistent manner. Struts 2 follows the MVC design pattern. The View component, which is responsible for displaying the model and other object in a JSP page, uses OGNL EL.

OGNL expressions consist of elements which are mostly properties. These properties can represent many things depending on the object being accessed. Normally, these property names resolve to JavaBeans properties that conform to getter/setter method pattern.

The core unit of OGNL expression is the navigation chain, which consists of different parts. Following are the three different OGNL expression parts:

- Property name, such as **name** in a JavaBean.
- Method call such as `toString()` which returns the current object in a string format.
- Array indices such as **`employee[0]`**, which returns the first value of the current object.

OGNL expressions are evaluated in the context of current object. Thus, while evaluating the expression chain, the previous link in the chain acts as the current object for the next one. For example, consider the following expression and its order of evaluation:

```
Name.toCharArray()[0].numericValue.toString()
```

Following steps are performed to evaluate this expression:

1. The value stored in the name property of the object is extracted.
2. The `toCharArray()` method is invoked on the resulting `String`.
3. The first character from the resulting character array is extracted.

4. The numeric value is obtained from the character by invoking the `getNumericValue()` method of the `Character` class.

5. The `toString()` method is finally invoked on the Integer object to obtain the final result of the expression as `String`.

The two most important parts of OGNL are expression language and type converters.

4.2.1 Expression Language

OGNL's expression language is used in the form input field name and in JSP tags. In both these cases, there is a binding between the Java side data properties and strings in the text based view layer.

Code Snippet 1 shows the use of OGNL expression in Struts 2 tag.

Code Snippet 1:

```
...
<h5>Congratulations! You have successfully created your login </h5>
<h3>The <s:property value="loginName" /> Login Details </h3>
...
```

In the given code, a congratulation message is displayed after successful creation of the login page. The OGNL expression language is specified within the double quotes of the `value` attribute.

The `<s:property>` tag of Struts 2 takes the value from the property of one of the Java objects and writes it into the HTML in place of the tag. Thus, the `<s:property>` tag residing in the results page obtains the value from the property named **loginName** present in the Java environment.

OGNL expression is also used in an HTML form as the destination for data transfer. While tags help you to pull data from these properties, the `param` Interceptor helps to move data from the request parameter to these properties. Hence, it can be said that expression languages help to use simple syntax for referring to objects residing in Java environment.

As mentioned earlier, OGNL can also be used to invoke methods on Java objects that it can access. Thus, it simplifies the process of accessing data.

4.2.2 Type Converter

Even in a simple Web application, a data type conversion is made from the Java type of the property referred by the OGNL expression language to the string format of the HTML output. Thus, always an appropriate conversion of data takes place when the data moves in and out of the Java environment and HTML. Struts 2 framework provides built-in OGNL type converters.

Figure 4.1 shows how data moves in and out of the Struts 2 framework and, in addition, helps to bind and convert the data.

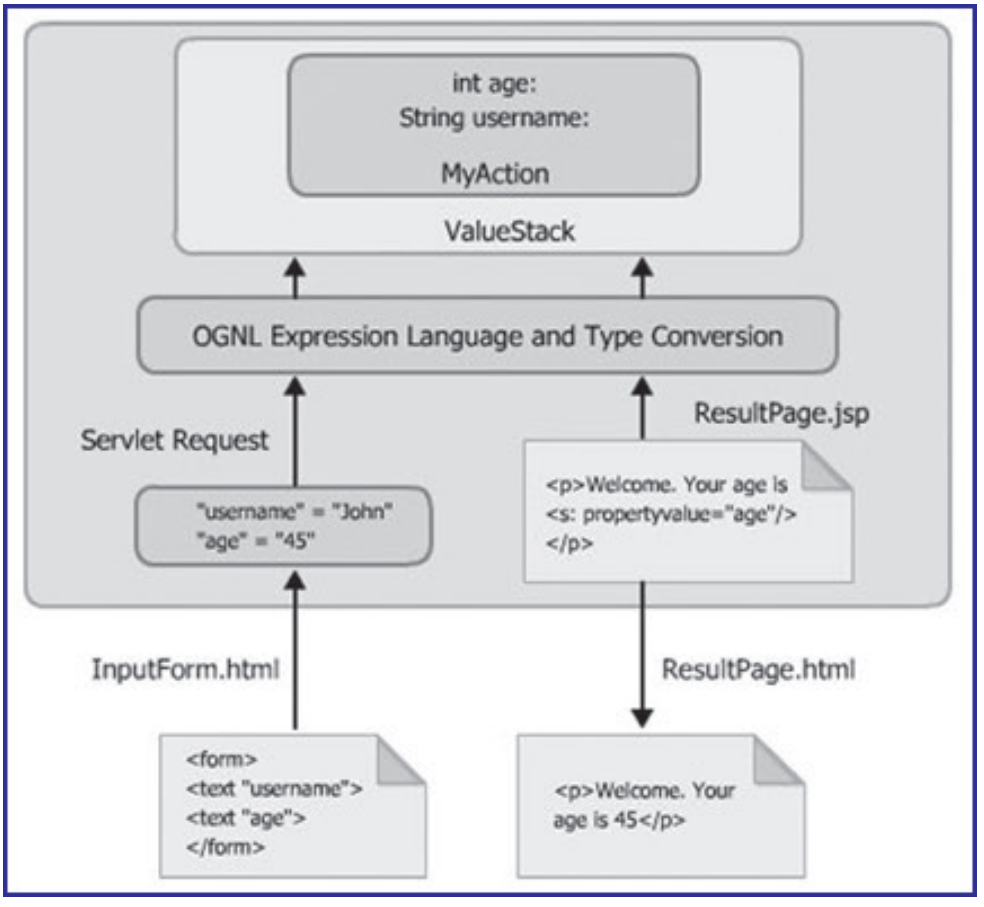


Figure 4.1: OGNL Framework for Transferring Data

The process starts with the user entering data into an HTML input form and submitting the request and ends with response that comes back to the user.

The flow of data in and out of the Struts 2 framework is explained as follows:

- **Data Input**

  The form, **`InputForm.html`** displays two input fields, which are valid HTML expressions. The field names are valid OGNL expression. Once the value is entered, the form is submitted to the framework. The request parameters are received by the framework as an `HttpServletRequest` object and are stored as name/value pairs. The name/value pair exists in a string format.

  The `param` interceptor transfers this data from the request object to ValueStack because the action object is present in the `ValueStack`. The most important activity that takes place before transferring the data is mapping of the property name with actual property on the ValueStack. The `params` interceptor interprets the parameter name as an OGNL expression to locate the correct destination property for the value.

The OGNL resolves its expression against a default object, which is a type of virtual object created by the ValueStack.

ValueStack holds the objects and their properties. In other words, ValueStack exposes the properties of the object it contains. Therefore, when the expression **age** is evaluated as an OGNL expression against the ValueStack, it checks whether the ValueStack has a **user** property. If the property is present, then it checks whether the property named **user** has an **age** property. Each of the properties is separated by a period. If the age property exists in the ValueStack, the data needs to be moved.

Thus, the data is moved to the property by invoking the setter method of the property. However, since the value is in string format, it needs to be converted to the appropriate Java data type, that is, `int`. In order to do this conversion, OGNL type converters come into action and check whether this type conversion, that is, string to `int` is possible. If it is possible, then the type converter converts and sets the value on the object named **user**.

- **Data Out**

Once the data has been processed by the action, the result will be invoked, which will provide a new result view of the application to the user. During data processing of the request, the object remains on the ValueStack. When the rendering process of the result starts, the result will access the ValueStack using the OGNL expression language tags. The tags will retrieve the value from the ValueStack by referring to each individual values with OGNL expressions. Finally, the **Result.jsp** page displays the result by retrieving the value stored in the property using the `property` tag. The value is converted again from the Java type of the property on the ValueStack to a string, so that it can be rendered as an HTML output. Thus, the integer data type is converted back to `String` data type.

4.2.3 Built-in Type Converters

Struts 2 framework provides built-in converters for converting an HTTP string data type to any of the Java data types. The different data types in Java that are supported by built-in converters are as follows:

- `String` - Represents a string data type.
- `boolean/Boolean` - Converts true and false strings to primitive or object version of Boolean data type.
- `char/Character` - Converts string to primitive or object version of character data type.
- `int/Integer` - Converts string to primitive or object version of integer data type.
- `float/Float` - Converts string to primitive or object version of float data type.
- `long/Long` - Converts string to primitive or object version of long data type.
- `double/Double` - Converts string to primitive or object version of double data type.
- `Date` - Converts string to SHORT format of current locale.
- `Arrays` - Converts each string element to the array's type.

- `Lists` - Populated with strings.
- `Maps` - Populated with strings.

Thus, the framework will look for a converter of a particular type when it locates a Java property targeted by a given OGNL expression. The setter method provided in the action class can change from **`setId(String Id)`** to **`setId(int Id)`**.

The developer is no longer required to perform conversions for each value as the developer can now use the value with correct data type that has been set on the action. The type converter does the data type conversion between the string-based HTTP form and the strictly typed Java objects.

4.2.4 Mapping Form Field Names to Properties

For automatic type conversion to take place, the developer needs to link the form field names with the Java properties of the action class and vice versa.

This is a two-step process that are as follows:

1. Developer writes the OGNL expression for the name attribute of the form field.
2. Developer creates the properties in Java code (Action class) that will receive the data.

The different types of data type conversions are discussed in detail in the following section:

- **Primitive and Wrapper Class**

  The built-in conversion between the Java primitive and wrapper classes and OGNL expressions in the form fields are quite simple.

  Code Snippet 2 shows the use of OGNL expressions in the form fields.

Code Snippet 2:

```
<h3>Registration Field</h3>
<s:form action="Register">
<s:textfield label="Username" name="username" />
<s:password label="Password" name="password" />
<s:textfield label="Enter your age...!" name="age" />
<s:textfield label="Enter your birthday.(mm/dd/yy)"
name="birthdate"
/>
<s:submit/>
</s:form>
...
```

In the given code, each of the input field is an OGNL expression. These expressions are resolved with respect to the action objects present in the ValueStack. The action objects are placed on the ValueStack when the processing of request starts. In this case, the **`Register`** action defines the JavaBeans property.

Code Snippet 3 shows the code in **`Register.java`**.

Code Snippet 3:

```
...
public class Register extends ActionSupport {
 private String username;
 private String password;
 private String portfolioName;
 private Double age;
 private Date birthdate;
@Override
 public String execute(){
 return SUCCESS;
 }
public String getPassword() {
 return password;
 }
public void setPassword(String password) {
 this.password = password;
 }
public String getUsername() {
 return username;
 }
public void setUsername(String username) {
 this.username = username;
 }
```

```
public Double getAge() {
 return age;
}
public void setAge(Double age) {
 this.age = age;
}
public Date getBirthdate() {
 return birthday;
}
public void setBirthdate(Date birthday) {
 this.birthday = birthday;
}
}
...
```

In the given code, when the `Submit` button on the form is clicked, the request is sent to the framework as a name-value pair. The framework places the `Action` object in the ValueStack. The **`name`** in the form input field is associated with the string value entered. The **`name`** is an OGNL expression, which is used to locate the target property on the ValueStack.

For example, the OGNL expression `<s:textfield label="Birthdate" name="birthdate" />` is resolved in Java using the `getRegister().getBirthdate();`.

OGNL finds that the **birthdate** property is of `Date` data type in Java. Therefore, the OGNL converts the property from string to `Date` using the `String` to `Date` converter and sets the converted value on the property. A conversion exception is thrown when the incoming string does not represent a valid instance of the primitive data type. The exception is raised when it is unable to bind the HTTP string value to Java data type.

Handling Multiple Values

Multiple parameter values coming from a request can be mapped to a single parameter name. A form can submit multiple values using a single parameter name. These multiple values can be mapped to Arrays, Lists, or Maps.

- **Arrays**

  Struts 2 provide support for converting data to Java arrays.

Code Snippet 4 shows the use of the array in the struts form.

Code Snippet 4:

```
...
<s:form action="ArraysDataTransferTest">
<s:textfield name="marks" label="Marks"/>
<s:textfield name="marks" label="Marks"/>
<s:textfield name="marks" label="Marks"/>
<s:textfield name="names[0]" label="names"/>
<s:textfield name="names[1]" label="names"/>
<s:textfield name="names[2]" label="names"/>
<s:submit/>
</s:form>
...
```

In the given code, there are two arrays named **marks** and **names**. Thus, the form submits data to two different array properties. The two array properties named **marks** and **names** will receive data from the first and the last three form fields, respectively. These properties should be present on the ValueStack.

In the given code, the form displays two ways for accessing arrays with OGNL expressions. Thus, the form submits four request parameters. The first three fields have the same name and thus it will result in the single request parameter with three values. The second set of fields has distinct request parameter with a single value mapped to it.

Table 4.1 displays the request parameter name with its value.

RequestParameter Name	Parameter Value
Marks	75, 65, 55
name[0]	Angel
name[1]	Jessica
name[2]	John

Table 4.1: Request Parameter Names and Values

Code Snippet 5 shows the implementation of the properties in the action class that will receive the data.

Code Snippet 5:

```
...
private int[] marks;
public int[] getMarks() {
 return marks;
}
public void setMarks(int[] marks)
 { this.marks = marks; }

private String[] names = new String[10];

public String[] getNames() {
 return names;
}
public void setNames(String[] names)
 { this.names = names; }
...
```

In the given code, arrays are displayed using the getter and setter methods. OGNL, on the other hand, handles the indexing details. The framework, when it transfers the **marks** property, first searches for the **marks** property and resolves the mapping of the property with the request parameter. The request parameter named **marks** stores three values in it as an array.

The **marks** property present in the `Action` class is also an array and thus the data transfer between the request parameter and the property is simple and easy.

The text input field uses the same name to submit multiple values. The **marks** property is of `int` data type and OGNL executes the type conversion for each element of the array. Since the array was created by the framework, there was no need to initialize the array.

On the other hand, for the second group of statements, the framework handles three individual request parameters. The Servlet API sees them as three unique names. When the framework provides these names to OGNL, they are correctly interpreted as specific elements in an array. The framework transfers and converts the form data into action properties.

- **Lists**

Struts 2 framework supports the conversion of sets of request parameter to properties of various collection types such as Lists. List is used in the same manner as an array, that is, either it can be indexed or the name itself can be used without indexing. By default, the request parameter value will be stored in the List property as strings.

One can either specify the type of the element or accept the default type for a List. The points to be considered before using Lists are as follows:

- Lists need not be initialized.
- Elements in a List will be treated as String objects if the type is not specified.

If the type of the element needs to be specified for List, then OGNL needs to be informed. This is done with the help of properties file. To specify the element type for properties on your `Action` object, you create a properties file according to the naming convention as shown in figure 4.2.

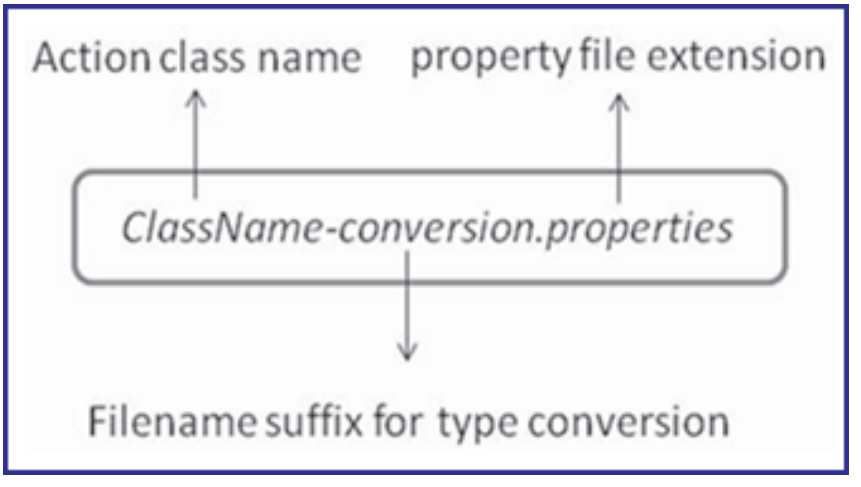


Figure 4.2: File Naming Convention

For example,

```
Element-weights=java.lang.Double
```

Code Snippet 6 shows the JSP page containing the weights property, which will be used as a List in the action class.

Code Snippet 6:

```
...
<s:textfield name="weights[0]" label="weights"/>
<s:textfield name="weights[1]" label="weights"/>
<s:textfield name="weights[2]" label="weights"/>
...
```

The property on the Action object does not change.

Code Snippet 7 shows the use of the `List` property in the `Action` class.

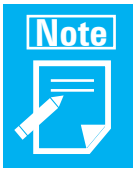
Code Snippet 7:

```
...
private List weights;

public List getWeights() {
 return weights;
}

public void setWeights(List weight) {
 this.weights = weight;

}
...
```



If you are using generics to specify the Collection type, then the properties file configuration need not be used. With type specific List conversion, you cannot pre-initialize the List.

- **Maps**

Struts 2 also support the conversion of a set of values from the request parameter to a `Map` property. `Map` stores the value in a key-value pair. You can specify the data types for the key-value pair in the `Map`.

Code Snippet 8 shows the implementation of the `Map` property in a JSP page.

Code Snippet 8:

```
...
<s:textfield name="maidenNames.mary" label="Maiden Name"/>
<s:textfield name="maidenNames.jane" label="Maiden Name"/>
<s:textfield name="maidenNames.hellen" label="Maiden Name"/>
<s:textfield name="maidenNames['beth']" label="Maiden
Name"/>
<s:textfield name="maidenNames['sharon']" label="Maiden
Name"/>
<s:textfield name="maidenNames['martha']" label="Maiden
Name"/>
...
```

In the given code, the specified key-value pair is of String data type. The key-value pair can either be specified using the dot-notation expression or in an array format. The **`maidenNames`** is the key and the different values such as **`mary`**, **`jane`**, and so on are the values. This form submits the data to a Map property named **`maidenNames`** in the action class.

Code Snippet 9 shows the implementation of the `Map` property in the Java action page.

Code Snippet 9:

```
private Map maidenNames;
public Map getMaidenNames(){
 return maidenNames;
}
public void setMaidenNames (Map maidenNames) {
 this.maidenNames=maidenNames;
}
...
```

In the given code, the getter and the setter method for the map property have been defined. You do not have to initialize the Map object explicitly.

