

Web Component Development Using Java

Are you registered with
Onlinevarsity.com?

Yes



No



Did you download this book
from **Onlinevarsity.com?**

Yes



No



Scores

For each **YES** you score **50**

For each **NO** you score **0**

If you score less than 100 this book is illegal.

Register on **www.onlinevarsity.com**

Web Component Development Using Java Learner's Guide

© 2014 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

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

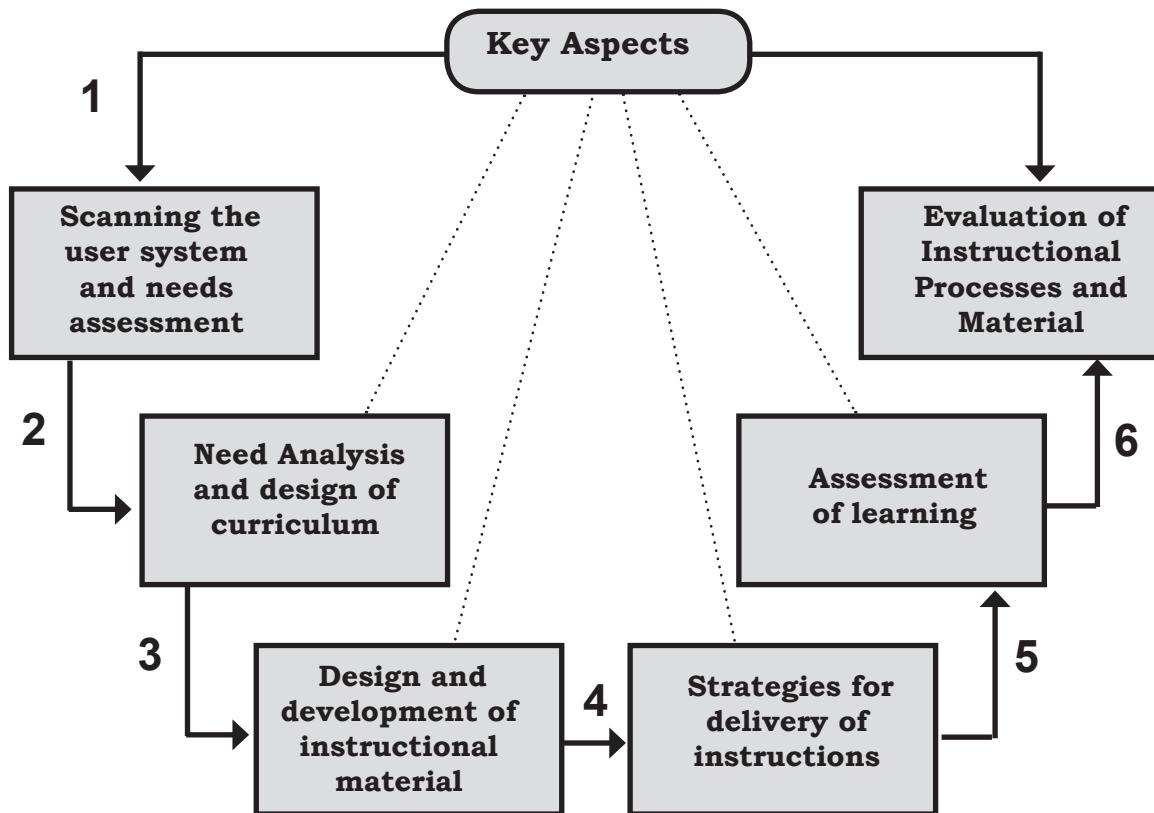
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended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



WRITE-UPS BY

EXPERTS AND LEARNERS

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE



FOR FURTHER READING, LOGIN TO

Preface

With the advent of Internet and its technologies, people are able to communicate and transfer knowledge at the click of the mouse. There are several platforms evolving which are powering Web with dynamic capabilities. One such technology, which has helped Web to become what it is today, is Java technology. Last two decades have seen tremendous changes as far as computing is concerned. Programmers are being pushed to create online applications. Under such scenario, Java platform offers Servlets and JSP as two main components for development application to be hosted on the Web.

Servlet, being a small pluggable extension to a server, enhances the capabilities of the server's functionalities, such as Java enabled Web server, mail server, an application server or any customized server. Servlet APIs available in Java support a higher degree of portability, power, security, flexibility and endurance. Servlets are elegant, object oriented and simple. They can be organized more effectively into manageable parts. They are persistent and fast, as they are loaded, once but can be used many times.

JavaServer Pages, or JSP in short, is a Java-based technology that simplifies the process of developing Web-based applications. It makes the task of maintaining information rich and dynamic content really very easy. Web-based application, which are platform independent, are developed through JSP. This makes JSP all the more desired technology for Web application development. Any Web development process or a large application consists of a combination of application and presentation logic. Using JSP, presentation logic can be separated from application logic quite easily.

The book begins with an introduction to Web applications and its architectures. Then, it explains the Servlet and JSP API that help the programmers to do server-side processing of the request and response. It explains the use of HTTP protocol and its methods used in client-server model of Web applications. The book also cover the new features of Servlet specifications provided on Java Enterprise Edition (Java EE) 7 platform. These includes: Asynchronous Servlet, file uploading, use of annotations in developing Servlets, and so on. It explains the concept and architecture of JSP technology. Further, explains the use of expression language, Java Standard Tag Library (JSTL), use of JavaBeans, and so forth in JSP.

The book also provides an overview of Model-View-Controller (MVC) architecture used in developing Web applications on Java platform.

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

BIG
B

I
G

Balanced Learner-Oriented Guide

for enriched learning available



Table of Contents

Sessions

1. Introducion to Web Applications
2. Java Servlet
3. Session Tracking
4. Filters and Annotations
5. Database Access and Event Handling
6. Asynchronous Servlet Communication
7. JavaServer Pages
8. JSP Implicit Objects
9. Standard Actions and JavaBeans
10. Model-View-Controller Architecture
11. JSP Expression Language
12. JavaServer Pages Standard Tag Library
13. JSP Custom Tags
14. Internationalization
15. Securing Web Applications

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION





Welcome to the Session, **Introduction to Web Applications**.

This session explains the basics concept of Web applications. It explains the architecture of Web application and technologies involved in the Web development. It also explains the request-response mechanism offered by HTTP protocol in accessing the Web resources from the Web servers. Further, the session explains the role of Java in Web application developments and its components such as Servlet and JSP used for extending the functionality of the Web server. The session explains the Web application life cycle, services of Web container, and packaging of Web application in the Web archive file. Finally, the session demonstrates the building and testing of Java Web application in NetBeans IDE.

In this Session, you will learn to:

- Explain the purpose of different applications
- Explain Web applications and their advantages
- Describe the architecture and components of Web application
- Describe the role of HTTP protocol and its methods used for accessing Web pages
- Explain the use of Common Gateway Interface (CGI) language
- Explain the different types of components used in developing Web application
- Explain the advantages and disadvantages of Servlets
- Explain the services provided by a Web container
- Describe the life cycle and directory structure of a Web application
- Explain how to package Web application
- Develop a Web application in NetBeans Integrated Development Environment (IDE)

1.1 Introduction

An application is a collection of programs designed to perform a particular task. Different types of applications are designed for different purposes. Based on the environment in which an application is run and their accessibility by the users, they are classified as follows:

- Desktop Application

A desktop application runs only in a local machine and can neither be viewed nor be controlled by other users. For example, Microsoft Word.

Figure 1.1 depicts the desktop application.

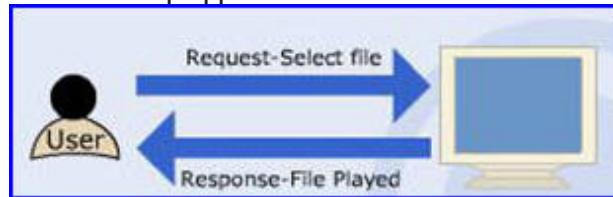


Figure 1.1: Desktop Application

- Networking Application

A network application runs in a Local Area Network (LAN), Metropolitan Area Network (MAN), or World Area Network (WAN) and can be viewed and controlled by users in that particular network only. For example, Novell Netware.

Figure 1.2 depicts the networking application.

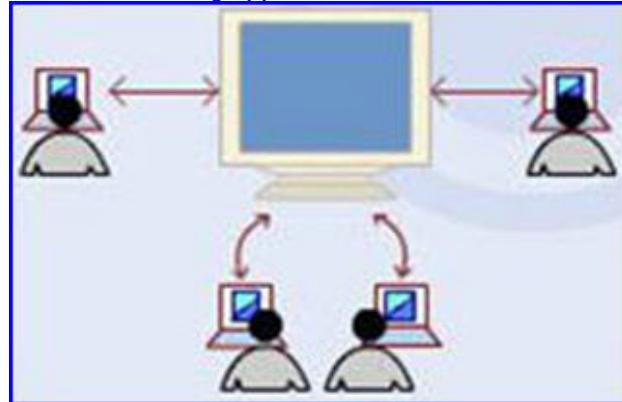


Figure 1.2: Networking Application

- Web Application

A Web application runs at a remote location. It can be viewed and controlled by all the users having administrative or equivalent privilege, throughout the globe, at any instance of time. For example, Internet mail services, such as www.yahoo.com or www.rediff.com.

Figure 1.3 depicts the Web application.

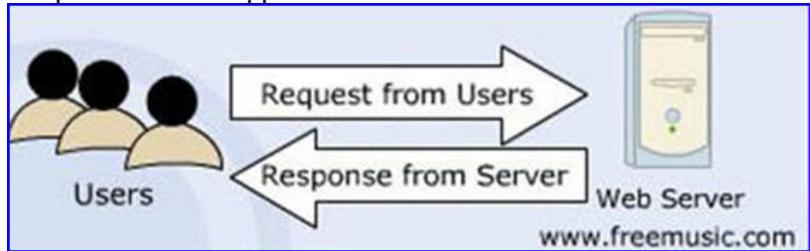


Figure 1.3: Web Application

1.1.1 Web Applications

A Web application is a software application that runs on a Web server. A Web site that a user visits is actually an application running in a Web server at a remote location and executed in a Web browser.

A Web application consists of Web pages which can be static and dynamic. The static Web pages are created using Web technologies such as HyperText Markup Language (HTML), Cascading Style Sheet (CSS), and JavaScript. However, to achieve dynamic functionalities on Web pages, server-side programs are used. The dynamic Web pages provide interactivity with the users. For example, an online shopping Web application allows user to select and deselect the items on the Web page.

Web applications have the following advantages over desktop applications:

- **Easier access to information**

The protocol used for accessing a Web application is Hypertext Transfer Protocol (HTTP) and is supported by most of the Operating System (OS). Moreover, the client software required is just a Web browser, which now comes packaged with most of the OS.

- **Lower maintenance and deployment costs**

Unlike desktop applications, Web applications run in a Web browser and do not need client software to be installed on every client system. Web application code can be modified and maintained at the server end. This saves time and cost involved in upgrading and deploying the applications.

- **Platform independence**

Web applications run in Web browsers and therefore, can be accessed on any machine with different OS.

- **Wider visibility**

The Web application can easily be accessed around the globe at any instance of time.

1.1.2 Web Application Architecture

A Web application basically has three components: the presentation layer, the application layer, also known as business layer, and the data access layer. The architecture of an application defines how these three components will be clubbed together and their interaction with each other. Based on this structure, following are the four application architectures:

- One-tier architecture**

In one-tier architecture, the code related to presentation, business, and data access logic are all clubbed together. Figure 1.4 shows the one-tier architecture.

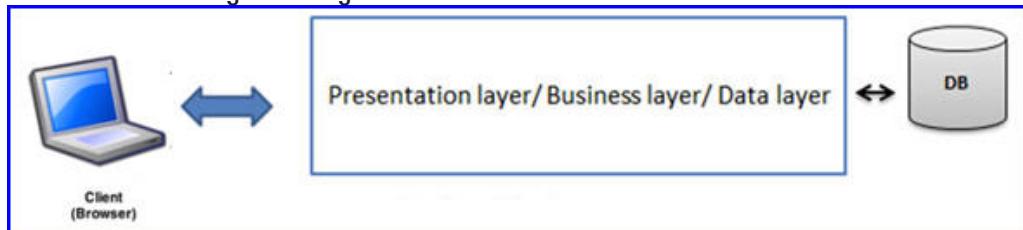


Figure 1.4: Web Application One-tier Architecture

- Two-tier architecture**

In two-tier architecture, the code related to data access logic is separated from the other two components. Any interaction with data tier will be done through business tier. Figure 1.5 shows the two-tier architecture.

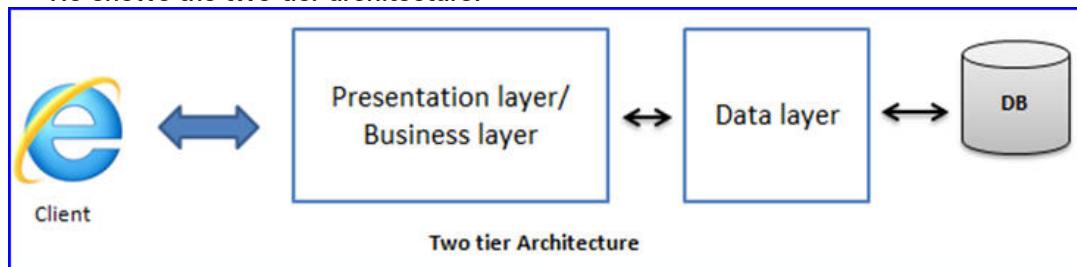


Figure 1.5: Web Application Two-tier Architecture

- Three-tier architecture**

In three-tier architecture, code related to all three components is separated from each other. However, the business tier acts as an interface between the data tier and the presentation tier. Note that the presentation tier cannot directly communicate with the data tier. Figure 1.6 shows the three-tier architecture.