4.2.5 Type Conversion Annotations

The type conversion annotations are used for generics defined in Maps and Collections. An annotation is present for each option that is configured in the ***-conversion.properties** file. To use conversion type annotation, annotate the class or interface with the custom annotation.

Some of the type conversion annotations are described in table 4.2.

Annotation Name	Description
`CreateIfNull`	Checks if the new element should be created if it currently does not exist in the list or map.
`Type Conversion`	Determines the converter class to use. For collections, a rule of PROPERTY, MAP, KEY, KEY_PROPERTY, or ELEMENT can be used to specify which part to be converted.
`Element`	Used for generic type to specify the element type for Collection types and Map values.
`Key`	Used for generic types to specify the key type for Map keys.
`KeyProperty`	Used for generic type to specify the key property name value.

Table 4.2: Conversion Annotations

4.3 Validation

All Web applications require user input which range from a simple username and password to data entered in a complex form. Values entered in the application are required to be validated before they are stored in the database.

User input needs to be validated so as to ensure that correct data has been entered as required by the application. The task of validating user input is complex. One of the main features of the Struts 2 framework is its built-in validation support. Thus, this helps to reduce the code specification and makes the tasks of validating the user input easy and simple.

The Struts 2 framework supports both server and client side validation. It provides a wide range of validation rules and pre-defined validators which can be applied to form fields for validations. Developers can also create custom validators based on the validation requirements.

For using pre-defined validators, no initial configuration is required. Validation is implemented by the framework using validation Interceptor which is configured in the default Interceptor stack.

Three layers present in the Validation Framework are Domain Data, Validation MetaData, and Validators.

- **Domain Data**

  To perform validation, you need to have some data. The data entered by the user using a browser exists as properties in an action class in Struts 2. For example, in the **Login Form**, a user is required to enter username and password. This will exist as properties named username and password in an action class. These properties will store the data that your action will work with when it begins execution.

- **Validation Metadata**

  This is the middle layer component that lies between the validators and the data properties. This component will associate the individual data properties with the validators which are used for verifying the correctness of the values during runtime. There is no limit to the number of validators that can be associated with each property.

  Mapping of validators to data properties can be done using Java annotations or XML files.

- **Validators**

  The actual work of validating the values present in the data properties is done by the validators. Validators can be defined as a reusable component that contains the logic for performing fine-grained validations. For example, to ensure that the **username** property has a string value and is not empty, this property can be mapped to the `requiredstring` validator. The **`password`** property can be mapped to `requiredstring` and `stringlength` validator.

  Thus, the validators have to be associated with the properties either through an XML file or through Java annotations. When the validation executes, each property is validated by the set of validators with which it is associated.

4.3.1 Working of a Validator

An action class extends `ActionSupport` class which implements the `Validateable` and `ValidationAware` interfaces of the `com.opensymphony.xwork2` package. The `validate()` method in the `Validateable` interface contains the validation code whereas the `ValidationAware` interfaces contains methods for storing the error messages generated when validation of a property fails.

The validator interfaces work with workflow Interceptor. Once the user submits the input form, the workflow Interceptor is executed. It first checks whether `Validateable` interface is implemented by the action. If it is implemented, then the `validate()` method is invoked by the workflow Interceptor. On encountering an error, the validator creates an error message and adds it to one of the methods of the `ValidationAware` interface. Once the execution of the validate() method is completed, the workflow Interceptor will invoke the `hasErrors()` method of the `ValidationAware` interface. This method is invoked to check for the existence of any error messages. In case of error existence, the workflow Interceptor stops further execution of the action and returns the input result.

This result takes the user back to the input form that was submitted.

Code Snippet 10 shows the sequence of Interceptors from the `defaultStack` as defined in `struts-default.xml` file.

Code Snippet 10:

```
...
<interceptor-ref name="params" />
<interceptor-ref name="conversionError" />
<interceptor-ref name="validation" />
<interceptor-ref name="workflow" />
...
```

In the given code, params and `conversionError` Interceptors are responsible for transferring and converting the values in the HTTP request parameter to correct Java types. Both these Interceptors are fired before the validation Interceptors are fired.

The `validation` Interceptor is used for making an entry into the validation framework. When this Interceptor fires, all the validations are checked that have been defined in the validation metadata. If conversion and validation errors are encountered, then they are stored or added in the methods of the `ValidationAware` interface.

The `workflow` Interceptor works in two phases. In the first phase, it invokes the `validate()` method if present in the current action and performs the basic validation. In the second phase, it checks for errors in the `hasErrors()` method of the `ValidationAware` interface. If there are no errors, then the control passes to the rest of the action invocation process.

When the `defaultStack` is used, both the `validation` and the `workflow` Interceptors fire every time. This means that both the forms of validation can be used. Thus, first the `validation` Interceptor runs all the validations that are defined with the validation framework metadata. Next, when the workflow Interceptor runs, it checks whether the `validate()` method is implemented in the action class. The `validate()` method can contain some extra validation code.

4.3.2 Validation Scope

Two types of validators present in the Validator Framework are Field Validators and Non-Field Validators.

- **Field Validators**

  These validators operate on a single field accessible through an action. A validator is generic and can perform validations in the full action context involving more than one field in validation rule. Validations can be defined on each field. There can be more than one validator on a single field.

Following are the steps for applying pre-defined validators:

- Create an XML Configuration file
- Name it as per the guidelines
- Place it in the correct directory

The name of the validator file is derived from the name of the class that implements the action for which the validation rules apply. The naming rule for the XML validation metadata file is **ActionClass-validation.xml**. Thus, if the action class is named as `Registration` then the validation XML file is named as **Registration-validation.xml**. The file is placed in the package directory structure next to the action class. Each `<field-validator>` is executed in the order of definition.

Code Snippet 11 shows the implementation of validators in an XML file.

Code Snippet 11:

```
...
<!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="username">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
```

```
<field name="password">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
<field name="age">
<field-validator type="required">
<message key="errors.required" />
<!-- <message> You must enter age </message> -->
</field-validator>
<field-validator type="int">
<param name="min">1</param>
<param name="max">100</param>
<message key="errors.range"/>
</field-validator>
</field>
...
</validators>
```

In the given code, the `doctype` element includes all the validation XML files. The `<validators>` element includes the declarations of the individual validators that should run when this action is invoked.

The first field declared is the **`username`** field. Once the field element is placed, you need to put `<field-validator>` elements inside that field element to declare which validators should validate this piece of data. For example, in the case of the **username**, you declare only one validator, the **`requiredstring`** validator which verifies that the string that has been submitted is not an empty string. If the **`username`** string does not pass this verification, then a message is displayed to the user when the workflow Interceptor sends the user back to the input form.

The `<type>` element refers to the name of the registered Validator. The `<message>` element contains the text of this message that the user will see when validation error takes place. You can also use OGNL expression to make the message dynamic. You need not always specify a message with the `<message>` element. The message can be specified in a resource bundle. The `<message>` element will then have the attribute

named key which will be set to a value. The value in the `key` is used to retrieve the message from the properties file resources.

The `<param>` element takes parameters to configure the behavior. The element accepts name and value attributes to set arbitrary parameters into the `Validator` instance. They are resolved at runtime against the ValueStack. The **username** present in the stack is pulled to customize the message that the user sees when the user returns to the input page. The `stringlength` validator is placed on the `ValueStack` and thus its properties are also exposed.

- **Non-Field Validators**

  These validators are not restricted to a single specific field. They apply to the whole action and often contain checks that apply to more than one field values. The `expression` validator is the only non-field validator available in the built-in validators. Complex validation logic can be written in the expression validator using OGNL EL.

4.3.3 Built-in Validators

The framework provides built-in validators to handle the validation requirements. The validators are present in XWork jar file which contains an XML file that declares all the built-in validators.

Table 4.3 displays a list of built-in validators.

Validator Name	Parameters	Description
required		Checks that the value is not null.
requiredstring	trim	Checks that the value is not null and is not empty. The default value for the attribute trim is true indicating that it trims white space.
stringlength	trim, minlength, maxlength	Checks that the length of the string is within the parameters specified by minlength and maxlength. If length is not specified then checks are not made. In minlength is not specified then an empty string would pass validation.
int	min, max	Checks that the integer value is within the value specified by min and max.
double	minInclusive, maxInclusive, minExclusive, maxExclusive	Checks that the double value is within the inclusive or exclusive specified parameters.
date	min, max	Checks that the date value is within the specified date as specified by min and max attribute. Format of the date value is MM/DD/YYYY.

Validator Name	Parameters	Description
email	-	Checks the format of the email address.
url		Checks the URL format.
fieldexpression	expression()	Evaluates the OGNL expression against the current ValueStack and returns true or false depending on the result of evaluation.
regex	expression(), caseSensitive, trim	Checks that the string confirms to the regular expression.
expression	expression()	Used at action level and is same as fieldexpression.

Table 4.3: Validators – Field and Action

4.3.4 Using a Validator

In order to demonstrate the use of Validators, a `NewsletterRegistration` application is created in NetBeans. Modify the default **index.jsp** page to contain a link to **NewsLetterRegistration.jsp** page. Create a registration form **NewsLetterRegistration.jsp** that accepts user details such as `name`, `password`, `age`, `email`, and `telephone number`.

Following are the steps:

- **Creation of the JSP page**

  Code Snippet 12 shows the code for **NewsLetterRegistration.jsp** page.

Code Snippet 12:

```
...
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>NewsLetter Registration Page</title>
</head>
<body>
<h1>News Letter Registration Page!</h1>
<s:form action="NewsLetter" method="post" >
```

```
<s:textfield name="username" key="username" size="20" />
<s:textfield name="password" key="password" size="20" />
<s:textfield name="age" key="age" size="20" />
<s:textfield name="email" key="email" size="20" />
<s:textfield name="telephone" key="telephone" size="20" />
<s:submit />
</s:form>
</body>
</html>
...
```

In the given code, five text fields have been defined to accept the username, password, age, email address, and telephone number, respectively. The newsletter registration page accepts the registration detail from the users.

Creation of an Action class

Next, create an action that will receive the data from the registration form such as username, password, and so on.

Code Snippet 13 shows the code for the action form.

Code Snippet 13:

```
...
public class NewsLetter extends ActionSupport {
public String execute() throws Exception {
return SUCCESS;
}
/**
* Provide default value for Message property.
*/
public static final String MESSAGE = "HelloWorld.message";
```

```
/**
* Field for Message property.
*/
private String message;
* Return Message property.
* @return Message property
*/
public String getMessage() {
return message;
}
public void setMessage(String message) {
this.message = message;
}
private String username;
private String password;
private String telephone;
private String email;
private int age;
public String getUsername() {
return username;
}
public void setUsername(String userName) {
this.username = userName;
}
```

```
public String getPassword() {
return password;
}
public String getTelephone() {
return telephone;
}
public void setTelephone(String telephone) {
this.telephone = telephone;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
}
...
```

In the given code, the action class receives the data from the JSP page and sets the property value.

- **Declaration of the validation metadata with ActionClass-validations.xml**

Next, create an XML file that will establish association between the data properties and the validators that contain the desired logic. This will be the metadata file of the user-defined action class.

Code Snippet 14 shows the NewsLetter action's validation metadata file, **NewsLetter-validation.xml**.

Code Snippet 14:

```
...
<!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="username">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
<field name="password">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
<field name="age">
<field-validator type="required">
<message key="errors.required" />
<!-- <message> You must enter age </message> -->
</field-validator>
```

```
<field-validator type="int">
<param name="min">1</param>
<param name="max">100</param>
<message key="errors.range"/>
</field-validator>
</field>
<field name="email">
<field-validator type="requiredstring">
<message key="errors.required" />
</field-validator>
<field-validator type="email">
<message key="errors.invalid" />
</field-validator>
</field>
<field name="telephone">
<field-validator type="requiredstring">
<message key="errors.required" />
</field-validator>
</field>
</validators>
...
```

In the given code, name of the file is derived from the name of the class that implements the action for which the validation rules apply. The file is placed in the package directory next to the action class.

The `doc` type element must be included in the validation XML file. The `<validator>` element contains all the declaration of the individual validators that should run when the action is invoked. Once the field element is declared for the data, the `<field-validator>` element is placed within the field element to specify which validators would validate this data. For example, for the field element **username**, the field validator, `requiredstring`, is used. The validator verifies that the string has been submitted and is not empty. If **`username`** does not pass this verification test then a message is displayed to the user and the workflow Interceptor sends the user back to the input form. The `<message>` element contains the text of the message.

The <message> element can also externalize the message itself in a resource bundle. In the code, the <message> element does not have text body. Instead, it sets the key attribute to a value to be used to look up the message using the TextProvider implementation provided by ActionSupport. In other words, the key attribute is used to look up for the message in the properties file.

Creation of Properties file

Finally, create a properties file that will contain the error message for the specified key attributes.

Code Snippet 15 shows the implementation of **properties** file.

Code Snippet 15:

```
...
HelloWorld.message=Welcome to Struts Validation ...
username=Username
age=Age
email=Email
telephone=Telephone
password=Password
errors.invalid=${getText(fieldName)} is invalid.
errors.required=${getText(fieldName)} is required.
errors.number=${getText(fieldName)} must be a number.
errors.range=${getText(fieldName)} is not in the range ${min} and
${max}.
```

In the given code, the different error messages are stored for each of the text fields to be validated. For example, if the requiredstring validator encounters a null value for the **username** property of the action class, the respective error message is retrieved and displayed.

Figure 4.3 displays the application startup page.

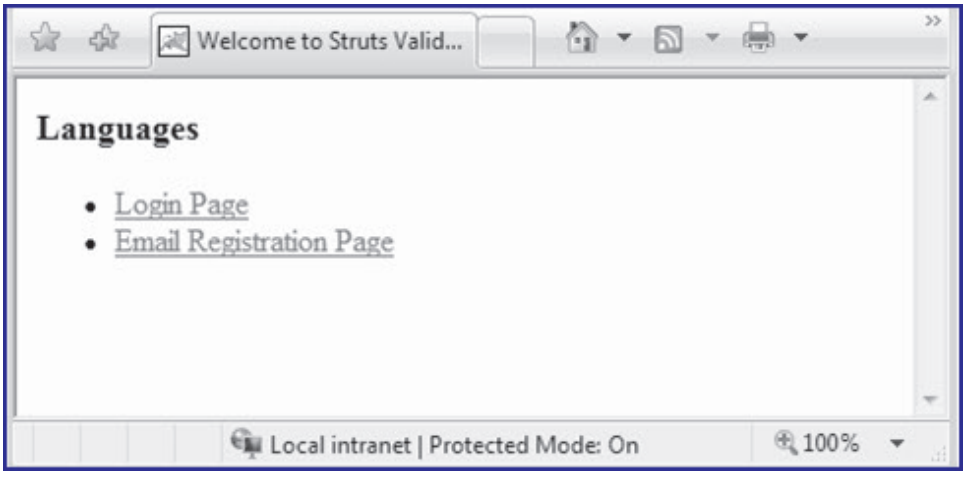


Figure 4.3: Application Startup Page

Figure 4.4 displays the error page showing the validation messages when the user clicks the Submit button without entering all the details.

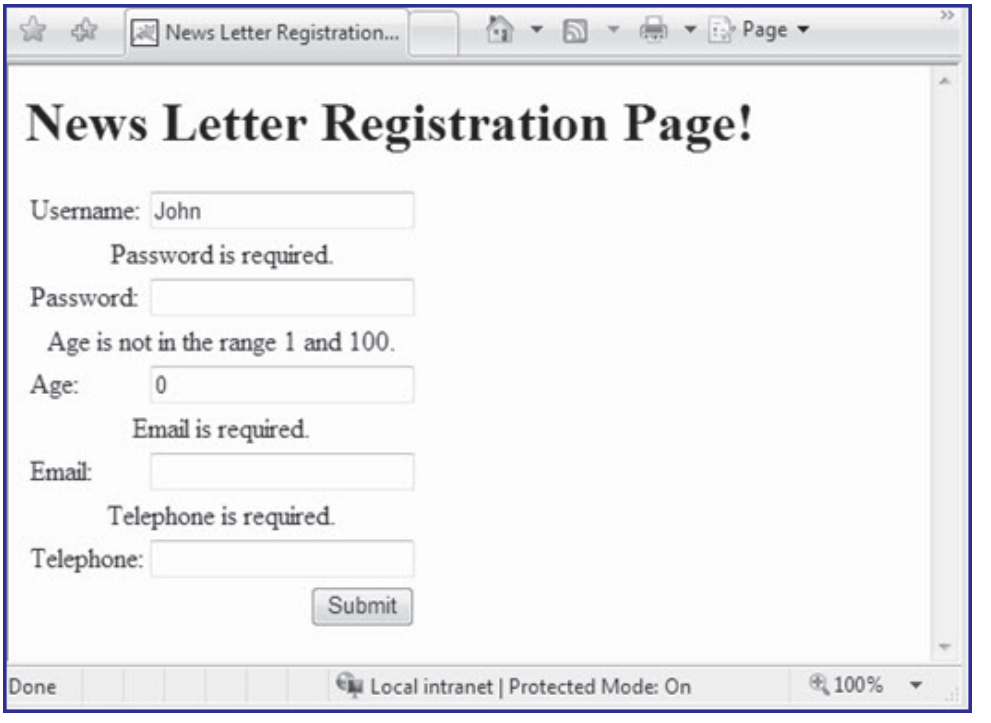


Figure 4.4: Error Page Showing Validation Messages

4.3.5 Creating a Custom Validator

A custom validator must implement a certain interface. In other words, it must implement the `Validator` or `FieldValidator` interface. These two interface present in the `com.opensymphony.xwork2.validator` package represents the field and non-field validators.

The Struts 2 framework provides some convenience classes that help to write custom validator classes easily. Thus, these custom validator classes normally extend either the `ValidatorSupport` or the `FieldValidatorSupport` classes present in the `com.opensymphony.xwork2.validator.validators` package.

Consider a scenario where the password validator makes the following validation checks on the password field:

- It must contain a letter either in uppercase, or lowercase
- It must contain digits from 0-9
- It must contain at least one single special character such as !@#$

Code Snippet 16 shows the implementation of the validator class, named **`PasswordIntegrityValidator`**.