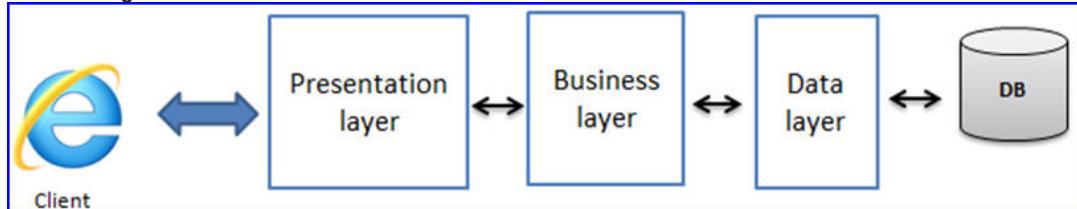


Figure 1.6: Web Application Three-tier Architecture

□ **N-tier architecture**

In N-tier architecture, the layers are further subdivided for ease of functioning. In this architecture, the presentation layer is a graphical user interface that displays data to users. Business layer and application layer are separated reducing the number of locations implementing the logic. Figure 1.7 shows the N-tier architecture.

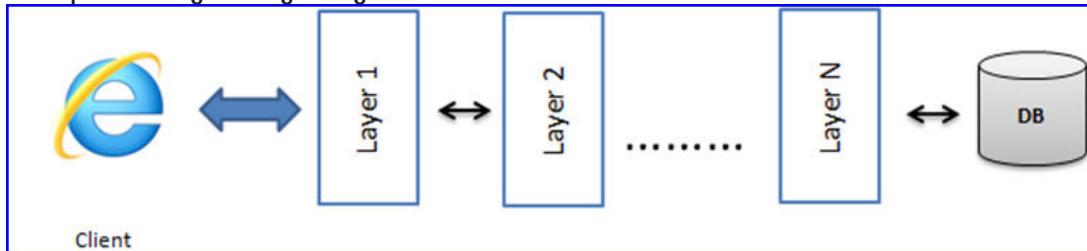


Figure 1.7: Web Application N-tier Architecture

1.2 Web Application Technologies

The most common technologies for communication on the Web are HTML and HTTP.

1.2.1 HTML

HTML is a presentation language which enables Web designer to create visually attractive Web pages. These pages are stored on Web. The users accessing the Web pages on the World Wide Web (WWW) are referred to as Web clients.

HTML allows Web designers to add images, create forms, and embed media objects on the Web pages. All these are referred as Web resources which are stored on the Web server along with HTML pages.

Figure 1.8 shows the Web page designed using HTML.

```

<html>
  <body>

    <h1> Sample HTML Page </h1>

    <!-- Displaying an image -->
    An image:
    </p>

    <p>
      This web page is created in <b> HTML </b> language which has following
      features: </P>

    <!-- Displaying a list -->
    <ol>
      <li> Stands for Hyper Text Markup Language </li>
      <li> Is a markup language </li>
      <li> Contain HTML tags and plain text </li>
      <li> Also called as web pages </li>
    </ol>

  </body>
</html>
  
```

Figure 1.8: Web Page Designed Using HTML

The Web pages are transferred between the Web client and Web server through HTTP protocol.

1.2.2 HTTP Protocol

The requests and responses sent to a Web application from one computer to another computer are sent using HTTP.

An HTTP client, such as a Web browser, opens a connection and sends a request message to an HTTP server asking for a resource. The server, in turn, returns a response message with the requested resource. Once the resource is delivered, the server closes the connection.

HTTP does not store any connection information about the page and hence, it is referred to as a stateless protocol.

Figure 1.9 depicts HTTP request and response to a Web application.

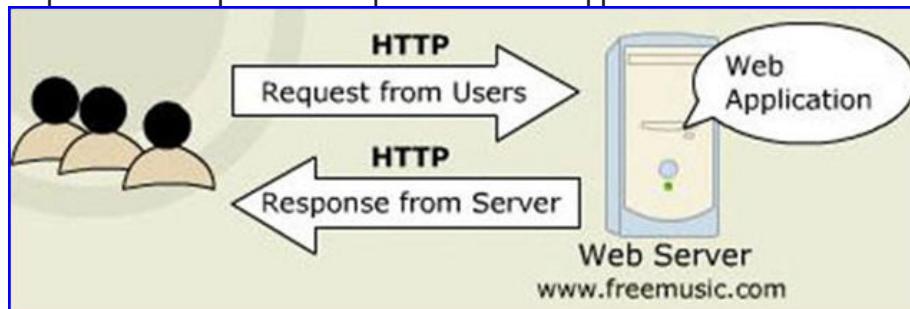


Figure 1.9: HTTP Request and Response Cycle

1.2.3 HTTP Request

The request message sent by the client is a stream of text. Figure 1.10 depicts the request message structure.

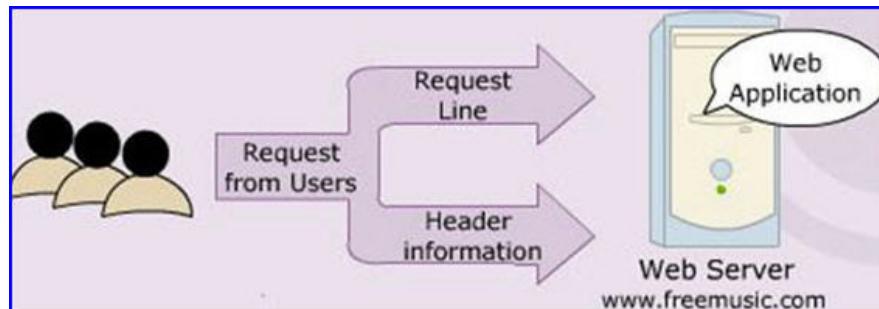


Figure 1.10: Request Message Structure

It consists of the following elements:

- Request line**

This element contains the HTTP method, the address of the Web page or document referred to as Uniform Resource Locator (URL), and the HTTP version it is using to send the request message.

Figure 1.11 shows the request line of the request message.

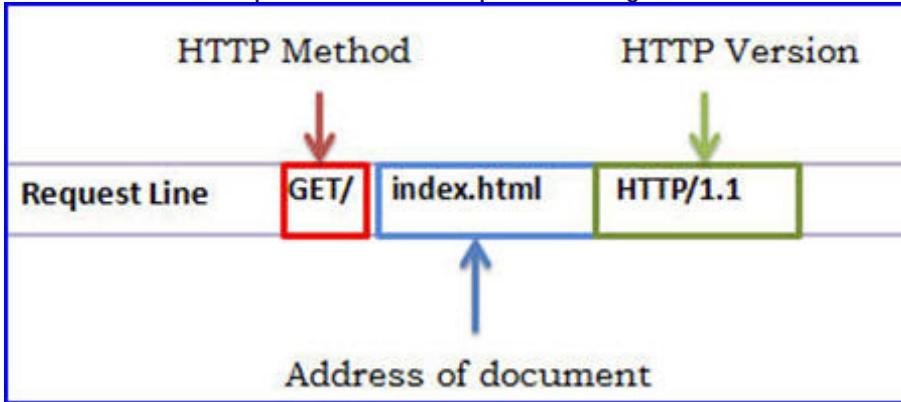


Figure 1.11: Request Line Structure

□ Header information

This element specifies the User-Agent along with the Accept header. The User-Agent element header indicates the browser used by the client. The Accept header element provides information on what media types the client can accept. After the header, the client sends a blank line indicating the end of request message.