Code Snippet 16:

```
...
public class PasswordIntegrityValidator extends
FieldValidatorSupport {


static Pattern digitPattern = Pattern.compile("[0-9]");

static Pattern letterPattern = Pattern.compile("[a-zA-Z]");

static Pattern specialCharsDefaultPattern = Pattern.compile(
"!@#$");


public void validate(Object object) throws ValidationException {

System.out.println("object being validated = " + object.
getClass().getName());

String fieldName = getFieldName();

String fieldValue = (String) getFieldValue(fieldName, object);

//trim the password in case some spaces were added

fieldValue = fieldValue.trim();

//check security level and do the integrity checks
```

```
Matcher digitMatcher = digitPattern.matcher(fieldValue);
Matcher letterMatcher = letterPattern.matcher(fieldValue);
Matcher specialCharacterMatcher;
if (getSpecialCharacters() != null){
Pattern specialPattern = Pattern.compile ("["
+ getSpecialCharacters() + "]");
specialCharacterMatcher = specialPattern.matcher(fieldValue);
}
else
{
specialCharacterMatcher = specialCharsDefaultPattern.matcher(
fieldValue);
}
if (!digitMatcher.find()) {
addFieldError(fieldName, object);
}else if (!letterMatcher.find()) {
addFieldError(fieldName, object);
}else if (!specialCharacterMatcher.find()) {
addFieldError(fieldName, object);
}
}
/* JavaBeans property to receive the parameter from the validator
* mapping. Special characters are a set of characters, at least
* one of which the password must contain.
*/
private String specialCharacters;
public String getSpecialCharacters() {
return specialCharacters;
}
```

```
}public void setSpecialCharacters(String securityLevel) {
this.specialCharacters = securityLevel;
}
}
...
```

In the given code, the class extends from the `FieldValidatorSupport` class. For non-field validator, the validator class must **`extends`** from the `ValidatorSupport` class. The validation logic is placed in the `validate()` method. This abstract method is the entry method defined by the `Validator` interface.

In this code, some static members are defined containing the definition of regular expressions patterns. A default set of special characters have also been defined. Next, two helper methods are used to retrieve the field value. The `validate()` method receives the object that is being validated. This object is our action because the object has been defined at the action level.

The value in the **`password`** field is retrieved using the two helper methods defined in the support class. They are the1 and the `getFieldName()` method. Finally, the matcher objects are created and the value in the **`password`** field is searched for the required characters. If the required characters are not found, then errors are generated and stored. The workflow Interceptor retrieves the error and diverts the user to the input page.

Code Snippet 17 shows the implementation of the custom validator named **RegisterUser-Validation.xml** file.

Code Snippet 17:

```
...
<!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="password">
<field-validator type="requiredstring">
<message >Password is required.</message>
</field-validator>
<field-validator type="stringlength">
<param name="maxLength">10</param>
```

```
<param name="minLength">6</param>
<message>Your password should be 6-10 characters.</message>
</field-validator>
<field-validator type="passwordintegrity">
<param name="specialCharacters">$!@#?</param>
<message>Your password must contain one letter, one number, and
one of the following "${specialCharacters}".</message>
</field-validator>
</field>
<field name="username">
<field-validator type="requiredstring">
<message >User name is required.</message>
</field-validator>
<field-validator type="stringlength">
<param name="maxLength">8</param>
<param name="minLength">5</param>
<message>While ${username} is a nice name, a valid username must
be between ${minLength} and ${maxLength} characters long. </
message>
</field-validator>
</field>
<field name="email">
<field-validator type="requiredstring">
<message>You must enter a value for email.</message>
</field-validator>
<field-validator type="email">
<message key="email.invalid"/>
</field-validator>
</field>
</validators>
...
```

In the given code, the built-in as well as the custom validator are used for the field named `password`. The **`password`** field is mapped to three validators. They are `required-string`, `stringlength`, and `passwordintegrity`. The `passwordintegrity` validator is the custom validator and is used just like the built-in validator. The `<field-validator>` element is used to declare the type. The logical name used is same as the one declared in the **validators.xml** file.

Next, the `specialCharacters` parameters are passed to indicate the set of required characters. This is received by the JavaBeans property that is exposed on the `passwordIntegrityValidator`. Finally, the message is passed which the user views when the password does not match the integrity test.

Code Snippet 18 shows the implementation of the `validators.xml` file.

Code Snippet 18:

```
. . .
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator Config 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-validator-config-1.0.dtd">
<validators>
<validator name="passwordintegrity" class="register.utils.PasswordIntegrityValidator"/>
</validators>
```

In the given code, **validators.xml** file declares the custom validator named **`PasswordIntegrityValidator`** as an available validator. This validator is referred using a logical name **`passwordintegrity`**. This logical name of the custom validator is mapped to the `password` field in the **RegisterUser-validation.xml** file.

4.3.6 Validation Annotation

For each of the XML configured validators, there is a corresponding annotation. There are annotations available, which define that a class will be using annotation-based validation. Validation Annotation is used for configuring custom validators and to group validators for a property or a class.

Table 4.4 describes the different validation annotations.

Annotation Name	XML Equivalent	Description
`RequiredFieldValidator`	Required	Ensures that the property is not null.

Annotation Name	XML Equivalent	Description
RequiredStringValidator	Requiredstring	Ensures that the String property is not null or empty.
StringLengthFieldValidator	stringlength	Checks that the String length is within the specific range.
IntRangeFieldValidator	Int	Checks that the integer property is within the specific range.
DateRangeFieldValidator	Date	Checks that the date property is within the specific range.
ExpressionValidator	Expression	Evaluates the OGNL expression using the ValueStack.
FieldExpressionValidator	Fieldexpression	Validates a field using the OGNL expression.
EmailValidator	Email	Ensures that the property is a valid email address.
UrlValidator	url	Ensures that the property is a valid URL.
RegexFieldValidator	Regex	Checks whether the property matches the regular expression.
Validation		Signifies that the class using annotation based validation can be used on interfaces or classes.
Validations		Groups multiple validations for a property or a class.

Table 4.4: Validation Annotation

4.4 Add-ins Internationalization in Struts 2

With globalization the business is no more constrained to a specific geographic region. The Internet has made it possible to transact business across the globe. The software developers nowadays are faced with a daunting task of designing an application that will serve a customer from around the world regardless of their location and native language.

Internationalization is the technique for developing applications that support multiple languages and data formats without the need to re-engineer the application.

By designing application ahead of time, developers need not write programming logic again when a new language or country is to be supported by the application.

Sometimes the term Internationalization is abbreviated as I18N, because there are 18 letters

between the first 'I' and the last 'N.'

Each language and country has a unique codes representing the language and country, respectively. These codes are assigned by International Organization for Standardization (ISO).

Table 4.5 shows the list of few country and language codes.

Country Name	Country Code	Language	Language Code
Italy	IT	Italian	it
China	CN	Chinese	zh
France	FR	French	fr
Germany	DE	German	de
Japan	JP	Japanese	ja
Spain	ES	Spanish	es
United States	US	English	en

Table 4.5: Country and Language Codes

Struts 2 supports internationalization through interceptors, resource bundles, and tag libraries, in the Web application.

- **i18n Interceptors**

  The i18n interceptors provide the support for multi-language support in Struts 2 Web application. It allows the developer to set the locale for the action. A locale is an identifier and represents the local details of the user to the application. The local can be created by `java.util.Locale class`. When a user session begins, the `Locale` object is sent in the HTTP request as parameter.

  The i18n interceptor is a part of the `defaultStack` interceptor stack. The developer inherits it by extending the package from the struts-default and does not require the explicit mapping with the Action class.

  The i18n interceptor provides two parameters that are as follows:

  - **parameterName** - It specifies the name of the HTTP request parameter. It is set to **request_locale** by default. For example, the default parameter for the request can be `login.action?request _ locale=en _ US`, which is the locale for US English. However, the developer can overwrite this parameter.
  - **attributeName** – It specifies the name of the session key to store the locale. The default value for the parameter is **WW_TRANS_I18N_LOCALE**.

- **Message Resource Bundles**

  These allow support for localization and internationalization by having application content placed into resource bundles instead of being hard coded in an application. These contents can then be retrieved using custom tags. Thus, these bundles can be used as central repository for contents that have multiple uses. In order to use resource bundles one needs to create a property file that contains key/value pairs.

`ResourceBundle` class is a central database for holding resources used by the application. An application simply requests for resource to `ResourceBundle` class using its name and locale. It facilitates in grouping a set of resources like error-messages, button labels for specific locale.

This class is an `abstract` class. The subclasses are `ListResourceBundle` and `PropertyResourceBundle`. The abstract `ListResourceBundle` class provides mechanism for using lists to store resources. The concrete and widely used `PropertyResourceBundle` class provides mechanism for using properties files to store resources. It requires that resource bundle properties files should be named using format `bundlename_language_country_variant.properties`. All the locale components may not be specified.

Struts support for internationalization is essentially centered on the `Message Resources` component. All the application's text is stored in this resource bundle. At the time of internationalizing the application, locale-specific versions of this resource bundle are created. The information in the resource bundle is made available to JSP's via bean library tag or as return objects of Action Class.

Resource bundles are searched in the following order:

- ActionClass.properties
- Interface.properties for every interface and sub-interface
- BaseClass.properties
- ModelDriven's model
- package.properties
- i18n message key hierarchy
- global resource properties

To use message resources, the steps to be followed are:

1. Create a properties file for the resources to be stored in resource bundle.
2. Save it with '`.properties`' extension. Struts application use `ApplicationResources.properties` as standard way to refer to this file.
3. Provide the locale-specific information in the filename. For example, to create a Spanish version, name the file as `Applicationresources _ es.properties`. The class `PropertyMessageResource` of Struts will transparently select the required locale-specific resource bundle. In case of unavailability, Struts will load the resource from the default resource bundle.
4. To facilitate access to `.properties` file, store it on the classpath of application. (For example, in /WEB-INF/classes/ directory). This is because Java's class loader is used by Struts to load the properties file.

5. Use <message-resources> element in the Struts configuration file to specify the location of file. The value of parameter attribute of `message-resources` element is set to the location of `.properties` file. The `key` attribute of element can be used to specify each additional resource bundles.

Tag Libraries

Struts provide several library tags to support internationalization. The attribute value of these tags specifies the keys to properties in a Message Resources bundle. Struts provides several ways to access the message resources such as `getText()` method, the text tag, key attribute of UI tags, and the i18n tag.

Code Snippet 19 shows the use of text tag to retrieve the message from the resource bundle.

Code Snippet 19:

```
<s:text name="some.key" />
```

4.4.1 Implementing Internationalization

Consider the development of login page with different languages.

Code Snippet 20 creates the common `index.jsp` page which will be displayed based on the selected language.

Code Snippet 20:

```
. . .
<%@ taglib prefix="s" uri="/struts-tags"%>
. . .
<body>
 <h1><s:text name="global.heading"/></h1>
 <<s:url id="indexEN" namespace="/" action="locale" >
  <s:param name="request_locale" >en</s:param>
 </s:url>
  <s:url id="indexFR" namespace="/" action="locale" >
  <s:param name="request_locale" >fr</s:param>
 </s:url>
 <s:a href="%{indexEN}" >English</s:a>
 <s:a href="%{indexFR}" >France</s:a>
```

```
<s:form action="login" method="post" namespace="/">
  <s:textfield name="name" key="global.name" size="20" />
  <s:textfield name="password" key="global.password" size="20" />
  <s:submit name="submit" key="global.submit" />
 </s:form>
</body>
...
```

In the given code, the Struts 2 text tag is used that it retrieves a `ResourceBundle` message based on the key passed into the `name` attribute.

Code Snippet 21 shows the different resource bundle properties created for different languages. The name of the properties file will be `global_languagecode.properties`.

Code Snippet 21:

```
# global.properties
global.name = Name
global.password = Password
global.submit = Submit
global.heading = Select Locale
global.success = Successfully authenticated
#global_fr.properties
global.name = Nom d'utilisateur
global.age = l'pâssws
global.submit = Soumettre des
global.heading = Sé lectionnez Local
global.success = Authentifié avec succès
```

Code Snippet 22 shows the configuration of resource bundle in the `struts.xml` file.

Code Snippet 22:

```
. . .
<struts>
 <constant name="struts.custom.i18n.resources" value="global" />

 <package name="example" extends="struts-default" namespace="/">
  <action name="login"
   class="com.example.LoginAction"
   method="execute">
   <result name="input">/index.jsp</result>
   <result name="success">/success.jsp</result>
  </action>

  <action name="locale"
   class="com.example.Locale" method="execute">
   <result name="success">/index.jsp</result>
  </action>
 </package>
</struts>
```

Code Snippet 23 shows the `LoginAction` and `Locale` class.

Code Snippet 23:

```
/*
 * LoginAction.java class
*/
public class LoginAction{
 private String name;
 private String password;
 public String execute()
```

```
 {
  return SUCCESS;
 }

 //getter and setter methods
 ...
}

/*
 * Locale class
*/

public class Locale extends ActionSupport{
 public String execute()
 {
  return SUCCESS;
 }
}
```

Figure 4.5 shows the output of the code.

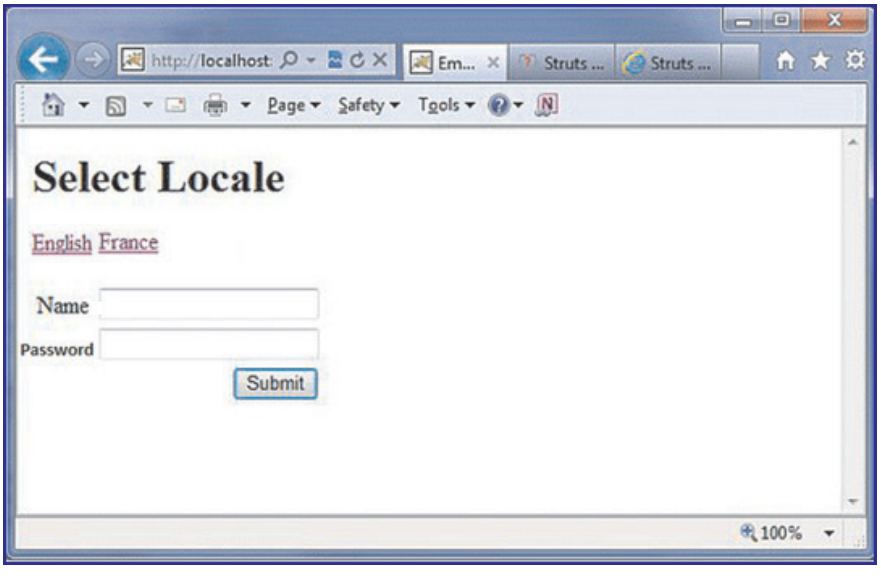


Figure 4.5: Output - Internationalization

4.5 Life Cycle Callback Annotations

Life cycle callback annotations are invoked at a specified time during the processing of an action. There are three life cycle callback annotations. To use these annotations, you have to specify the `AnnotationWorkflowInterceptor` to your Interceptor task.

- **`@Before`**

  This annotation marks an action method that should be executed before the main action method. This annotation is used at the method level. It helps in pre-processing of common tasks.

  Code Snippet 24 shows the use of the `@Before` annotation.

Code Snippet 24:

```
...
public class SampleAction extends ActionSupport {
@Before
public void isAuthorized() throws AuthenticationException {
// authorize request, throw exception if failed
}
public String execute() {
// perform secure action
return SUCCESS;
}
}
...
```

- **`@After`**

  This annotation marks an action that should be executed after the logic in the main action method and the result is executed but before the result is returned to the user. The return value is ignored. This annotation is applicable at the method level.

Code Snippet 25 shows the use of the `@After` annotation.

Code Snippet 25:

```
...
public class SampleAction extends ActionSupport {
@After
public void isValid() throws ValidationException {
// validate model object, throw exception if failed
}
public String execute() {
// perform action
return SUCCESS;
}
}
...
```

In the given code, the `isValid()` method will be executed after the `execute()` method.

- **`@BeforeResult`**

This annotation will be invoked after the method that executes the logic for the action, but before the result is executed. The return value is ignored. This annotation is applied at the method level.

Code Snippet 26 shows the use of the `@BeforeResult` annotation.

Code Snippet 26:

```
...
public class SampleAction extends ActionSupport {


@BeforeResult
public void isValid() throws ValidationException {
 //validate model object, throw exception if failed
}
public String execute() {
 //perform action
return SUCCESS;
}
}
...
```

In the given code, the `isValid()` method will be executed after the `execute()` method but before the execution of the result.

Check Your Progress

1. The binding language between the GUI elements and the model objects in Struts 2 framework is _____.

(A)	Type Converter	(C)	String
(B)	OGNL	(D)	Expression

2. Which of these statements is true and which is false?

Statement A: OGNL's expression language is used in the form input field name and in JSP tags.

Statement B: OGNL is a binding language between the GUI elements and the view objects.

(A)	Statement A is TRUE and Statement B is FALSE.	(C)	Both the statements are TRUE.
(B)	Statement A is FALSE and Statement B is TRUE.	(D)	Both the statements are FALSE.

3. Which of the following annotation is applied at the method level?

(A)	@BeforeResult	(C)	@Namespace
(B)	@After	(D)	@Before

4. The _____ annotation evaluates the OGNL expression using the ValueStack.

(A)	`ExpressionValidator`	(C)	`DateRangeValidator`
(B)	`RequiredStringValidator`	(D)	`RequiredFieldValidator`

Check Your Progress

5. Which of the following annotation ensures that the property cannot be null?

(A)	`RequiredField`	(C)	`Namespace`
(B)	`ResultField`	(D)	`RequiredFieldValidator`

6. Which of the following annotation is a valid Life cycle callback annotation?

(A)	`@Namespace`	(C)	`@ParentPackage`
(B)	`@Results`	(D)	`@Before`

Answer

1.	B
2.	A
3.	A
4.	A
5.	D
6.	D

Summary

- OGNL is integrated into the Struts 2 framework to provide data transfer and type conversion.
- OGNL's expression language is used in the form input field name and in JSP tags.
- Struts 2 framework provides built-in converters for converting an HTTP string data type to any of the Java data types.
- Two types of validators present in the Validator Framework are Field Validators and Non-Field Validators.
- One of the main features of the Struts 2 framework is its built-in validation support.
- The validate() method in the Validateable interface contains the validation code whereas the ValidationAware interfaces contains methods for storing the error messages generated when validation of a property fails.
- Two types of validators present in the Validator Framework are Field Validators and Non-Field Validators.
- Struts 2 Web application support internationalization.
- Struts 2 supports internationalization through interceptors, resource bundles, and tag libraries, in the Web application.
- Life cycle callback annotations are invoked at a specified time during the processing of an action.