Figure 1.12 shows the sample header information sent in the request message.

```
User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: image/gif, image/jpeg, text/*, */*
```

Figure 1.12: Header Information

1.2.4 HTTP Response

The Web application processes the request sent by a client and generates a response message for the requesting client.

Figure 1.13 depicts the response message structure.

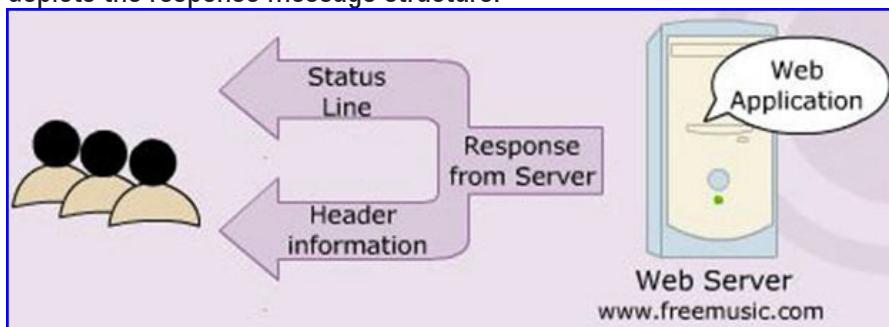


Figure 1.13: Response Message Structure

The response message consists of the following elements:

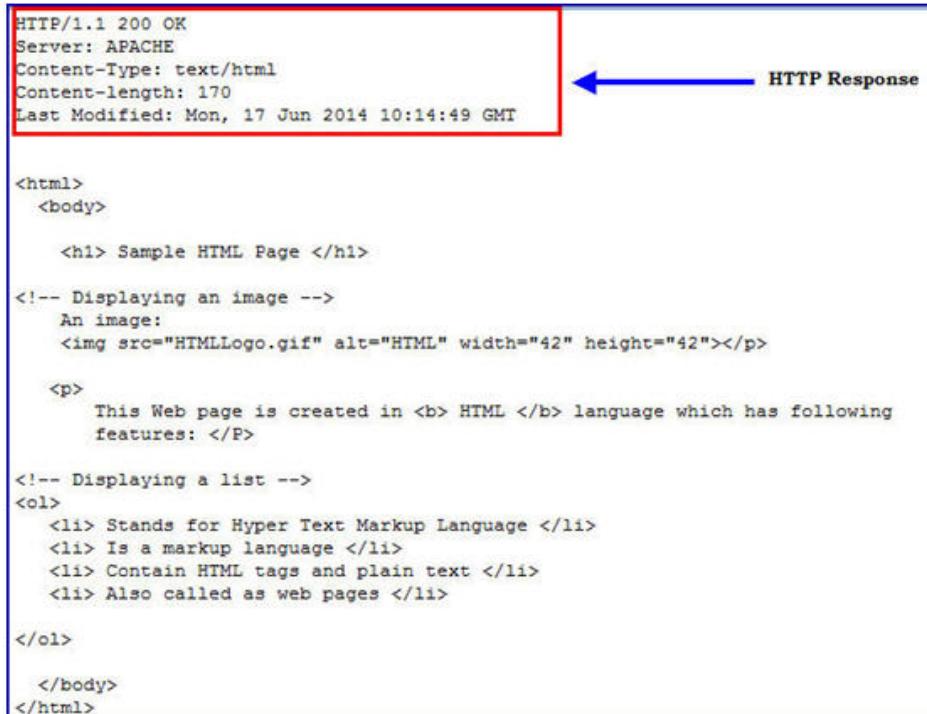
Status line

This element indicates status of the requested process.

Header information

The header information contains information such as server, last modified date, content-length, and content type. The server header specifies the software being used on the server. Last-modified header indicates the last modified date of the requested file. The content-length specifies the size of file in bytes. Content-type header specifies the type of document.

Figure 1.14 shows the sample message format sent in the response with the HTML content from the Web server.



```
HTTP/1.1 200 OK
Server: APACHE
Content-Type: text/html
Content-length: 170
Last Modified: Mon, 17 Jun 2014 10:14:49 GMT

<html>
  <body>

    <h1> Sample HTML Page </h1>

    <!-- Displaying an image -->
    An image:
    </p>

    <p>
      This Web page is created in <b> HTML </b> language which has following
      features: </P>

    <!-- Displaying a list -->
    <ol>
      <li> Stands for Hyper Text Markup Language </li>
      <li> Is a markup language </li>
      <li> Contain HTML tags and plain text </li>
      <li> Also called as web pages </li>
    </ol>

  </body>
</html>
```

Figure 1.14: Response Message Format

1.2.5 HTTP Methods

Web applications allow users to enter information using forms and send to the server for processing. The data entered into the fields in a form is sent to the Web server through URL. The data is processed on the Web Server and the appropriate response is generated which is sent back to the user.

For example, in a banking site, the user enters username and password. The information is sent to the Web server through the request parameters and after validating the data received with the database, the account information is sent as a response. The server treats the data entered in each field as a request parameter. The server can extract the values of each request parameter in the request for processing.

Figure 1.15 shows an example of request parameters.



Figure 1.15: Example of Request Parameters

The HTTP request messages uses the following HTTP methods for transmitting request data over the Web. The HTTP methods specified in the `request-line` element of the request indicate the type of operation to be performed on the resource.

The most commonly used HTTP methods are as follows:

GET

The `GET` method informs the server to retrieve information from the request URL. The information is passed as a sequence of characters that are appended at the end of the request URL. This forms a query string. For example, search engine such as `www.google.com` uses the `GET` method to retrieve search results for search strings entered by user.

Following is a sample query string constructed while sending user data in the request URL.

`http://www.abcBank.co.uk/search?name=john&pass=j7652`

The length of query string is restricted to 240 to 255 characters depending on the server. Thus, this method cannot be used to send large amount of data from the forms. Also, the data sent on the URL is visible; hence, this method is not reliable to send the important information such as credit card number or password.

POST

The `POST` method is used when sending information, such as credit card numbers or information to be saved in the database. Data sent using `POST` is in encrypted format and not visible to the client and there is no limit on the amount of data being sent. The data is passed to the server in the body of the HTTP message. Thus, the request cannot be bookmarked or emailed.

Figure 1.16 shows the request structure with HTTP POST method.

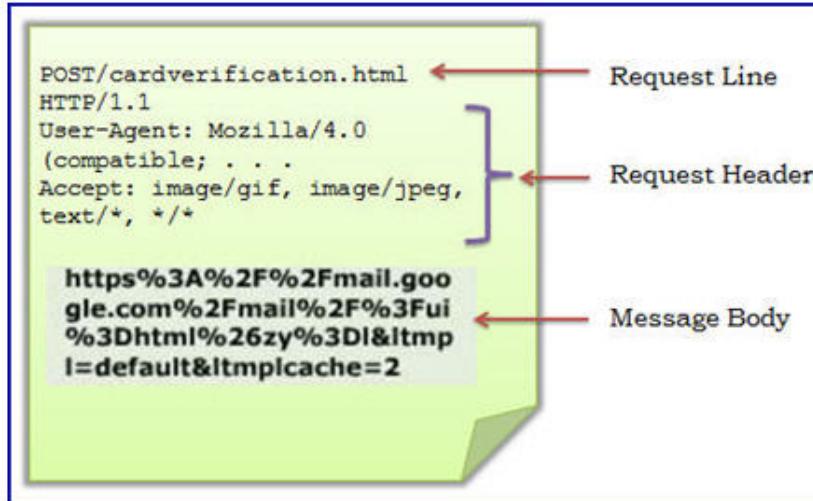


Figure 1.16: POST Method

HEAD

The **HEAD** method is functionally same as **GET**, except that the client uses **HEAD** method to receive only the headers of the response and not the message body. This method is advantageous when the user want to check characteristics of a resource without actually downloading the complete resource.

PUT

The **PUT** method is used to request the server to store the data enclosed in the HTTP message body at a location provided in the request URL.

DELETE

The **DELETE** method is used to request the server to delete file at a location specified in the request URL.

OPTIONS

The **OPTIONS** method can be used to query the server on the methods supported for a particular resource available on the server.

TRACE

The **TRACE** method is basically used for debugging and testing the request sent to the server. It asks the user to echo back the sent request. This method is useful when the request sent to the server reaches through the proxies.

1.3

Java in the Web

The potential of Java as a server-side development platform extends the power of the Web in creating Web applications. However, the first release of Java did not support the APIs for developing Web application, but was intended for use in embedded devices. Also, during that time, even the Web pages contain largely static content.

Soon with the development of Java Applets which support the execution of Java code within the browsers, Java came out as a powerful platform for developing Web components accessed on multiple platforms providing dynamic functionality to the Web pages.

Web components are functional software running on the Web server which handles the incoming HTTP request and provides responses.

Java Web application consists of the following Web components:

- Servlet:** Servlets are Java classes that dynamically process HTTP requests and construct responses.
- JSP:** A JSP page is a text document that creates dynamic Web content, usually to display the content in the Web browser.

The Java Web technology named Servlet formed a key component in Web development. However, before Servlets, there were technologies such as Common Gateway Interface (CGI), Active Server Pages (ASP), and so on which were practiced for generating dynamic content for the pages.

1.3.1 Prevalent Server-side Technologies

Currently, there are various server-side technologies, which execute various programs on the Web server. These are as follows:

- Common Gateway Interface (CGI)
- Proprietary Web server API (ISAPI)
- Server-side JavaScript
- Active Server Pages
- PHP
- Java Servlets
- Java Server Pages (JSP)

1.3.2 CGI - Solution to Dynamic Content Generation

CGI is one of the most popular server-side technologies. CGI is a set of standards that specify how data is transferred between the Web server and server-side CGI programs. The data provided by the client in the Web page is sent from HTTP server to the gateway or CGI program. The CGI program processes the data and returns the result to the Web server which in turn sends it to the client.



A program that gives dynamic output thereby executing in real time is written at the server-side using standards of CGI.

Figure 1.17 depicts the server process for running CGI.

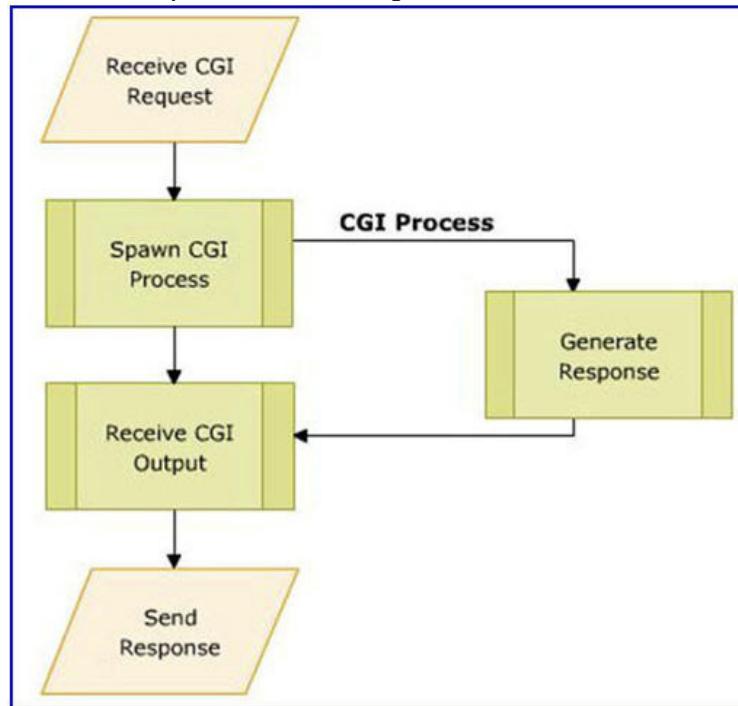


Figure 1.17: Server Process for Running CGI

When a Web server receives a request that needs to access a CGI program, it creates a new instance or process of the CGI program and then passes the client data to it. This means for every new request, the server creates a separate instance of CGI program which is responsible to generate a response for its request.

Figure 1.18 depicts the working of CGI program.

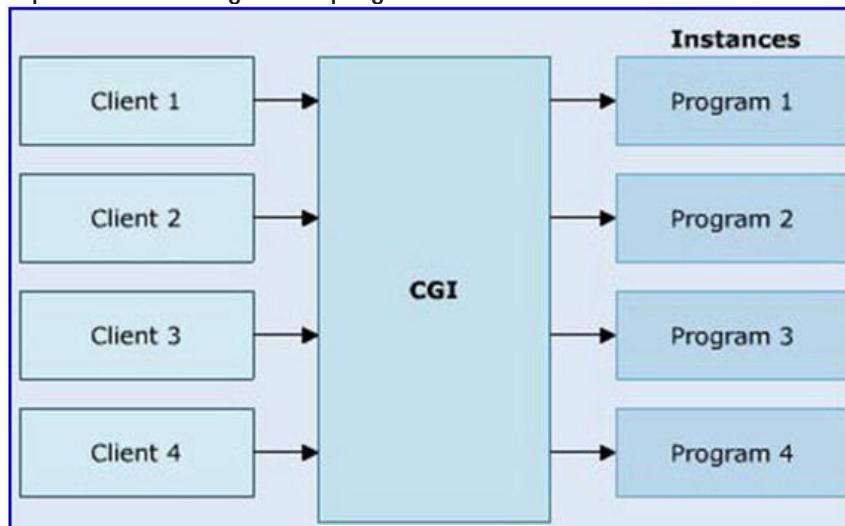


Figure 1.18: Working of CGI Program

Creating a process for each request occupies server resources, thus, limiting the number of requests a server can handle concurrently.