Onlinevarsity
your e-way to learning

Visit

Frequently Asked Questions

@

www.onlinevarsity.com

05 Introduction to JavaServer Faces

Welcome to the Session, **Introduction to JavaServer Faces**.

This session covers the role of JavaServer Faces (JSF) as a User Interface (UI) framework. The session explains the architecture of JSF, which describe all the models contained in it such as UI Component model, Component Rendering model, Data Conversion model, Navigation model, and so on. Further, the session explains the life cycle of JSF application. It explains the development process of JSP applications and explains each component configured in the application. Finally, the session explains various JSF Tag libraries associated in JSF framework.

In this Session, you will learn to:

- Explain JSF framework
- Explain the components of JSF framework
- Explain the different types of configuration files used in JSF applications
- Describe JSF architecture and JSF lifecycle
- Explain the process of developing JSF application
- Explain UI components and the renderers in JSF
- Explain the concept of managed beans in JSF
- Explain basic tag, converter tag, and validator tag in JSF

Session
05

Introduction to JavaServer Faces

5.1 Introduction to JSF

Sun Microsystems provided two specifications for developing server-side components namely, Servlet and JSP.

Servlets help to generate dynamic contents, whereas JSP pages separate presentation from the application logic. However, both the technologies fail to provide separation of User Interface (UI) components from the model. Thus, developers are dependent on client specific UI for designing the Web pages. For example, for a Web client, HTML Web pages are designed. As HTML controls are static in nature, which means they are not powered with the dynamic functionalities of handling the data. Thus, developers access the data from the form controls using HTTP request and response objects.

Apart from static nature of HTML controls, there is a lack of event handling on the controls. For example, a view component in MVC architecture should generate events on the controls that can be handled. For example, Swing controls are developed with MVC architecture. When value changes in a UI component, the model gets updated based on the event fired on the control.

Some of the challenges faced by the developer, while designing UI controls for the clients of the Web application are as follows:

1. **Rich client controls**: Rich controls are based on the ability to query the model and updated the data with event-handling mechanism. Thus, Web applications often create their own customized mechanisms to store the state of the data on the Web server. They also had to code methods to implement event handling.

   For example, when a registration form is submitted, a programmer-written code extracts values for each form element and stores them in variables for further processing. For editing this data, values have to be reassigned to the form elements by retrieving them from the database into program variables and then reassigned to the form.

2. **Support for custom controls**: Sometimes, Web applications need to be customized with custom components, such as a grid viewer. The grid viewer components can be created to display data from database or query builder. However, it takes time, efforts, and expertise to build, test, and integrate such custom components.

   A need was felt by the developers for a development environment that would allow them to:

   - Implement custom UI components and add or remove them dynamically.
   - Provide UI support to various clients such as Web browser, PDA, mobile phones, and so on.
   - Use an event model such as Swing to trap events fired by components or controls in Web pages.

   In order to deal with the mentioned requirements, Sun Microsystems provided a server-side UI component framework named as JavaServer Faces (JSF) technology.

JSF is an open source, component based, event driven framework for building user interface for Web applications.

The main components of JSF technology are as follows:

- A well-defined programming model that provides various UI components and API to manage their state, event-handling, validation, data conversion, and so forth.
- JSP custom tag libraries for providing UI components on a JSP page and bounding the components to server-side objects.

In JSF, programmers work with UI components such as text boxes, radio buttons, and so on and write code for handling events such as button clicked rather than performing request–response processing. Thus, JSF can be thought of as 'Swing for server-side applications', helping rapid UI development to server-side Java.

5.1.1 JSF Framework

The main goal of JSF is to develop Web applications quickly and easily.

Figure 5.1 shows the high-level architecture of JSF application.

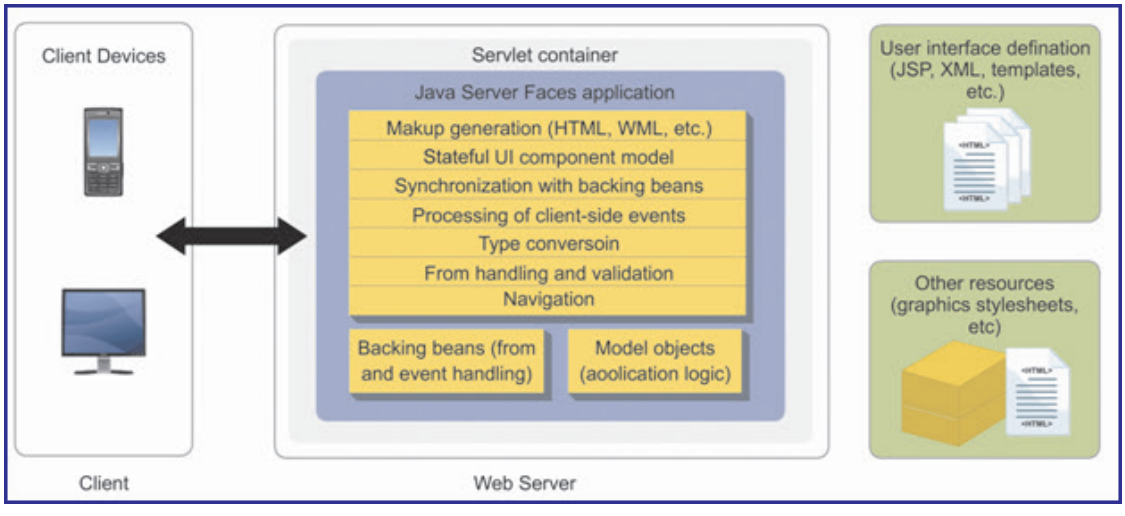


Figure 5.1: High-level Architecture Model of JSF Application

The component-based framework of JSF defines the support for standard UI components such as button, text fields, check boxes, and so on. It also supports third-party components. Components are based on event-driven programming model in which events are handled at the server-side. JSF supports the architecture for displaying same components to different types of clients such as Web browsers, PDAs, and mobiles. JSF applications are deployed at the server-side and are executed within a Web container.

Figure 5.2 shows the JSF APIs are built on top of Servlet API.

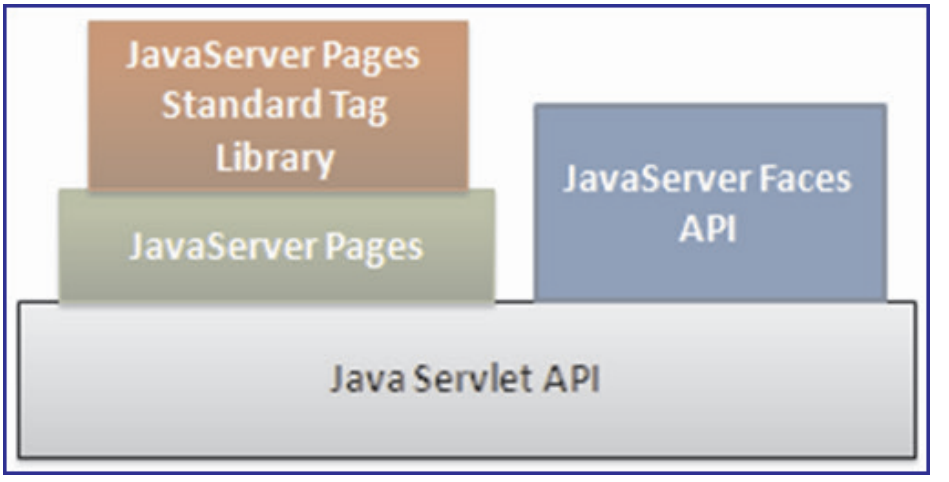


Figure 5.2: JSF API Built on Servlet API

In figure 5.2, the JSF layer reduces the need of using JSP for presentation. This enables the use of component classes for creating the presentations for different types of client devices.

5.1.2 Components of JSF

JSF is a Java based server-side user interface component framework for developing Web applications in Java. Each component of this component centric framework is described as follows:

- **Managed Beans**

  One of the critical function of the JSF framework is to separate the definition of UI component object from those objects that perform application specific processing and also saving the state of the data.

  In JSF, a UI component can be associated with a Java Bean which defines UI component's properties. It also defines methods that perform functions such as event handling, validation, and navigation processing associated with the component. These beans get instantiated at runtime using managed bean creation facility.

- **UI Components**

  UI Components are also called controls or components and focuses on interacting with the end user. One major issue in components for Web is that they need to be stateful objects. For example, in a Web application, if a form is filled with some errors and submitted, then it gets redisplayed with an error message. In such cases, all the form components should look the same and display the original values that were filled by the user. This means that the components must remember their state. A JSF UI component can handle this automatically.

  The JSF UIComponent classes specify the User Interface component's functionality. This includes storing the state of the component, maintaining reference to objects, and driving event handling and rendering of standard components.

- **Converter**

  JSF application may have association between a component and server-side data object which will be a Java Bean. Converter components basically convert form (UIComponent) data to Java objects for storing onto the model data and from model's Java object to presentation view.

  Converters ensure that the user's input is in a specified format as specified by the converter. JSF provides a set of standard Converter implementations as well as facility to customize the Converter implementations.

- **Validator**

  JSF supports data validation before the form (UIComponent) values update the object data. Similar to Converter model, the Validation model also defines a collection of standard classes for common data validations and a developer can also define custom validations.

- **Renderers**

  JSF component architecture separates the functional aspect of a UI component from the rendering of UI components. Separate classes called renderers handle the rendering process. Collection of renderer classes forms a render kit. The render kit defines mapping of component classes to component tags suitable for a specific client.

Some of the other core components of JSF deal with event handling, message handling, and navigation through JSP pages.

- **Events and listeners**

  JSF supports an event handling mechanism based on the model defined by JavaBean specification. Here, a component event gets represented by an instance of an event class and there is an event listener interface for every JSF component event. When a component is used by activating it such as 'button clicked', then an event is fired. JSF implementation then invokes the corresponding listener method to process the event.

- **Messages**

  Messages in JSF can display either information or errors. Following are the two types of error messages in JSF:

  - **Logical Error Messages**

    These error messages can be generated either due to errors in business logic or database errors or connection errors.

  - **Input Error Messages**

    These error messages can be generated due to user input errors such as empty field or invalid text and so on.

All the standard messages that are required by an application are defined as a resource in the Java resource bundle. Application developer can create their own resource bundle and it needs to be defined in **faces-config.xml** file.

- **Navigation**

  Page navigation means the ability to move from one page to another. JSF has a powerful page navigation system.

  It handles page navigation by allowing the developer to use a configuration file to specify and control the page navigation. Page navigation takes place when a user clicks either a command button or a command link.

  JSF provides a default action listener for such navigational event. These default action listeners know which page to present next because they are bound to page navigation rules in the applications configuration file.

5.1.3 Benefits of JSF

JSF simplifies the development of complex Web applications. Figure 5.3 depicts the benefits of JSF.

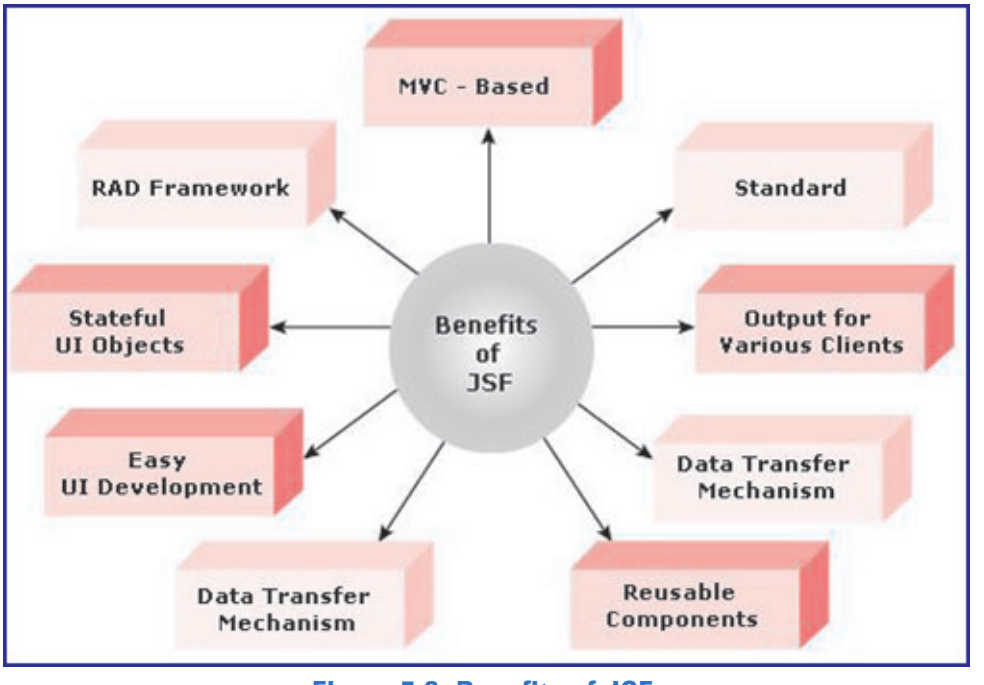


Figure 5.3: Benefits of JSF

The benefits of using JSF are as follows:

- **MVC-based**

  JSF as a framework encourages the use of MVC architecture to separate the presentation and the business logic. The presentation logic which may be a complex GUI may be combined into a single component which is easy to manage as compared to the presentation logic in Struts.

- **Output for Various Clients**

  JSF technology has JSF Custom tag library. This allows representation of components on a JSP page. However, since JSF APIs are layered directly on the Servlet API, it allows the usage of functionalities such as alternative presentation technique instead of JSP or generating output for various clients such as HTML or WML.

- **Reusable Components**

  The JSF framework allows creation and integration of third party components, thus promoting the reuse of components at the component level.

- **Easy UI Development**

  UI components can be created as reusable objects. A Renderer separates UI components functionality from its view. Thus, multiple look and feel renderers can be created and used to define different appearances of the same UI component for the same or different clients. As such JSF implementation comes with default render kit for HTML.

- **Stateful UI Objects**

  Unlike JSP, JSF applications can manage the user interface element objects as stateful objects. JSF framework brings traditional GUI framework such as environment for Web application development.

Some additional benefits of JSF are as follows:

- **Support for Data Transfer Mechanism**

  JSF implementation allows easy transfer of data between application and UI. On just specifying the destination and nature of the data, JSF implementation manages the data transfer.

- **Easy Event Handling Mechanism**

  JSF supports an easy way to manage event handling which is similar to the model used in other UI frameworks such as Java Swing.

- **RAD Framework for Web Applications**

  JSF allows the developers to visualize in terms of components, beans and their interactions, validating user inputs, handling events rather than only requests-responses, and markup. This kind of top view allows a better focus for application development.

5.1.4 Comparison between JSP and JSF

Table 5.1 shows the comparison between JSP and JSF.

Java Server Pages (JSP)	Java Server Faces (JSF)
Java based technology that is utilized to develop and design dynamic Web pages.	Web application, which is used to simplify development and integration of Web based UIs.
Does not support Validator, Conversion, and Ajax support.	Supports Validator, Conversion, and Ajax support.

Java Server Pages (JSP)	Java Server Faces (JSF)
No features such as managed beans and template-based component system.	Core features such as managed beans and template-based component system.
Not a request-driven MVC. Accessed by the dynamically generated pages such as XML or HTML.	Request-driven MVC.
Compiled within the server, not within a view template.	Interface within a view template.

Table 5.1: JSP and JSF Comparison

5.2 Introduction to JSF Versions

Since its first release, JSF have gone through major and minor version enhancements. JSF 1.0 was the initial specification released which combines the MVC design pattern with a powerful component-based UI framework. This simplifies the development of Web applications on Java EE platform. The JSF 1.2 release was introduced with many improvements and used on Java EE 5 platform.

The later version JSF 2.0 was introduced with the main focus of simplifying and building of UI components. In version 2.0, JSF is released in two minor version namely, JSF 2.1, and JSF 2.2.

The major change between JSF 1.0 and JSF 2.0 versions is the introduction of annotations for configuring JSF components, which makes development faster and easier. In addition, the shift in the View technology from JSP to Facelets.

5.3 Introduction to JSF Configuration Files

JSF applications use one or more XML descriptor files to configure applications. A developer needs to use minimum of two XML configuration files, while working with JSF. These two files, namely, `web.xml` and `faces-config.xml` are crucial to the flexibility and provide flexible coupling for the JSF architecture. The configuration files let the Java code to be easily shared between JSP pages.

5.3.1 web.xml File

This is the standard Web application configuration file. JSF requires the central configuration list web.xml in the directory `WEB-INF` of the application. `web.xml` lists bean resources and navigation rules. This is similar to other Web applications which are based on Servlets. The `web.xml` configuration file is located in the /`WEB-INF`/ directory of the Web application that is to be deployed. The Web server end of the application is configured by `web.xml`. In addition, to define the files types that may be accessed by users or the directories that can be retrieved, `web.xml` file can be used.

In `web.xml`, you must specify that a `FacesServlet` is responsible for handling JSF applications. The central controller of JSF application is `FacesServlet`, and it receives all requests for the JSF application. In addition, `FacesServlet` initializes the JSF components before the JSP is displayed.

Code Snippet 1 shows the configuration of `FacesServlet` in `web.xml` in JSF 2.0.

Code Snippet 1:

```
. . .
<Servlet>
 <Servlet-name>Faces Servlet</Servlet-name>
 <Servlet-class>javax.faces.webapp.FacesServlet</Servlet-class>
 <load-on-startup>1</load-on-startup>
</Servlet>

<!-- Servlet Mapping to URL pattern -->
 <Servlet-mapping>
  <Servlet-name>Faces Servlet</Servlet-name>
  <url-pattern>*.xhtml</url-pattern>
 </Servlet-mapping>
 <Servlet-mapping>
  <Servlet-name>Faces Servlet</Servlet-name>
  <url-pattern>*.jsf</url-pattern>
 </Servlet-mapping>
 <Servlet-mapping>
  <Servlet-name>Faces Servlet</Servlet-name>
  <url-pattern>*.faces</url-pattern>
 </Servlet-mapping>
</web-app>
. . .
```

In JSF 2.0, the `FacesServlet` can map to different URL patterns, such as `*.xhtml`, `*.jsf`, and `*.faces`. Also, note that while running JSF 2.0 on a Servlet 3.0 container, the `web.xml` file is optional. If `web.xml` is not present, then the `FacesServlet` is automatically mapped to the common URLs such as `/faces/*`, `*.jsf`, `*.faces`, and so on.