CGI as a server-side scripting language has certain disadvantages as described here:

Reduced efficiency

Number of processes that can be executed by a Web server is limited to each server. Each time a process is executed, different instances of the same processes is created on the server, thereby, resulting in overloading of server and in reducing efficiency of server.

Reloading Perl interpreter

The widely accepted platform for writing CGI script is Perl. Each time the server receives a request, the Perl interpreter needs to be reloaded. This reduces the efficiency of the server.

1.3.3 Proprietary Web Server APIs (ISAPI, NSAPI)

Microsoft and Netscape, each developed their own APIs that allows developers to write server applications very quickly. The server application can be used as shared libraries. Several libraries are loaded into the same process area with the Web server. These libraries are able to service multiple requests without creating a new process. These can either be loaded when they are needed. Once any of these have been idle for a set amount of time, they are unloaded by the Web server from the memory.

1.3.4 Server-side JavaScript

The server-side uses JavaScript as its scripting language. It is an extension of the core JavaScript language with features such as database access, session management, inter-operability with server-side Java classes, using Netscape LiveWare technology. The compiled server-side JavaScript is not platform specific, however, specific to Netscape's HTTP servers.

1.3.5 Servlet - Solution to CGI Problems

Servlet are Java codes that are used to add dynamic content to Web server. The processing of Servlet is very similar to CGI program, as it also responds to HTTP request and generates dynamic response. However, it is different in its execution.

There is only a single instance of Servlet created on the Web server. To service multiple clients' request, the Web server creates multiple threads for the same Servlet instance. Each thread handles request received from the client and generates response that is sent to the Web engine which in turn sends the response to the client.

Figure 1.19 shows the working of a Servlet.

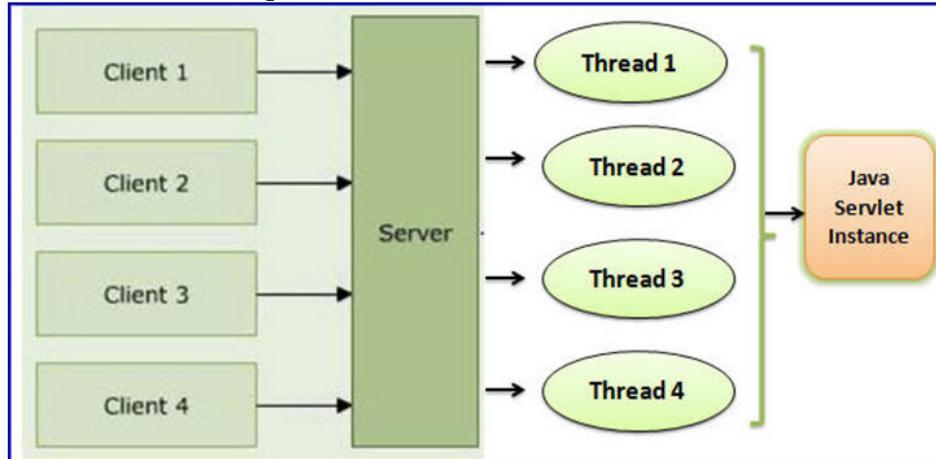


Figure 1.19: Working of a Servlet

The initialization code in Servlets is executed only once at the beginning of the program and is refreshed automatically on receiving separate requests. Whereas in CGI, separate instances of the program are being invoked each time a process is executed.

The multi-threading property of servlets helps in handling separate requests by allocating resource to separate threads. This gives a boost to the performance, thereby, increasing the efficiency of servlets.

1.3.6 Merits of Servlet

Servlets are written using Java, which makes extensive use of Java API. The merits of servlets are as described follows:

- **Enhanced efficiency**

Servlet is required to be executed only once at the beginning with the initialization code. Thereafter, it gets auto refreshed each time a request is sent for execution.

- **Ease of use**

Servlets does not have too many complexities. It is just basic Java along with HTML implemented within the Java code, which increases the ease of use.

- **Powerful**

The usage of Java APIs makes the Servlets a powerful server-side scripting language.

- **Portable**

Java is platform independent and since Servlet uses Java code for writing scripts, it can be executed in any platform.

- **Safe and cheap**

Usage of Java codes provides high security of data to be sent and received thereby maintaining the safety.

1.4 Developing Web Applications

An application registered into World Wide Web (WWW) and accessible to its users from a public server is a Web application. Java Web application comprises servlets, JSP pages, images, HTML files, JavaBeans, Applets, and Java classes.

If a Web application is to be ported to some other server or system, a developer has to copy or move all these files to the new system. However, an easy alternative is to package all the files associated with Web application into a single Web archive file (war) and deploy this war file on the Web server.

The Java Web applications are deployed on a Java-enabled Web server such as Tomcat. Every Java Web server contains a Web container which is responsible for handling the Web components present in the Web application.

Figure 1.20 shows a Web server containing a Web container used for processing the HTTP request and HTTP response for the accessed Servlet or JSP.

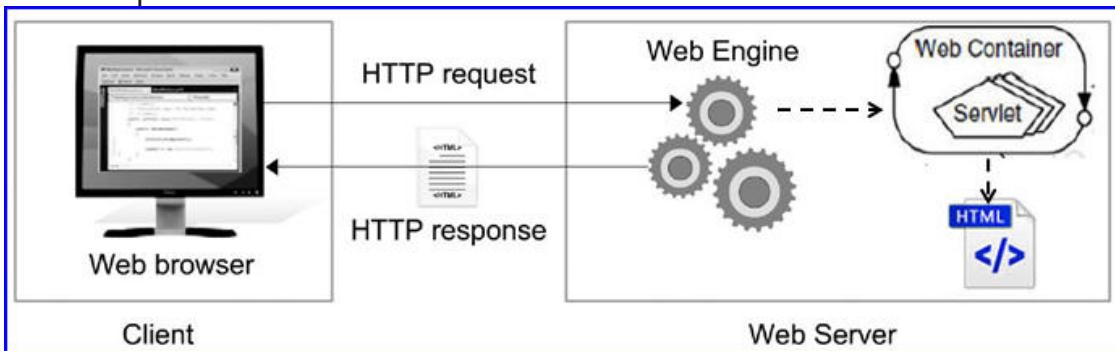


Figure 1.20: Java Web Server

1.4.1 Web Container

The Web container is a program that manages execution of Servlets and JSP pages. The Web container takes request from a Web server and passes it to a servlet for processing. It manages the servlet life cycle and other services such as security, transaction, and so on for the Web components. The performance of a servlet depends upon the efficiency of the Web container. The Web container is also referred as Servlet container.