5.3.2 faces-config.xml File

The `faces-config.xml` file contains the configuration of the JSF application. It allows the configuration of the application, converters, validators, managed beans, and navigation. The `faces-config.xml` file also defines the behavior of the Faces Servlet.

While `web.xml` file is typically edited only when structural modifications are carried out on the application or at the beginning of a project, `faces-config.xml` changes along with the developmental progress of the application. In addition, `web.xml` generally comprises common configuration options, a `faces-config.xml` file tends to be more specific to a certain application. This is because, `faces-config.xml` file may contain, for example, navigation details and other configuration which are specific to the application.

Code Snippet 2 shows the faces-config.xml file created in the directory, `/WEB-INF` of a JSF application.

Code Snippet 2:

```
<?xml version="1.0"?>
<faces-config . . .version="2.2">
<!-- Configuring managed bean -->
<managed-bean>
<managed-bean-name>messageBean</managed-bean-name>
<managed-bean-class>JSF.SimpleController</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>

<!-- Configuring navigation -->
<navigation-rule>
<from-view-id>/starting-page.xhtml</from-view-id>
<navigation-case>
<from-outcome>return-value-1</from-outcome>
<to-view-id>/result-page-1.xhtml</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>return-value-2</from-outcome>
<to-view-id>/result-page-2.xhtml</to-view-id>
</navigation-case>
. . .
</navigation-rule>
</faces-config>
```

The `faces-config.xml` allows the developer to configure the elements of a JSF application. However, from JSF 2.0, this file is optional, as all the information about the elements can be configured through annotations.

5.4 Introduction to JSF Architecture

JSF is an MVC-based application framework. It provides a rich architecture for defining user interface components, managing their state on the server, and handling client-generated events. It also provides support for validating user input and controlling page navigation.

Figure 5.4 shows the main components of the JSF architecture and depicts the processing flow of a client request.

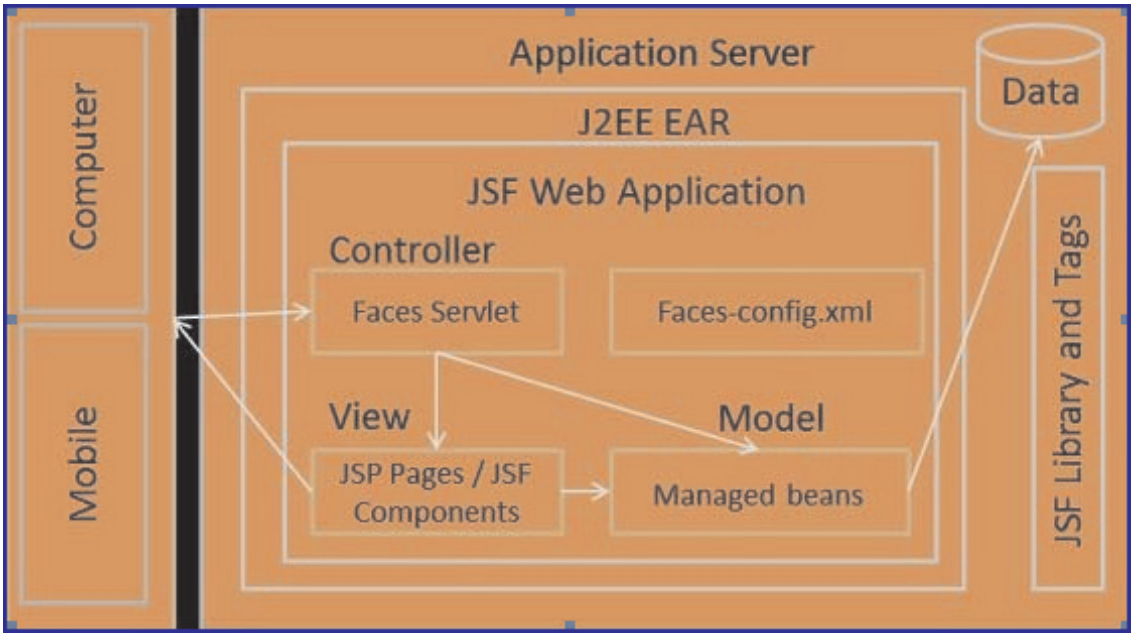


Figure 5.4: JSF Architecture

- **Controller**

  As shown in figure 5.4, the `FacesServlet` is the entry point for all requests for a JSF page. It initializes the resources needed by the framework to control the page's life cycle and subsequently, invokes the page to process the request. It serves as a Front Controller to the application.

- **View**

  The user interface components in the JSF page are represented on the server by a component tree. This is also referred to as a View. When a request is processed by the framework, this tree is either created for an initial page request or restored from its saved state (for subsequent page requests).

- **Model**

  A Backing Bean is a JavaBean that holds the data for a JSF Page's user interface components and implements methods that support their behavior. It includes logic to perform event handling, validation, and navigation control.

A Backing Bean usually invokes methods from a model object to perform business logic. JSF allows you to declare all the Backing Beans used by the page in the faces configuration file named as `face-config.xml`, so that they will be automatically instantiated by the Web container at application startup time. These beans are called Managed beans.

5.4.1 Introduction to JSF Life Cycle

The sequence of events that are fired when a request to a JSF page is made is known as JSF life cycle. The `FacesServlet` processes all the requests and carries out a series of steps (request processing life cycle) to prepare a response. The life cycle phases manage the input data, so that the Web developer does not need to write the code for processing the request manually.

Figure 5.5 depicts the JSF life cycle.

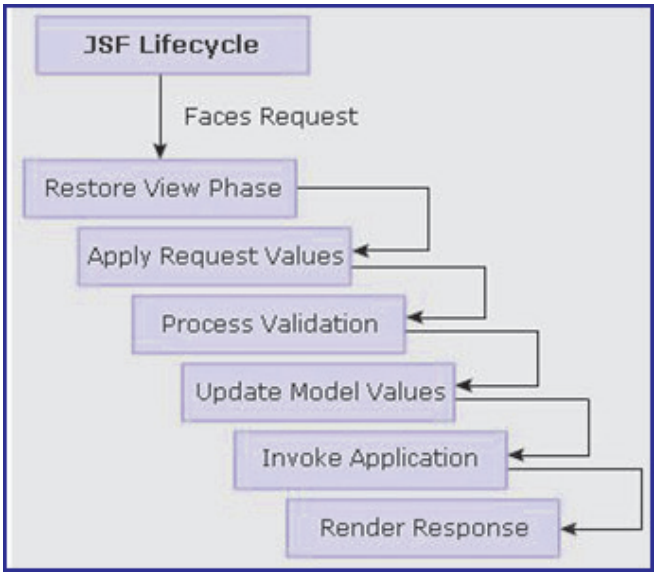


Figure 5.5: JSF Life Cycle

The first three phases of JSF life cycle are described as follows:

- **Restore View Phase**

  This phase starts when the application responds to a request coming from FacesServlet controller. For example, this request is received when a link or a button is clicked. When a request is received initially, the JavaServer Faces implementation creates a view tree for the page. A view tree is a server-side UI component tree representation providing the mirror representation of all UI elements present in a client. A new view is stored in the FacesContext parent object.

- **Apply Request Values Phase**

  In this phase, each component's state is retrieved. In other words, after restoring the view tree, request processing is performed. Each component in the view tree extracts the new value of requested parameter by using the component name-value pair of information.

The value is stored locally on the component. If some component on a page has its 'immediate' attribute set to true, then its validation, conversion, and event handling will be processed during 'Apply Request Values' phase.

- **Process Validations Phase**

  In this phase, values associated with each component is validated against the validators registered for this component, as per predefined validation rules. If the local value is found invalid, JSF adds an error message to the FacesContext instance, and the component is marked invalid. To mark the component invalid, the life cycle phase advances to the render response phase for displaying the error message.

- **Update Model Values**

  Post validation and conversion, the actual values of server-side model classes get updated to the component's local value. This involves converting the component's string data to model property type. If the component's values cannot be converted, then JSF advances to Render Response phase and flag messages against the respective components.

- **Invoke Application**

  In this phase, all those application-specific events which have not yet been handled are handled by invoking the Web application. All the form data that has passed through conversions and validations will be now processed as per the business logic.

- **Render Response**

  In the final phase of JSF life cycle, the response is rendered to the client. JSF implementation transfers the task of rendering the page to the JSP container if the application is using a JSP page. If this rendering is in response to an initial request, then the components represented on the page are added to the view tree. After rendering, the state of the responses sent is saved for later restoration during the Restore View phase.

5.5 Developing JSF Applications

A JSF application like other Web application runs inside a Web container also called as Servlet container. The major goal of JSF is to remove the dependency of a markup or scripting language used for defining UI components and other Web-tier concepts.

Some of the functionalities performed by the components in a JSF application are as follows:

- Custom tag library used for embedding and rendering UI components within the View page.
- JavaBean components, also known as Backing beans which define the properties and functionalities of UI components.
- Custom tag libraries for managing event listeners, validators, converters, and other core actions.
- Server-side classes, also known as helper classes for obtaining the functionalities such as database access.
- A configuration file, `faces-config.xml`, for configuring application resources such as JavaBean, page navigation rules, and other custom objects used in the JSF application or applying annotations to the components.

5.5.1 User Interface Component Model

User interface components are simple and reusable elements that are used to design the user interface of JSP applications. These components can be simple such as text field or button as well as complex such as trees or data table. In JSF, user interface controls are represented by `UIComponent` classes on the server-side. The class defines the behavior of the components in the JSF application environment regardless of how it will be presented to the client. It not only includes the `UIComponent` class, but also the other helper components such as Renderer for rendering the component, Validator for registering validators onto a component, Converter for registering data converters on to a component, and so on. All these different components work together to make the make component work in a JSF application. JSF uses the abstraction technique where the developer just needs to know what the component does and how to use it. The internal working of the component is hidden from the developer.

The base class for all the UI components is `UIComponentBase` class defined in `javax.faces.component` package. The `UIComponentBase` class provides the default state and behavior for all UI components. There is a set of HTML component classes defined in `javax.faces.component.html` package. These components are derived from the `UIComponentBase` class. The UI components are associated with UI tags used in designing the Web page of the application.

The JSF framework creates a tree of UI components at server-side while displaying them on the page. This helps the components to remember the values between the requests forwarded to the server. The component tree is also called as View.

Figure 5.6 shows the component tree managed on the server for the UI components displayed on the Web page.

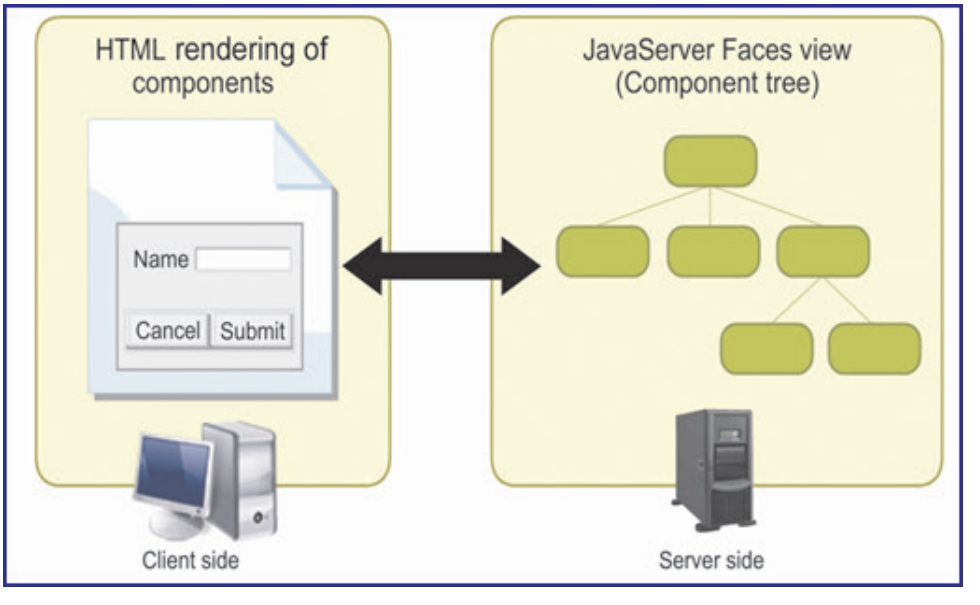


Figure 5.6: Component Tree Generated for Components at the Server-side

As shown in figure 5.6, the component tree generated automatically at the server-side while displaying the Web page to the client. Each component in the tree maps to the component tag provided in the Web page.

5.5.2 Renderers

The components in JSF are not responsible for how they will be displayed to the Web client. They are designed to define the functionality of a component whereas the rendering is performed by separate renderer. The rendering model defines the way the components are displayed to the clients. For example, a `UISelectOne` component can be rendered as a set of radio buttons, as a combo box, or as a list box.

The JSF API includes a standard HTML kit for rendering the component as HTML object to be sent to HTML client.

Table 5.2 describes some of the UI component classes with their corresponding UI tags and the rendered output as HTML in the JSF technology.

Component Class	Component Tag	Functionality	Rendered as HTML	Appearance on the Page
UIColumn	<h:column>	Represents a single column of data and is used with UIData component	A column of data in an HTML table	Column in a table
UICommand	<h:commandButton>/ <h:commandLink>	Represents controls which submits the form data to an application or links to another location when activated	An HTML tag <input type="XXX" /> where XXX is a type value that can be a submit, reset or image	A button or a hyperlink
UIForm	<h:form>	Collection of controls whose data is submitted to the application for processing. This tag resembles the <form> tag in HTML	An HTML <form> element	No appearance
UIGraphics	<h:graphicImage>	Used to display an image	An HTML <img> tag	An image

Component Class	Component Tag	Functionality	Rendered as HTML	Appearance on the Page
UIInput	<h:inputText>/ <h:inputTextArea> </<h:inputSecret> </<h:inputHidden>	Accepts data from user and is a base class for other component classes such as HTMLInputHidden, HTMLInputText, HTMLInputTextArea, and HTMLInputSecret	inputHidden -includes hidden variable in the page inputText - accepts a String value inputTextArea - accepts multiline String inputSecret - accepts a string without echoing it on the screen	inputHidden - no appearance inputText - a text field inputTextArea - a multi-row text field inputSecret - a text field displaying special characters
UIPanel	<h:panelGrid>	Manages the layouts of child components embedded in it	An HTML <table> element with <tr> <td> elements	A table

Table 5.2: UI Component Classes in JSF

5.5.3 Navigation Model

A single Web page is not sufficient to model the application logic. Any commercial Web application requires multiple Web pages that together serve the application functionality. A Web developer has to define a correct sequence of Web pages to be loaded based on user's action. This is a daunting task since the functionality is distributed across multiple pages.

The JSF Navigation model is an easy way to declare a set of rules that define the next view for user based on his actions. These rules are specified using XML elements in the application's configuration resource file, often named as faces-config.xml. The rules uses action event invoked by clicking of button or hyperlink. They can also handle additional processing required to select the correct sequence in which pages are to be loaded.

Figure 5.7 depicts a navigation model.

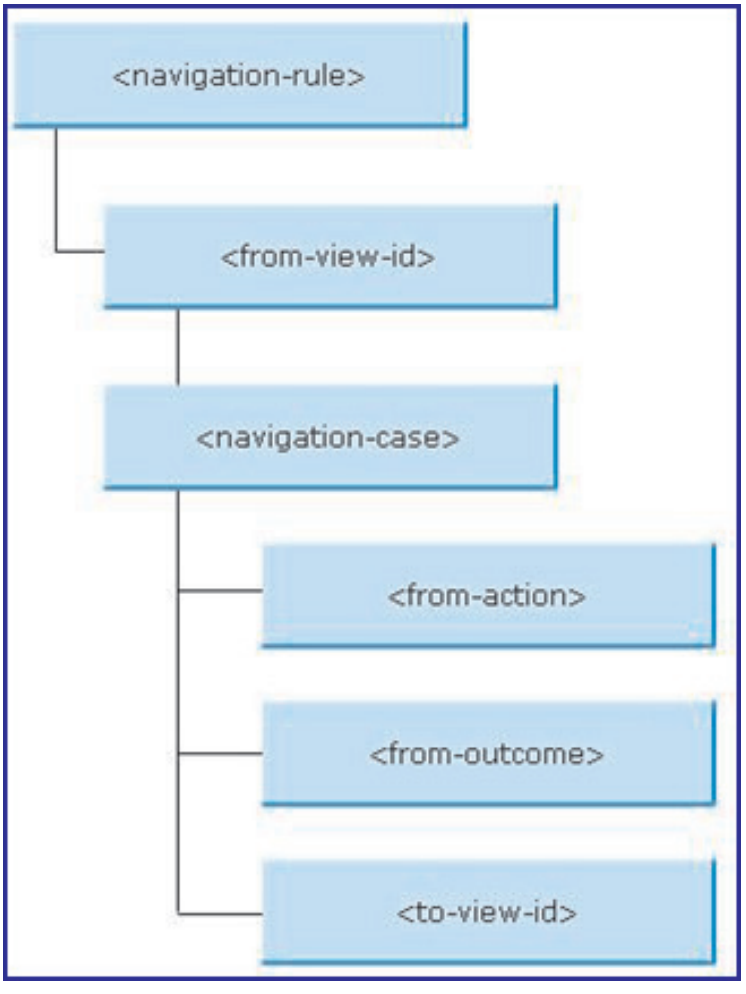


Figure 5.7: Navigation Model

To facilitate navigation, the Web developer performs the following steps:

- **Define set of navigation rules**

  The `<navigation-rule>` defines which pages should be chosen from a set of pages. A `<navigation-case>` represents each path to a page. A navigation rule can optionally specify the page to which it is bound with the `<from-outcome>` tag. Each rule must have the `<to-view-id>` to define which page to load next. The navigation case is selected based on logical outcome and/or the expression for action method.