Figure 1.21 depicts a Web container.

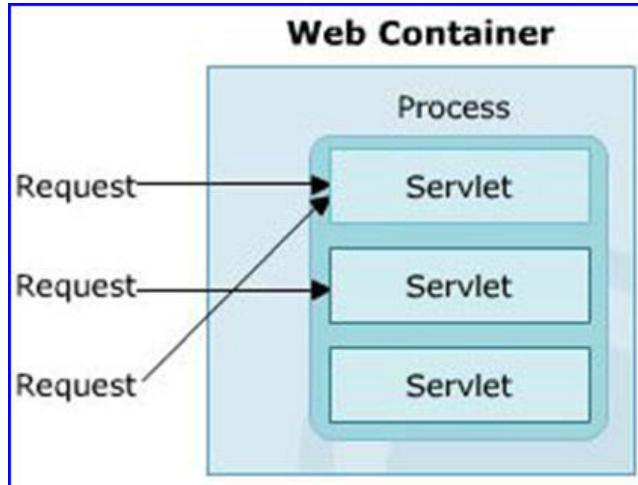


Figure 1.21: Web Container

1.4.2 Web Application Life Cycle

Web applications are needed to create a dynamic application using different Web components such as Java Servlets, JSP, HTML, Images, and JavaScript helper classes and libraries.

The process or life cycle for creating, deploying, and executing a Web application can be summarized as follows:

- Develop the Web component code.
- Develop the Web application deployment descriptor.
- Compile the Web application components and helper classes referenced by the components.
- Optionally package the application into a deployable unit.
- Deploy the application into a Web container.
- Access a URL which references the Web application.

Figure 1.22 describes the Web application life cycle.

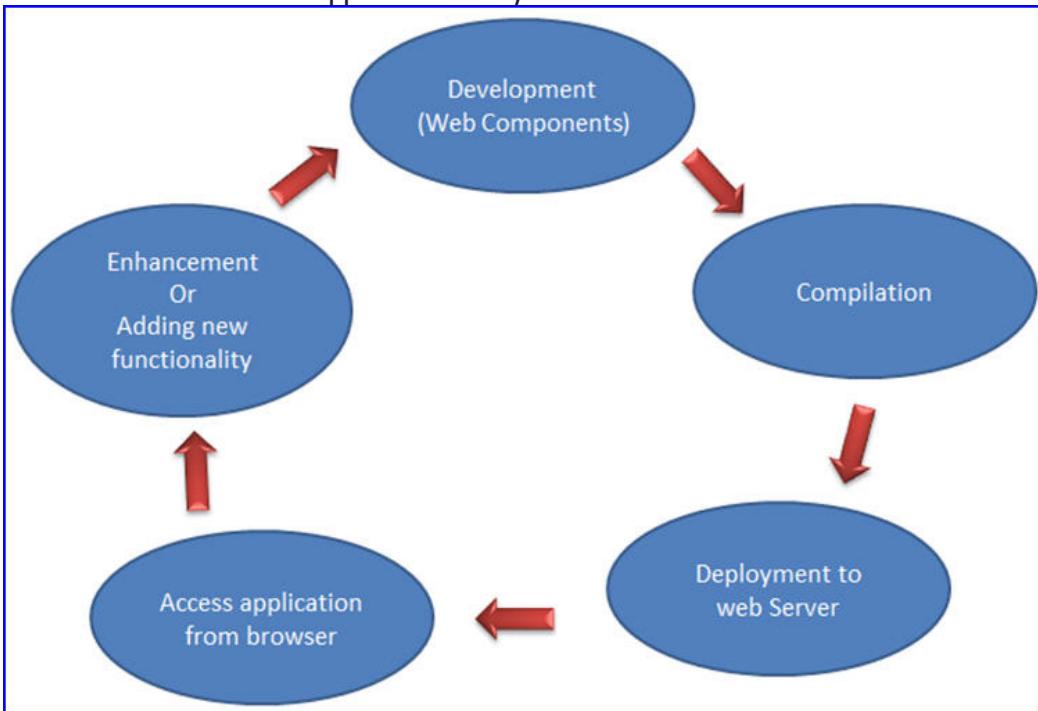


Figure 1.22: Web Application Life Cycle

1.5

Packaging Web Applications

A Web application is composed of Web pages and Web components which are collectively referred as Web resources. These files are to be placed in the correct structure, so that the files are located properly by the Web engine for correct functionality.

Java applications are packaged as Java Archive (JAR) which contains multiple classes and other related resources. All the components required in the Java application are packaged into a single jar file. Similar to Java applications, the Web applications also need to be packaged with Web resources before the application is deployed on the Web server.

The package file used for a Web application has a specific set of directory structure which helps the Web container to ensure proper functionality of the application. The Web applications are therefore, packaged in a Web Archive (WAR) which distinguishes it from the normal Java application. The Web application resources are packaged in the .war file.

1.5.1 Web Application Context Root

The Web container supports multiple Web applications, so each application is identified or assigned with a context root. The context root or context path of the Web application is the base URL used for locating Web pages and Web components such as Servlet and JSP within the application.

For example, consider a .war file named WebApp contains a file named index.html in a page folder. When the war file is deployed on the Java Web server, then you need the URL to access the pages from the application.

Suppose the application is deployed on the local Web server, then the URL to access the index.html page will be `http://localhost:8080/WebApp/page/index.html`. Thus, the context root of the deployed application is the /WebApp.

1.5.2 Web Application Directory Structure

As discussed, the contents of the Web application are accessed from the application's context root. The context root of the Web application contains two main components in the directory structure. These are described as follows:

- Static files**

All the HTML pages and images comprise the static files. Images can be collectively stored in an images directory. For example, to access a file `index.html` in the book folder on the Website `www.corejava.com`, the URL would be `http://www.corejava.com/book/index.html`.

- WEB-INF**

This directory exists within the context root. However, this directory cannot be referenced. For example, referencing `www.corejava.com/book/WEB-INF` is not possible.

Figure 1.23 shows the directory structure of the .war file.