- **Bind these rules to the UI component**

  Each navigation rule is coupled with one or more pages. The navigation occurs when the `UIComponent` within the page trigger an action event. To bind a navigation rule to `UIComponent`, it must implement faces `ActionSource` interface. The action attribute renders a String that specify mapping between navigation rule and user action.

  Code Snippet 3 shows the code to handle navigation for the current view `'/pages/register.jsp'`.

Code Snippet 3:

```
...
<navigation-rule>
<from-view-id>/pages/register.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/pages/welcome.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/pages/loginError.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

Code Snippet 3 defines two cases for navigation from `'/pages/register.jsp'`. In the first case, if the logical outcome is 'success', then it navigates to `'/pages/welcome.jsp'` page. If the outcome is 'failure', it navigates to `'/pages/loginError.jsp'` page.

JSF 2.0 also supports implicit navigation model. The implicit navigational model is based on auto view page resolver who is responsible for selecting the correct View page. To search for the appropriate View, the developer must put the View name in the `action` attribute.

5.5.4 Handling a Navigation Event

The navigation model internally relies on MVC-based event model. When a user clicks a UI component, a navigation event is generated.

Figure 5.8 depicts handling of a navigation event. It shows the flow of events when navigation occurs.

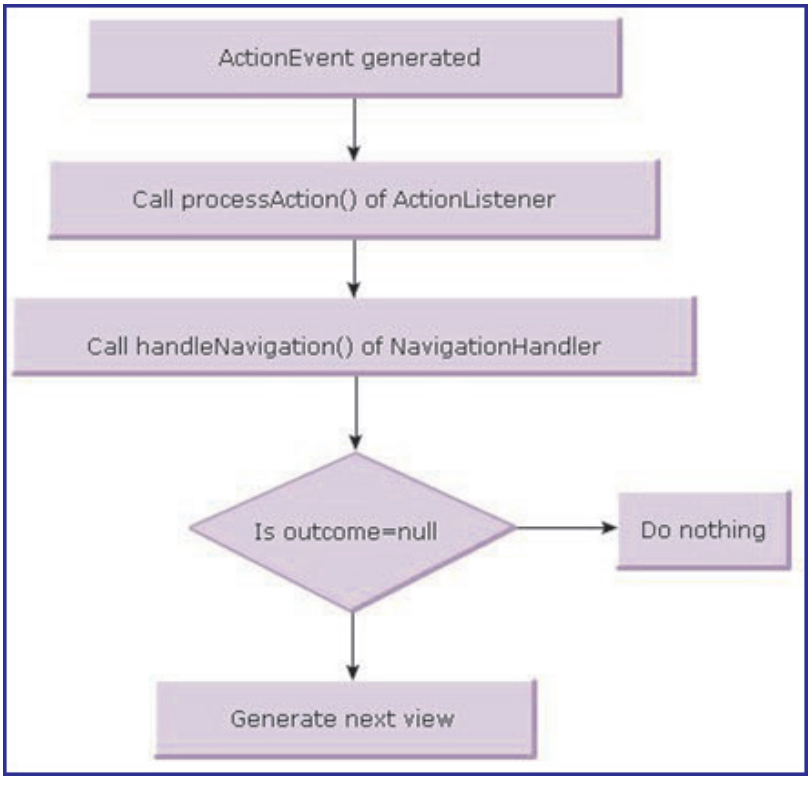


Figure 5.8: Handling a Navigation Event

Once the `ActionEvent` is generated, the request processing life cycle at runtime creates a default instance of `ActionListener` and call its `processAction()` method to determine how to process this event. The `ActionListener` object identifies the source component that triggered this event by accessing source property of `ActionEvent` class. The `action` property of the identified UI component renders either a `String` or a reference to `action` method. This reference is passed to `NavigationHandler`. The `handleNavigation()` method of `NavigationHandler` compares it against the navigation rules in the application configuration resource file. If no match is found, then user remains on the same page. When a match is found, navigation occurs to create new user view as per the logic defined in rule.

5.5.5 Concept of Backing Bean

A typical Web application uses many resources to implement its functionality. These resources can be managed efficiently if the application developer separates the objects defining user interface from the objects that implement the business logic. The scope of these objects needs to be properly defined and managed.

The JSF framework makes this task easier by providing Backing Bean Management facility. A backing bean defines the objects that hold business data and perform application-specific processing on that data.

A JSF based Web application couples one or more backing beans with each page in the application. The properties and methods of UIComponents appearing in the page are defined using the backing beans.

Each backing bean property is set to either a value or an instance of associated UIComponent.

Some of the functions performed by managed bean methods comprise the following:

- Handle an event activated by a component
- Execute the procedure of locating the next page where the application must navigate to
- Validate the data of a component

After development of backing beans, it needs to be registered with application's configuration resource file, so that JSF implementation can create their instances automatically whenever required.

Code Snippet 4 demonstrates the creation of a backing bean.

Code Snippet 4:

```
...
public class Product {
String name;
float price;
public Product() {
}
public void setName(String name) {
this.name = name;
}
public String getName() {
return name;
}
public void setPrice (float price) {
this.price = price;
}
public float getPrice() {
return price;
}
}
...
```

Code Snippet 4 creates a Java bean named 'Product' with 'name' and 'price' as its properties that store the name and price of the product.

Code Snippet 5 registers this Product bean to `faces-config.xml` file.

Code Snippet 5:

```
<managed-bean>
<managed-bean-name>ProductBean</managed-bean-name>
<managed-bean-class>com.aptech.Product</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

The code is to be written in applications configuration resource file. The `<managed-bean-name>` tag sets the name of backing bean to '**`ProductBean`**'. The `<managed-bean-class>` tag specifies fully qualified name of class implementing the business logic. The `<managed-bean-scope>` tag defines the scope of **`ProductBean`** to be within the request object.

5.5.6 Binding Backing Bean

The JSF framework supports Expression Language (EL) that can be used to access the application data stored in backing bean. Each UI component in the Web page is coupled with the property of associated backing bean. The UIComponent can be bound to either the property or methods of backing bean. This is achieved by setting the value attribute of component tag to the EL expression augmented with '#'.

Figure 5.9 depicts use of backing bean.

```
<h:form>
        <h:outputLabel value="Product Name"/>
        <h:inputText value="#{Productbean.name}"/>
        <h:outputLabel value="Product Price"/>
        <h:inputText value="#{Productbean.price}"/>
</h:form>
```

Figure 5.9: Backing Bean

The figure shows the code that uses a backing bean named 'Productbean' for accessing application data. The `<inputText>` tag creates two input text fields where user can enter name and price of the product. The `value` attribute of input text is mapped to properties of Productbean using EL expression. The values entered by user are stored in the properties 'name' and 'price' respectively.

5.5.7 Methods of Backing Beans

The various functions performed by the methods of backing bean are as follows:

- **Handling Navigation**

  A backing bean method that handles navigation processing is known as `action` method. It is user-defined public method that does not accept parameters and returns a `String`, which will be used to determine the next view for user. The `action` attribute of component tag is used to refer to the `action` method of backing bean.

- **Handling Action Event**

  A backing bean method handles any action event associated with UIComponent. It is a user-defined public method that accepts an ActionEvent as parameter and returns void. The `actionListener` attribute is used to refer to this action method.

- **Performing Validation**

  A backing bean method can perform validation of input data. It is a user-defined method that accepts two arguments: a `FacesContext` object representing the component whose data must be validated and data for comparison. The `validator` attribute refers to this method. For validation, the component must extend to `UIInput` class.

- **Handling ValueChange Event**

  A backing bean method can be coded to handle an action event that is triggered whenever the value associated with `UIComponent` is changed. It is a user-defined public method that accepts a `ValueChangeEvent` as parameter and returns `void`. The `valueChangeListener` attribute of component tag refers to this method.

5.5.8 Annotations for Managed Beans

In JSF 2.0, managed beans can be registered through annotations. The `@ManagedBean` annotation marks the bean as a managed bean. It takes the name attribute to specify the bean name. Similarly, the developer can use scope annotations to define the scope in which the bean will be stored. Developer defines one of the following scopes for a bean class.

- **Application (javax.enterprise.context.ApplicationScoped)**: Application scope prevails across all users' interactions with a Web application.
- **Session (javax.enterprise.context.SessionScoped)**: Session scope prevails across multiple HTTP requests in a Web application.
- **Flow (javax.faces.flows.FlowScoped)**: Flow scope continues during a user's interaction with a specific flow of a Web application.
- **Request (javax.enterprise.context.RequestScoped)**: Request scope continues during a single HTTP request in a Web application.
- **Dependent (javax.enterprise.context.Dependent)**: Dependent scope shows that the bean depends on some other bean.

The bean is incorporated whenever it is referenced, if it is defined `@Dependent`. To refer the managed bean by the binding attribute of a component tag, developer should define the bean with a request scope. If the bean is in session or application scope, the bean will take measures to

ensure thread safety, because each `javax.faces.component.UIComponent` instances is depended on running inside of a single thread.

When a developer configures a bean that allows attributes to be associated with the View, developer uses the `@ViewScoped` annotation. The attributes persist until the developer navigates to the next View.

Code Snippet 6 demonstrates the creation of a backing bean.

Code Snippet 6:

```
...
@ManagedBean(name="ProductBean", eager="true)
@RequestScoped
public class Product {
String name;
float price;
public Product() {
}
public void setName(String name) {
this.name=name;
}
public String getName() {
return name;
}
public void setPrice (float price) {
this.price=price;
}
public float getPrice() {
return price;
}
}
...
```

In the given code, the bean is marked with the name **`ProductBean`**. The attribute `eager="true"` specifies that the managed bean is created before it is requested for the first time. The life of the bean is as long as the HTTP request and response exists. This means that the bean will be created when the HTTP request is sent and destroyed when the response is received.

5.6 JSF Tag Libraries

JSF tag library is a collection of related tags called Component tags. The component tags are custom tags that define actions for the associated UI Component. A JSF-based Web application uses these component tags to interact with JSP pages in the application.

The main purpose of using tags is to keep Java code out of the pages and thereby enable front-end developers to keep Web page design simple and elegant.

JSF framework supports two types of libraries named Core Tag library and HTML Tag library. The HTML Tag library consists of all the component tags that are specific to HTML clients.

The Core Tag library consists of all the component tags that are independent of any page markup language. These tags allow you to take advantages of features of JSF framework, such as validation, conversion, and event handling.

5.6.1 Elements of JSF Core Tag library

Core tag library also contains tags for views and sub-views, loading resource bundle, and adding arbitrary text to a page. All the action elements in the core tag library are commonly prefixed with '`f`'.

A JSF-based Web application must import the core tag library before it can use its custom tags by specifying a `taglib` directive at the top of JSP file. The value of `uri` attribute of `taglib` directive denotes the reference path to Sun Website from where the JSF Core library can be accessed. The value of prefix attribute indicates that prefix '`f`' will be added to refer to each component tag under this library. For example, the `actionListener` tag will be denoted as `<f:actionListener>`.

Some of the tags defined in this library are as follows:

- **<f:actionListener>**

  It renders an instance of class defined by the type attribute, which implements the Action-Listener interface associated with parent UIComponent.
- **<f:attribute>**

  It adds a generic attribute for the parent UIComponent.
- **<f:convertDateTime>**

  It renders a converter instance that is used for formatting the appearance of date and/or time in JSP page.
- **<f:convertNumber>**

  It renders an instance of converter which is used to format the appearance of numeric data in JSP page.

Figure 5.10 shows the use of JSF core tag library.

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<html>
<body>
      <h:commandButton id="button1" value="Click Me">
        <f:actionListener type="aptech.CustomerBean" />
      </h:commandButton>
</body>
</html>
```

Figure 5.10: JSF Core Tag Library

5.6.2 Basic Tags

The other library supported by JSF is known as the HTML Tag library. It consists of all the component tags that render HTML controls such as text fields, buttons, and form.

The HTML tag library must be imported before use by specifying `taglib` directive at the top of your JSP file. The value of `uri` attribute refers to the path on Sun Website from where one can access this library. The value of `prefix` attribute denotes that prefix 'h' will be added to define each tag under this library.

Figure 5.11 shows the use of HTML tag library.

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<html>
<body>
      <h:form>
            <h:outputText value="Welcome to JSF" />
      </h:form>
</body>
</html>
```

Figure 5.11: JSF HTML Tag Library

The general syntax of specifying the html tag as a JSF tag is as follows:

Syntax:

```
<htmlprefixvalue:tagName attribute1="value" attributen="value">
</htmlprefixvalue:tagName attribute1="value" attributen="value">
```

where,

- `Htmlprefixvalue` - denotes the value of prefix that will be used while defining the tag such as 'h' or 'html'.

- `tagName` - denotes the name of the tag that renders UIComponent.
- `attributes` - 1 to n denotes the attributes that are defined for this component tag.

The JSF's HTML tags are used for designing interface by which user can interact with Web application. Based upon the user interface components that are rendered, the html tags can be divided into following categories:

- **Inputs**

  The tags under this category render HTML input elements that accepts an input from the user. For example, the `inputText` is used to create input text field.

- **Outputs**

  The tags under this category render HTML output elements that are used to output results to the user. For example, the `outputText` tag creates a component for displaying formatted output.

- **Commands**

  The tags under this category render Submit buttons that perform user actions. These buttons can be associated with beans. For example, the `commandButton` tag creates a Submit button.

- **Selections**

  The tags under this category render HTML selection components such as radio buttons, list boxes, and menu. For example, the `selectOneRadio` tag creates a radio button.

- **Layouts**

  The tags under this category are used to define appearance of data to be displayed on the Web page. For example, the `panelGrid` tag renders a grid component to display data.

- **Data Table**

  The tags under this category render HTML tables. For example, the `dataTable` creates a table on the Web page.

- **Errors and Messages**

  The tags under this category render customized error and informatory messages associated with the UIComponent. For example, the `message` tag displays most recent messages for the component.

5.6.3 Convertor Tag

JSF tag library has a collection of pre-built converters to convert string to int, float, Boolean, and so on. Converter tags converts data into appropriate types before the application processes the data.

With help of the converter tags, data such as age, salary, and year can be represented by Integer, Double, and Date objects.

The different types of converter tags are explained as follows:

- **Convert Number Tag**

  The `<f:convertNumber>` is used to convert numbers to different representations such as currency, percentage, and so on. It contains attributes to format the display information. For example, Currency Code and Currency Symbol are used to control the display of currency information.

  Code Snippet 7 demonstrates the `<f:convertNumber>` tag on the JSP page.

Code Snippet 7:

```
<f:convertNumber pattern="patternFormat" minIntegerDigits=
"minDigits"
maxIntegerDigits="maxDigits" minFractionDigits="minDigits"
maxFractionDigits="maxDigits" groupingUsed="trueOrfalse"
integerOnly="trueOrfalse" type="numberOrCurrencyOrPercent"
currencyCode="currencyCodeValue" currencySymbol=
"currencySymbol"
locale="locale">
</f:convertNumber>
```

Code Snippet 8 shows the `<f:convertNumber>` tag of currency that can be used to convert a value into particular currency.

Code Snippet 8:

```
<f:<h:outputText value="#{example.exchange}">
 <f:convertNumber type="currency"
</h:outputText>
```

- **Convert Date Time Tag**

  The `<f:convertDateTime>` tag converts the String to a Date/Time object. It has a set of attribute values to format both Date and Time values.

  Following is the syntax for `<f:convertDateTime>` tag.

  Syntax:

```
<f:convertDateTime dateStyle=
"default|short|medium|long|full"
      timeStyle="default|short|medium|long|full"
      pattern="pattern" type="time|date|both">
      </f:convertDateTime>
```

- **Convert Tag**

  The `<f:converter>` tag converts the user input. For example, a string value that has

to be converted into an object before being set as a property in the registered managed bean.

Following is the syntax for the <f:converter> tag.

Syntax:

```
<f:converter converterId="ClassNameOfTheConverter"/>
```

5.6.4 Validator Tag

In JSF framework, binding between JSF UI components and validation logic is done with the help of Validator tags. Validator tags helps in validating the data from the clients, before being processed by the server Web application.

The four JSF tags available for doing validation on the JSF UI components are as follows:

- **Validate Length Tag**

  Validation Length Tag is identified by <f:validateLength> tag. Validation Length Tag specifies the maximum and the minimum characters, a JSF UI Component can accept.

  The syntax for <f:validateLength> tag is as follows:

  Syntax:

```
<f:validateLength minimum="minRange" maximum="maxRange">
</f:validateLength>
```

- **Validate Long Range Tag**

  Validate Long Range Tag validates on JSF UI Components whose value is expected to fall between certain integer (long) values. The Validate Long Range Tag is identified by <f:validateLongRange> tag.

  The syntax for <f:validateLongRange> tag is as follows:

  Syntax:

```
<f:validateLongRange minimum="minLongValue"
maximum="maxLongValue">
</f:validateLongRange>
```

  This tag validates if the component value is within a specified range. The value must be of numeric type or string convertible to a long.

- **Validate Double Range Tag**

  Validate Double Range Tag operates on floating data. It executes range validations on UI Components that accepts floating values.

The syntax for the <f:validateDoubleRange> tag is as follows:

Syntax:

```
<f:validateDoubleRange minimum="minDoubleValue" maximum=
"maxDoubleValue">
</f:validateDoubleRange>
```

Validater Tag

Validator Tag performs customized validations on UI Components. For example, validating whether the entered user id is in the appropriate format or whether the stock symbol is available in the database.

The syntax for the <f:validator> tag is as follows:

Syntax:

```
<f:validator validatorId="IdForValidator">
</f:validator>
```

Check Your Progress

1. Match the purpose of the JSF components with the appropriate component name.

Purpose		Component	
(A)	Decides which page to present next.	(1)	Message
(B)	Formats an object to a suitable text value.	(2)	UI Component
(C)	Focuses on interacting with end user.	(3)	Managed Beans
(D)	Displays application and error messages.	(4)	Navigation
(E)	Represents model components of MVC.	(5)	Converter

(A)	(A)-(2), (B)-(1), (C)-(5), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(5), (C)-(4), (D)-(2), (E)-(1)
(B)	(A)-(5), (B)-(3), (C)-(4), (D)-(2), (E)-(1)	(D)	(A)-(4), (B)-(5), (C)-(2), (D)-(1), (E)-(3)

2. Match the description with the appropriate life cycle phases of JSF.

Description		Phase	
(A)	Final phase of JSF life cycle.	(1)	Apply Request Value
(B)	Server-side model classes get updated to the component's local value.	(2)	Render Response
(C)	In this phase, each component's state is retrieved.	(3)	Invoke Application
(D)	Form data passed is processed as per the business logic.	(4)	Restore View Phase
(E)	Starts when the application responds to a request coming from FacesServlet controller.	(5)	Update Model Values

Check Your Progress

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(5), (C)-(4), (D)-(2), (E)-(1)
(B)	(A)-(5), (B)-(3), (C)-(4), (D)-(2), (E)-(1)	(D)	(A)-(4), (B)-(5), (C)-(2), (D)-(1), (E)-(3)

3. Identify the correct option for specifying `taglib` directive to include the JSF Core tag library in your application.

(A)	`<%@ taglib prefix="core" %>`
(B)	`<%@ taglib uri="http://java.sun.com/jsf/core " %>`
(C)	`<%@ taglib uri="http://java.sun.com/jsf/html " prefix="core" %>`
(D)	`<%@ taglib uri="http://java.sun.com/jsf/core " prefix="f" %>`

4. Match the UI tags against their description.

Description		UI Tag	
(A)	This category renders HTML elements that are used to format output.	(1)	`selectOneRadio`
(B)	This category renders HTML elements that are used for selection.	(2)	`inputText`
(C)	This category renders HTML button.	(3)	`outputText`
(D)	This category renders HTML elements that accept data from user.	(4)	`commandButton`
(E)	This category renders HTML elements that display data for user.	(5)	`outputFormat`

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(5), (B)-(1), (C)-(4), (D)-(2), (E)-(3)
(B)	(A)-(5), (B)-(3), (C)-(4), (D)-(2), (E)-(1)	(D)	(A)-(4), (B)-(5), (C)-(2), (D)-(1), (E)-(3)

Check Your Progress

5. Which of the following statements are true about Validators?

Statement A: In JSF framework, binding between JSF UI components and validation logic is done with the help of Validator tags.

Statement B: Validator tags helps in validating the data from the clients, after being processed by the server Web application.

(A)	Statement A is TRUE and Statement B is FALSE	(C)	Both the statements are TRUE
(B)	Statement A is FALSE and Statement B is TRUE	(D)	Both the statements are FALSE

Answer

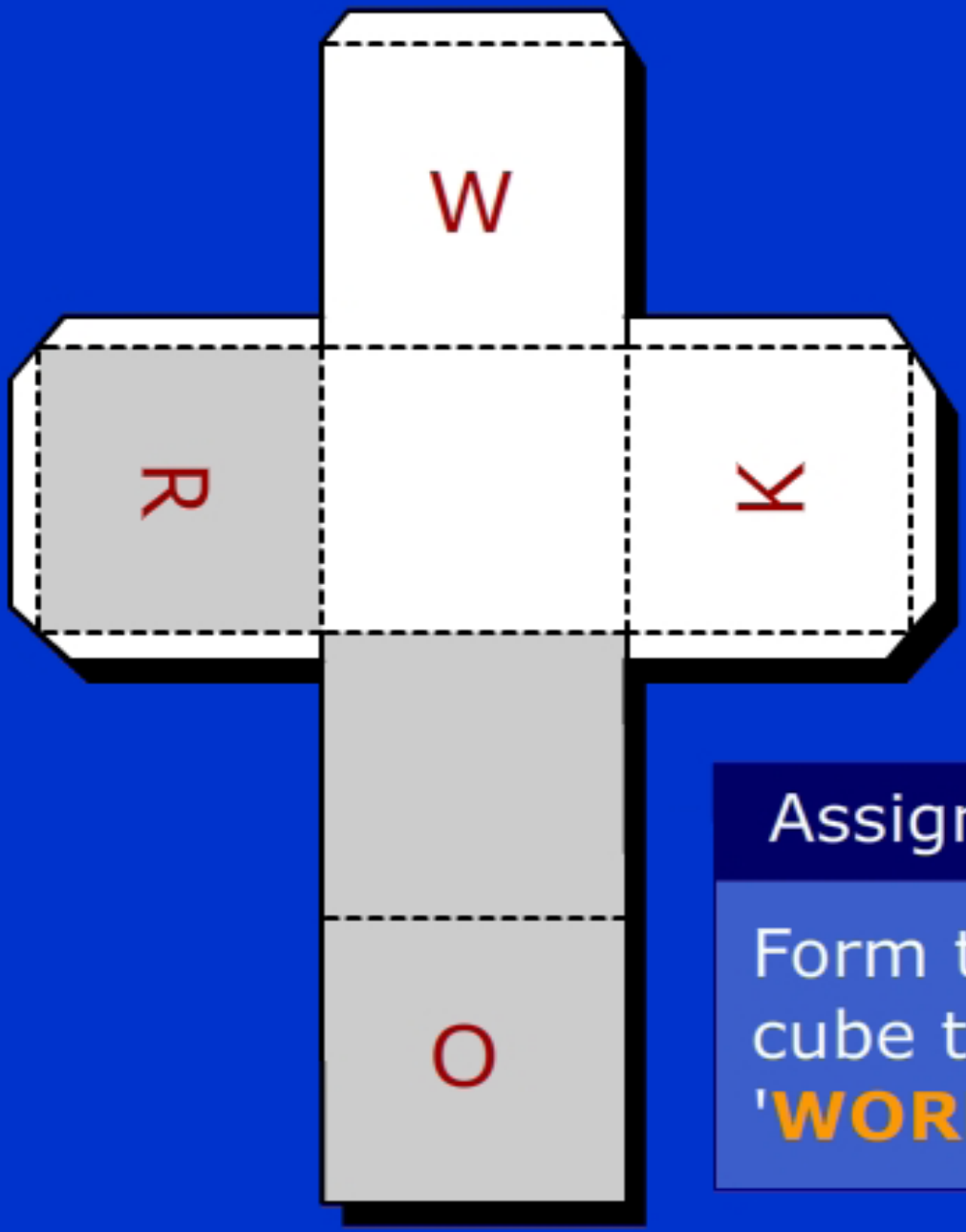
1.	D
2.	A
3.	D
4.	C
5.	A

Summary

- JSF is an open source, component based, event driven framework for building user interface for Web applications.
- JSF components are based on event-driven programming model in which events are handled at the server-side.
- The components of JSF are namely, Managed Beans, UI components, convertor, validator, renderers, events and listeners, and navigation.
- The faces-config.xml file contains the configuration of the JSF application. The Faces-Servlet is the entry point for all requests for a JSF page.
- The JSF life cycle phases manage the input data, so that the Web developer does not need to write the code for processing the request manually.
- User interface components are simple and reusable elements that are used to design the user interface of JSP applications.
- The rendering model defines the way the components are displayed to the clients.
- The JSF Navigation model is an easy way to declare a set of rules that define the next view for user based on his actions.
- JSF tag library is a collection of tags that assist Web developer in performing common tasks such as creating user interface components, performing formatting of displayed data.

WORK ASSIGNMENT

Onlinevarsity
your e-way to learning



Assignment

Form the cube to read '**WORK**'.

"Practice does not make perfect. Only perfect practice makes perfect."
- Vince Lombardi

For perfection, solve assignments @

www.onlinevarsity.com

Onlinevarsity

your e-way to learning

Balanced Learner-Oriented Guide

for enriched learning available @

www.onlinevarsity.com

06

Expression Language, Facelets, and Data Table

Welcome to the Session, **Expression Language, Facelets, and Data Table**.

This session describes the features of JSF 1.0 and JSF 2.0. The session explains about the Expression language used in JSF. It also explain the Facelets Views for declaring templates. Further, the session explains about JSF 2.0 DataTables and JSF 2.0 event handling. Finally, the session demonstrates the implementation of integrating JSF managed bean with JDBC API.

In this Session, you will learn to:

- Describe the features of JSF 1.2 and JSF 2.0
- Explain Expression language and its types in JSF
- Explain the use of facelets View in JSF 2.0
- Explain JSF 2.0 DataTables
- Explain JSF 2.0 Event Handling
- Explain the code to integrate JSF 2.0 with JDBC API

Session
06

Expression Language, Facelets, and Data Table

6.1 Introduction

JSF provides enhanced User Interface (UI) components for development of View tier. It uses a powerful, component based UI development framework that simplifies Java EE Web development.

Similar to Apache Struts, JSF can also be viewed as MVC framework for building Web forms, validating the input, invoking business logics from model, and displaying results.

Thus, we can summarize JSF in two major revisions that are as follows:

- **JSF 1.2**

  JSF 1.2 introduced Unified Expression Language (EL). The new unified EL introduced in JSF 1.2 represents the union of JSP and JSF expression languages. This new EL is more pluggable and flexible. EL allows developers to use simple expressions to dynamically access data from JavaBeans components.

- **JSF 2.0**

  JSF 2.0 introduced Facelets as the official view technology. Facelets allow the developers to create components using XML markup language rather than writing Java code. JSF 2.0 allows developers to use annotation for configuring the JSF components. This removes the need for `faces-config.xml` from the JSF application. Various annotations are provided for configuring navigation, managed bean, and page transition for the application.

  The main features of JSF 2.0 are as follows:

  - Includes bookmarking for URLs.
  - Expands the existing life cycle mechanism.
  - Gives more support for the **AJAX** requests, use annotations instead of faces-config.xml.
  - Allows development of custom component with little or no Java coding.
  - Provides default exception handling mechanisms. With this, all **runtime errors** will be forwarded to an error page.
  - Provides a mechanism to access persistent store.
  - Eliminates the need to author a JSP tag handler when writing JSF components.
  - Easily creates CRUD-based applications.
  - Separates the 'build the tree' and 'render the tree' processes into two separate life cycle phases.
  - Supports bundling and delivering static resources associated with images, stylesheets, scripts, and so on.
  - Provides a mechanism to minimize the 'Lost Update' and 'Duplicate Button Press' problems.

- Allows partial tree traversal during life cycle execution with the help of Ajax.
- Leverages annotations to declare components, managed beans, navigation rules, and so on to the runtime.
- Provides Date Picker, Tree, Tab View, File Upload components to the Standard HTML Render Kit.
- Streamlines the rendering process through caching.
- Improves the interceptor mechanism delivered through the Phase Listener feature. This helps the developer to control the exact requests that are allowed to be processed by each Phase Listener instance.
- Specifies command line interface for authoring JSF applications.
- Allows JSF application resources access through REST.
- Enables components that publish events through RSS/Atom.
- Improves the UI Component specification to increase the interoperability of UI component libraries from different vendors.
- Adds support for REST.
- Gives support for passing values from page to page.

6.2 Introduction to Expression Language

JSF, being a UI framework, needed its own expression language. This expression language when used with JSP tags had its own limitations. Hence, the specification writers and experts have developed a new unified Expression Language in JSF 1.1, which aligned these technologies by adopting the features offered by JSF. The use of this new EL increased the productivity and made the maintenance easier.

JSF UI components are evaluated in different phases when the Web page containing them is rendered for the first time. Once the user enters the value in the UI components and submits the page, these values are converted, validated, and transmitted to server-side data objects. The component events are processed. JSF divides these activities into phases to handle these tasks. Thus, it must be able to evaluate expressions at different phases of the life cycle.

The problem faced with immediate evaluation is that it is read-only. In addition, JSF components need to get data from the server-side objects during the rendering phase and set the data in the server-side objects during postback phase. JSF components need to invoke methods during their various life cycle stages. These methods are invoked to validate data and handle events. For these requirements, a need was felt to develop a powerful EL.

6.2.1 Expression Language in JSF

The new unified EL introduced in JSF 1.2, which represents the union of JSP and JSF expression languages. The same EL is also used in JSF 2.0.

This new EL is more pluggable and flexible. EL allows developers to use simple expressions to dynamically access data from JavaBeans components.

Following are the features of the new unified EL:

- Deferred evaluation of expression
- Supports expressions that can set values and invoke methods
- Supports usage of JSTL iteration tags with deferred expressions
- Provides a pluggable API for resolving expressions

Thus, it can be said that new EL uses simple expressions to perform the following tasks:

- Read application data stored in JavaBeans components and data structures dynamically
- Write data to forms and JavaBeans components dynamically
- Invoke static and public methods arbitrarily
- Perform arithmetic operations dynamically

6.2.2 Immediate and Deferred Evaluation

The new EL supports both immediate and deferred evaluation of expressions. In immediate evaluation, the expression is evaluated immediately and the result is returned, whereas for deferred evaluation, the expression is evaluated later during the page life cycle.

The `${}` and `#{}` syntax are used to represent immediate and deferred evaluation of EL expression. JSF uses deferred evaluation expressions because of its different phases of a page life cycle.

Immediate evaluation expressions are used within a template text or as a value of a tag attribute that will accept runtime expressions. They are always evaluated as read-only value expressions. Deferred evaluation expressions are evaluated at different phases of a page life cycle. In JSF, the controller evaluates the expression at different phases of the page life cycle, depending on how the expression is used in the page.

Code Snippet 1 shows the use of deferred evaluation expression.

Code Snippet 1:

```
<h:inputText id="empname" value="#{employee.name}" />
```

In the code, the `inputText` tag is a text field that accepts user input. The `value` attribute of the `inputText` tag points to the `name` property of the `employee` bean.

When an initial request is made for the page containing this tag, JSF evaluates the expression `#{employee.name}` during the render-response phase of the life cycle. In this phase, it just accesses the value of `name` from the `employee` bean.

JSF evaluates the expression at different phases of the life cycle during which the value is retrieved from the request, validated, and disseminated to the `employee` bean.

6.2.3 Types of EL Expressions

Two types of expressions that the unified EL supports are value expressions and method expressions.

- **Value Expression**

  Value expressions refer to data present in a bean in the form of property or other data structure or as a literal value. Method expressions refer to methods, which are invoked when the expression is processed.

  Deferred expressions can either be value or method expressions. Value expressions can be used to both read and write. A value expression can either return a value or set a value. The name of the attribute or property can be associated with a value expression using the `setValueExpression()` method. This supports binding of attribute and property of values to dynamically generate calculated results. Whenever it is required to dynamically calculate the result of an expression, the `getValue()` method of the `valueExpression()` method is invoked which returns the evaluated results.

  Value expressions can be used to dynamically compute attributes and properties.

  Code Snippet 2 shows the use of value expressions.

Code Snippet 2:

```
...
<h:outputText rendered="#{user.manager}" value="#{employee.
salary}"/>
...
```

In the code, the rendered property is used to evaluate the boolean expression. Thus, the value stored in the boolean property manager of the user JavaBean is checked to determine whether the salary property of the employee should be displayed or not. If the rendered property will have the value of false then it means the user is not a manager and hence, nothing will be displayed.

Value expressions can be categorized into `rvalue` and `lvalue` expressions. The `rvalue` expressions can only read data whereas `lvalue` expressions can both read as well as write data. Expressions evaluated using the `${}` delimiters are `rvalue` expressions and use immediate evaluation syntax. On the other side, expressions evaluated using the `#{}` delimiters act as both `rvalue` and `lvalue` expressions.

Value expressions can also be used to assign a value accepted from the user into an item obtained by evaluating the expression.

Code Snippet 3 shows the use of value expression.

Code Snippet 3:

```
...
<h:inputText value="#{employee.number}"/>
...
```

In the code, the expression is evaluated as an `rvalue` when the page is rendered and the `getNumber` method present in the `employee` JavaBean is invoked. Hence, the result is displayed as the default value in the text field. On the other side, when the user submits the page, the expression will be evaluated as an `lvalue`. Thus, the value provided by the user after conversion and validation will be pushed into the expression by invoking the `setNumber` method present in the `employee` JavaBean.

- **Method Expressions**

Method expressions are same as value expressions. Method expressions invoke an arbitrary public method of an arbitrary JavaBean object by passing a specified set of parameters. These invoked methods returns results to the method expression. The method expression renders these returned results on the page containing the expression. In JSF technology, a component tag represents a component on a page. The component tag uses method expressions to invoke methods that do some processing for the component.

Code Snippet 4 shows the use of method expressions.

Code Snippet 4:

```
<h:form>
<h:inputText
id="name"
value="#{employee.name}"
validator="#{employee.validateName}"/>
<h:commandButton
id="submit"
action="#{employee.submit}" />
</h:form>
```

In the code, the `inputText` tag is a text field and the `validator` attribute of this tag invokes the method named, `validateName` in the `employee` JavaBean. These methods can be invoked at different phases of the life cycle and thus, must be used as deferred evaluation syntax. The code specifies that the `submit` method must return a string that shows which page to navigate to after the `submit` button is clicked.

The `validateName` method is invoked during the validation phase of the life cycle, whereas the `submit` method is invoked during the application phase of the life cycle. Since, these methods are invoked at different life cycle phases, method expressions should always use deferred evaluation syntax.

Method expressions can use the . and the `[]` operators. For example, `#{object.method}` is same as `#{object["method"]}`. The value within the `[]` is converted into a string and is used to find the names of the method that matches it and once found, the method is invoked.

Different ways in which the method expressions can be used in tag attributes are as follows:

- **Single Expression Construct**

  In this, the method expression is written as `<some:tagvalue="#{bean.method}"/>`.

  where,

  `bean` refers to JavaBean and `method` refers to a method in the JavaBean component. The method in the method expression is invoked later.

- **Text Only**

  In this, the method expression is written as `<some:tag value="sometext"/>`.

  Literals are provided in the method expressions to support action attributes in JavaServer Faces technology. In this expression, when the method is invoked, it returns a String literal, which is then converted to the expected return type as specified in the tag's library descriptor.

  For example, the JSF component tag that uses a method expression can be:
  `<h:commandButton action="#{customer.buy}" value="buy"/>`

  In the code, the EL expression invokes the **buy** method of the **customer** JavaBean. You can also pass parameters to the method.

  For example, `<h:commandButton action="#{customer.buy('STOCK')}" value="buy"/>`. Here, the string '`STOCK`' is passed as a parameter to the buy method.

6.3 JSTL Tag

The integration of the EL in the entire Java EE Web tier helps to use JSTL's `forEach` tag with JSF components. In version 1.2, JSTL defines the attribute named, `items` of the `forEach` and `forTokens` tags, so that they accept both runtime and deferred value expressions. The `${}` syntax and the `#{}` syntax of EL is used to specify the runtime and deferred value expressions.

When a runtime expression is specified, it is evaluated immediately, whereas when a deferred value expression is specified for the `items` attribute, the tag handler adds a mapping for the `var` attribute into an EL `VariableMapper` instance during each iteration.

After Java EL has been unified with the Java EE Web tier, JSTL's `forEach` tag is used with JSF components in an intuitive way.

Code Snippet 5 shows the use of `forEach` tag.

Code Snippet 5:

```
...
<table>
 <tr><th>Book Name</th> <th>Book Price</th>
 <th>Quantity</th></tr>
<c:forEach var="book" books="#{shoppingCart.books}">
 <tr>
  <td><h:outputText value="#{book.name}" /></td>
  <td><h:outputText value="#{book.price}" /></td>
  <td><h:inputText value="#{book.quantity}" /></td>
 </tr>
</c:forEach>
<h:commandButton value="update quantities" action="update" />
</table>
...
```

In the code, the tag handler adds a mapping in the EL `VariableMapper` for each iteration of the book collection.

Similar to iteration tags, the `set` tag now accepts deferred value expression.

Code Snippet 6 shows the implementation of the same.

Code Snippet 6:

```
...
<c:set var="d" value="#{handler.everythingDisabled}"/>
...
<h:inputText id="i1" disabled="#{d}"/>
<h:inputText id="i2" disabled="#{d}"/>
...
```

6.4 Facelets Declaration Language for JSF 2.0

JSF 2.0 introduced facelets as a replacement for JSP as a View declaration language. Facelet is a powerful lightweight page declaration language that is used to build JSF views using HTML style templates. In other words, Facelets are used to create XHTML-based views for the application. The features of Facelets are as follows:

- Provides a server-side template facility that allows the developer to compose a single view page out of several separate files.
- Supports XHTML syntax for creating Web pages.
- Provides an extended tag library.
- Enforces clear separation of MVC by restricting the use of Java code in markup pages.

- Code reusability that is achieved using templates and composite components.
- Provides high performance rendering.
- Reduces the time and effort required for development and deployment.

Code Snippet 7 shows how to build a single logical view for the user.

Code Snippet 7:

```
<!-- This is the main page for the application -->
<f:view>
<include name="menubar" file="menubar.xml"
user="#{currentUser}"/>
<include name="sidebar" file="sidebar.xml"
user="#{currentUser}"/>
<include name="summary" file="summary.xml"
user="#{currentUser}"/>
</f:view>
```

The code snippet create a single page by combining three parts: **menubar**, **sidebar**, and **summary**. This can be reused in the multiple pages and can be further customized by using the EL.

The URI, `http://java.sun.com/jsf/facelets` is the JSF tag library that contains templating tags for the Facelets. To use these tags, the developer must prefix `ui:`. For example, `ui:insert` or `ui:component`. The `facelets` library also support tags for composite components.

Facelets support EL expressions to reference properties and methods of managed beans, bind component objects, or to assign values to methods or properties of managed beans.

6.4.1 Facelet Tags

Facelet tags are used to create a page that acts as the base or template for the other pages in the JSF application. Table 6.1 shows the Facelets tags that are used to perform templating in JSF.

Tag	Syntax	Description
`ui:composition`	`<ui:composition template="optionalTemplate">`	The tag is used in files that are acting as a template client. The tag is used to enable templating in Facelets. The tag adds the component as the direct child of the `UIViewRoot`.

Tag	Syntax	Description
ui:decorate	<ui:decorate template="requiredTemplate">	The tag provides features similar to ui:composition. However, it also includes the content surrounding the <ui:decorate> tag. The template attribute is required for this tag.
ui:define	<ui:define name="requiredName">	The tag defines the content that is inserted into a page by a template. It is used inside the ui:composition tag. It defines the region that will be inserted at the location specified with the ui:insert tag.
ui:insert	<ui:insert name="optionalName">	The tag inserts content into a template. It is used in the template file and the name attribute specifies the location where the template client needs to be inserted.
ui:include	<ui:include src="requiredFilename">	The tag ui:include is used to combine and reuse content for multiple pages.
Ui:remove	<ui:remove>	The tag is mainly used during development to 'comment out' a portion of the markup.
Ui:debug	<ui:debug hotkey="optionalHotKey" />	The tag enables a hot key that popups a new window displaying the component tree.

Table 6.1: Facelet Tags

6.4.2 Templating with Facelets

The facelet template consists of two main files such as template file and template client file.

- **Template file** – This file corresponds to a viewId, such as `greeting.xhtml`.
- **Template client file** – A template client uses one or more developed template to achieve reuse of the page content.

Figure 6.1 illustrates Facelet view with template and template client.

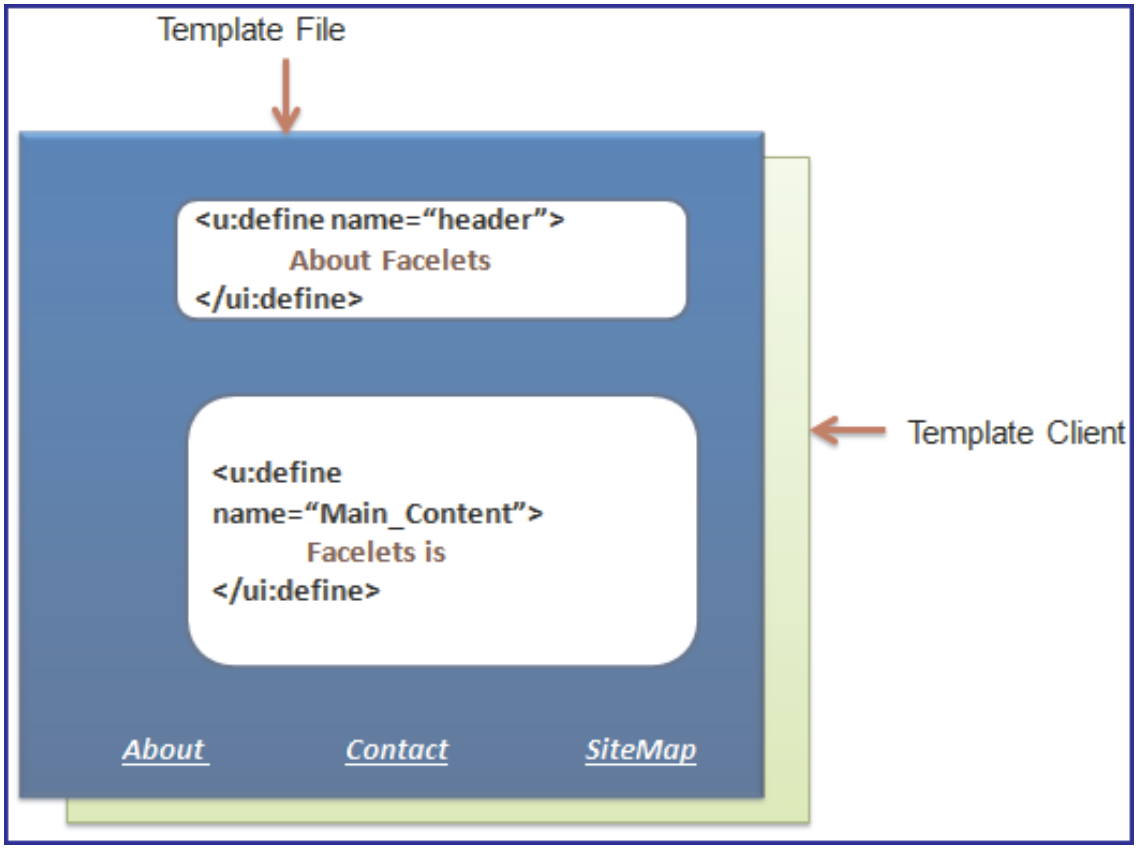


Figure 6.1: Templating with Facelets

Code Snippet 8 shows a template that defines the structure for a page.

Code Snippet 8:

```
<!—template.xhtml -->
. . .
<html xmlns=”http://www.w3.org/1999/xhtml”
  xmlns:ui=”http://java.sun.com/jsf/facelets”
  xmlns:h=”http://java.sun.com/jsf/html”>

 <h:head>
  <meta http-equiv=”Content-Type” content=”text/html;
charset=UTF-8” />
  <h:outputStylesheet library=”css” name=”default.css”/>
  <h:outputStylesheet library=”css” name=”cssLayout.css”/>
  <title>Facelets Template</title>
 </h:head>

 <h:body>
  <div id=”top” class=”top”>
   <ui:insert name=”top”>Top Section</ui:insert>
  </div>
  <div>

  <div id=”left”>
   <ui:insert name=”left”>Left Section</ui:insert>
  </div>

  <div id=”content” class=”left_content”>
   <ui:insert name=”content”>Main Content</ui:insert>
  </div>
  </div>
 </h:body>
</html>
```

The code defines an XHTML page that is divided into three sections: a top, a left, and a main section. The `ui:insert` tag is used to define a default structure for a page.

Code Snippet 9 shows the client page that invokes the template.

Code Snippet 9:

```
. . .
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html">

 <h:body>
  <ui:composition template="./template.xhtml">
   <ui:define name="top">
    Welcome to Template Client Page
   </ui:define>

   <ui:define name="left">
    <h:outputLabel value="You are in the Left Section"/>
   </ui:define>

   <ui:define name="content">
    <h:graphicImage value="#{resource['images:wave.med.
gif']}"/>
    <h:outputText value="You are in the Main Content Section"/>
   </ui:define>
  </ui:composition>
 </h:body>
</html>
```

The client invokes the template by using the `ui:composition` tag. The client page invokes the template page and inserts the content using the `ui:define` tag.

6.5 Introduction to JSF 2.0 DataTable

JSF 2.0 DataTable helps to render and format html tables. DataTable iterates over an array of values to display data. It provides attributes to modify the data.

Code Snippet 10 shows the tag library to be used for DataTable.

Code Snippet 10:

```
<html
xmlns="http://www.xyz.org/2014/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
</html>
```

The most important DataTable operations in JSF 2.0 are Display DataTable, Add data, Edit data, Delete data, and Using Data Model.

- Display DataTable displays a datatable.
- Add data adds a new row in a datatable.
- Edit data edits a row in a datatable.
- Delete data deletes a row in datatable.
- Using Data Model displays row numbers in a datatable.

JSF 2.0 support the use of `h:dataTable` by iterating a list of data filled inside a bean's list to create an HTML table. The main attribute at the `h:dataTable` is the `value` attribute that represents the data over which `h:dataTable` iterates, this data must be one of the following types:

- A Java Object
- An Array
- An instance of `java.util.List`
- An instance of `java.sql.ResultSet`
- An instance of `javax.Servlet.jsp.jstl.sql.Result`
- An instance of `javax.faces.model.DataModel`

The `h:dataTable` component will have only `h:column` and it discards all other component, though the `h:column` can render an unlimited number of the components. The `h:dataTable` component adds a header and footer for the `dataTable` that is created and thus, provides the developer capability.

Code Snippet 11 shows the use of `h:dataTable` tag in `displayproduct.xhtml` to loop over the array of '`order`' object.

Code Snippet 11:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
xmlns:ui="http://java.sun.com/jsf/facelets"
>
<h:head>
<h:outputStylesheet library="css" name="table-style.css" />
</h:head>
<h:body>
<h1>JSF 2 dataTable example</h1>

<h:dataTable value="#{orderbean.orderList}" var="o"
styleClass="order-table"
headerClass="order-table-header">

<h:column>
<!-- column header -->
<f:facet name="header">Order No</f:facet>
<!-- row record -->
#{o.orderNo}
</h:column>

<h:column>
<f:facet name="header">Product Name</f:facet>
#{o.productName}
</h:column>

<h:column>
<f:facet name="header">Price</f:facet>
#{o.price}
</h:column>
```

```
<h:column>
<f:facet name="header">Quantity</f:facet>
#{o.qty}
</h:column>

</h:dataTable>

</h:body>
</html>
```

Code Snippet 12 shows the `table-style.css` page.

Code Snippet 12:

```
.order-table{
 border-collapse:collapse;
 border:1px solid #000000;
}

.order-table-header{
 text-align:center;
 background:none repeat scroll 0 0 #E45EA5;
 border-bottom:1px solid #000000;
 padding:3px;
}
```

The `table-style.css` page contains the properties to change the appearance of the data table displayed on the page.

Code Snippet 13 shows the Order bean that will create a list of orders to be displayed in the data table component.

Code Snippet 13:

```
@ManagedBean(name="orderbean", eager=true)
@SessionScoped
public class OrderBean implements Serializable{

private static final ArrayList<Order> orderList
  =new ArrayList<Order>(Arrays.asList(
  new Employee("10","Beverages",30,5),
  new Employee("20","Stationary",35,3),
  new Employee("30","Appliances",25,25)
 ));
```

```
. . .
}
```

The code creates a managed bean named `OrderBean` and initializes an `ArrayList` object with the instances of `Order` class. The data from the `Order` are displayed in the data table.

6.6 Introduction to JSF 2.0 Event Handling

JSF event handling is based on the JavaBeans event model, where event classes and event listener interfaces are used by the JSF application to handle events generated by components.

Some examples of events in an application include clicking a button, selecting an item from a menu or list, and changing a value in an input field. When a user activity occurs, the component creates an event object that stores information about the event and identifies the component that generated the event. The event is added to an event queue. JSF tells the component to broadcast the event to the corresponding registered listener, which invokes the listener method that processes the event. The listener method may trigger a change in the user interface; invoke backend application code, or both.

- When a user clicks a JSF button or link or changes any value in text field, JSF UI component invokes an event and that event will be handled by the application code.
- Event handler is registered in the application code or managed bean to handle the event.
- JSF 2.0 creates an instance of the corresponding event class and adds it to an event list, when a UI component checks that a user event has happened.
- JSF Component invokes the event, that is, it checks the list of listeners for that event and call the event notification method on each listener or handler.
- JSF also provide system-level event handlers, which can be used to do some tasks when application starts or stops.

Important event handlers in JSF 2.0 are as follows:

- **ValueChangeListener** - Value change event is invoked when user make changes in input components.
- **ActionListener** - Action event is invoked when user clicks a button or link component.
- **Application Events** – Application is invoked during JSF life cycle: `PostConstructApplicationEvent`, `PreDestroyApplicationEvent`, and `PreRenderViewEvent`.

6.7 Integrating JDBC with JSF 2.0

Code Snippet 14 shows the integration of JSF 2.0 managed bean with JDBC API.

Code Snippet 14:

```
@ManagedBean(name="customerData")
@SessionScoped
public class CustomerData implements Serializable{

 public List<Author> getCustomers(){
  ResultSet rs=null;
  PreparedStatement pst=null;
  Connection con=getConnection();
  String stm="Select * from customers";
  List<Customer> records=new ArrayList<Customer>();
  try{
   pst=con.prepareStatement(stm);
   pst.execute();
   rs=pst.getResultSet();

   while(rs.next()){
    Customer customer=new Customer();
    author.setId(rs.getInt(1));

author.setName(rs.getString(2));
    records.add(customer);
   }
  }catch(SQLException e){
   e.printStackTrace();
  }
  return records;
 }
```

```
  public Connection getConnection(){
  Connection con = null;
  String url = "jdbc:sqlserver://localhost/testdb";
  String user = "user1";
  String password = "user1";
  try {
   con = DriverManager.getConnection(url, user, password);
   System.out.println("Connection completed.");
  } catch (SQLException ex) {
   System.out.println(ex.getMessage());
  }
  finally{
  }
  return con;
 }
}
```

Code Snippet 15 shows the `cust.xhtml` page accessing customer data in a `DataTable` element.

Code Snippet 15:

```
<html
. . .
 <h2>JDBC Integration</h2>
 <h:dataTable value="#{customerData.customers}" var="c"
  styleClass="CustomerTable"
  headerClass="CustomerTableHeader" >
  <h:column><f:facet name="header">Customer ID</f:facet>
   #{c.id}
  </h:column>
  <h:column><f:facet name="header">Name</f:facet>
   #{c.name}
  </h:column>
 </h:dataTable>
 </h:body>
</html>
```

Check Your Progress

1. JSF 2.0 introduced _____ as official View technology.

(A)	Facelets	(C)	UI
(B)	Ajax	(D)	Velocity

2. Which of following helps to access the JavaBeans component in the JSF Web application?

(A)	Event Handling	(C)	Annotations
(B)	Expression Language	(D)	Managed Bean

3. Which of following statements are true for Managed Bean Scope?

(A)	Application scope prevails across all users' interactions with a Web application.
(B)	Request scope continues during multiple HTTP requests in a Web application.
(C)	Session scope prevails across single HTTP request in a Web application.
(D)	Flow scope continues during a user's interaction with a specific flow of a Web application.

(A)	A, B	(C)	A, D
(B)	A, C	(D)	B, C

Check Your Progress

4. Which of the following statements are true for Expression Language?

(A)	Helps to do arithmetic, logical, and relational operations.
(B)	Helps in automatic type conversion.
(C)	Helps to create an instance of the corresponding event class.
(D)	Helps to retrieve all the data records from database.

(A)	a, b	(C)	a, d
(B)	a, c	(D)	b, c

5. Which of the following event is invoked when user make changes in input components?

(A)	`ValueChangeListener`	(C)	`ActionListener`
(B)	`PreRenderViewEvent`	(D)	`Application Events`

Answer

1.	A
2.	B
3.	C
4.	A
5.	A

Summary

- JSF 2.0 Expression Language helps to access the JavaBeans component in the JSF Web application.
- JSF helps to construct a UI from a set of reusable UI components and simplifies migration of application data to and from the UI.
- JSF permits custom UI components to be built and reused easily and gives an easy model for wiring client-generated events to server-side application code.
- The JSF 2.0 DataTable renders and formats html tables and iterates over an array of values to display data.
- JSF 2.0 Event Handling manages the events generated by components and is based on the JavaBeans event model.
- JSF 2.0 annotations are used in managed beans, registering listeners, resource rendering, and so on.

Onlinevarsity

your e-way to learning

GROWTH

RESEARCH

OBSERVATION

UPDATES

PARTICIPATION

www.onlinevarsity.com