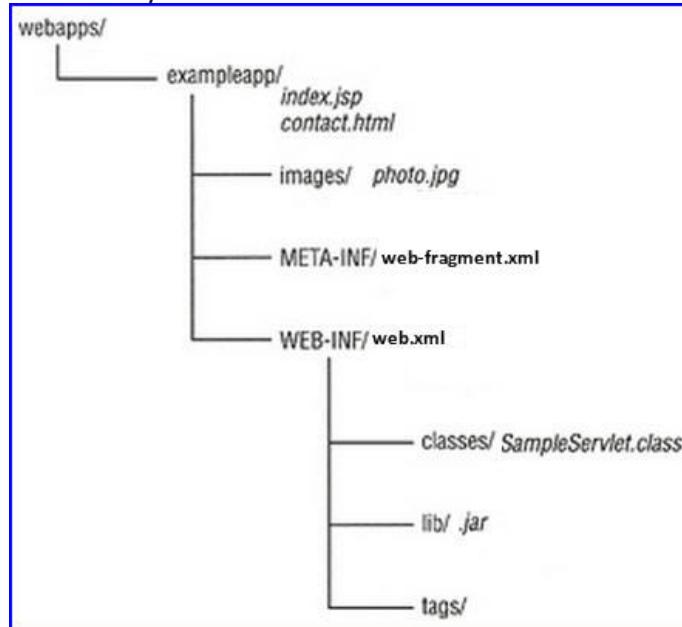


Figure 1.23: WAR Directory Structure

The WEB-INF directory consists of the following contents:

- classes directory**

Servlet classes, JavaBean classes, and any other classes required by the Web applications are stored in this directory. Servlets and JavaBean classes that are part of a package are placed to match their package names. For example, if a servlet or JavaBean class is part of a package such as `www.bookstore.com`, then the corresponding class file will be placed in `classes/com/bookstore` directory.

□ web.xml

This file is called the deployment descriptor of the Web application. It is basically an XML structured file which provides configuration information for the components of the Web application. The configuration information includes information, such as the default pages to show, mapping Servlets with their URLs, and so on.

□ lib directory

It contains .java archive files required for Web application such as database drivers. In other words, the library JAR files used by the application are stored within the WEB-INF/lib directory.

□ tags directory

This directory contains tag files, which provide implementation for custom tags. Tag files have .tag extension.

□ Tag Library Descriptor files

These files provide information about the attributes of a custom tag and the class to be invoked when the tag comes across in a JSP page. These files have .tld extension. However, tag library descriptor files are required only if tag files need to be distributed as a separate package or if the custom tags are implemented using Java.

1.5.3 Deployment Descriptor

A deployment descriptor is a configuration file which describes how the Web application should be deployed. It is written using XML with name `web.xml` and placed under the `WEB-INF` folder, that is, `WEB-INF/web.xml`.

When the Web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the Servlet that handles the request. For example, to map a URL request to a Servlet, you must do the following settings in the `web.xml` file:

1. Declare the servlet with the `<servlet>` element.
2. Define a mapping from a URL path to a Servlet declaration with the `<servlet-mapping>` element.

The `<servlet>` element declares the Servlet, which including a name used to refer to the Servlet by other elements in deployment descriptor and the Servlet class. The name for each Servlet must be unique across the deployment descriptor.

The `<servlet-mapping>` element specifies a URL pattern and the name of a declared Servlet (declared in `<servlet>` element) to use for requests whose URL matches the pattern.

Code Snippet 1 shows the configuration setting for the `HelloServlet` class created in the Web application named `FirstWebApplication`.

Code Snippet 1:

```
<!-- web.xml -->
<?xml version="1.0" encoding="UTF-8"?>
...
<servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>Mypkg.HelloServlet</servlet-
class>
</servlet>
<servlet-mapping>
    <servlet-name> HelloWorldServlet</servlet-name>
    <url-pattern>/sayhello</url-pattern>
</servlet-mapping>
</web-app>
```

As shown in the given code snippet, the URL pattern to access the `HelloServlet` class will be `http://localhost:8080/FirstWebApplication/sayhello`.

Figure 1.24 shows the mapping of Servlet with the URL through which it is accessible to the application.

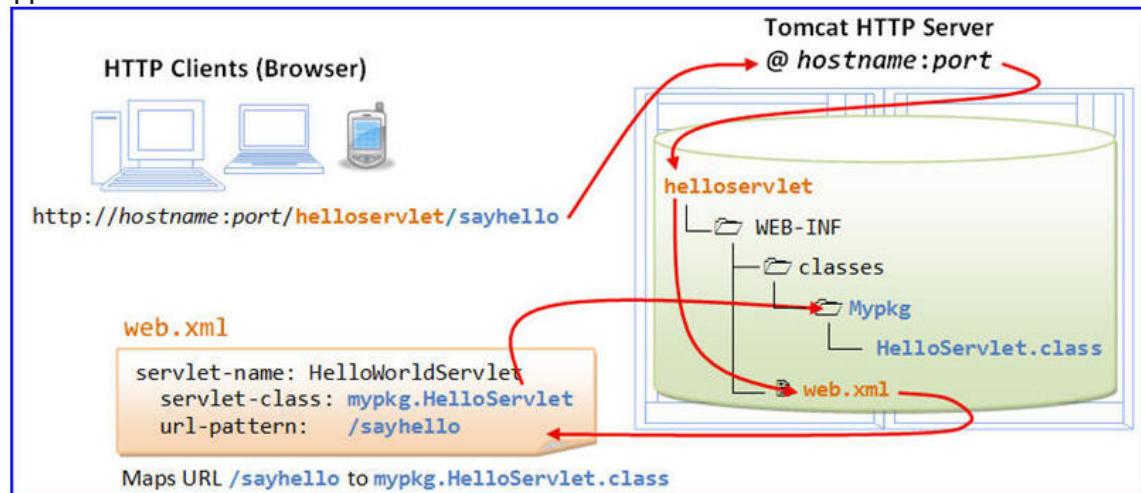


Figure 1.24: Servlet Mapping in `web.xml`

From Java EE 6 onwards, the `web.xml` file is optional, as the configuration information can be applied with the annotations. Annotations are meta information embedded in the Java source files. The meta information creates the dynamic configuration setting for the deployed component during runtime.

Servlet 3.0 has also introduced a new feature called **Web fragments**. Web fragment is the mechanism to include deployment descriptor information which is specific to a library. This helps to segregate the unrelated information from the main deployment information.

The developer might have created functionality for the application separately in some other Java EE project and import its `.jar` in `WEB-INF\lib` as plug-in within the Web application.

The content of `web.xml` and `web-fragment.xml` are almost the same, but the difference in its location, we put `web.xml` in `WEB-INF\web.xml` and `web-fragment.xml` in `META-INF\ web-fragment.xml`.

1.6

Developing Web Application in NetBeans IDE

The developer can create and work with Web applications using different frame works. Mainly, NetBeans Integrated Development Environment (IDE) is used to create Java EE based projects with JSP's and Servlets. NetBeans IDE is a software program which is designed to help programmers and developers build and test Java applications. NetBeans IDE provides text or source code editor, debugger, and compiler all in one in a single environment.

1.6.1

NetBeans IDE

Some of the features of the NetBeans IDE are as follows:

- A source code editor is similar to an HTML text editor. This is where programmers write the source code for their programs.
- A compiler compiles the source code into an executable program and an interpreter runs programs and scripts that don't need to be compiled.
- Build automation tools help automate the processes that need to happen with most software developments such as debugging, compiling, and deployment.
- A Debugger helps to pin-point the exact spot where there is problem or error in the source code.

1.6.2

Creating a New Project

To create a new project:

- Open NetBeans IDE and select **New Project** from the **File** menu.
- Select **Java Web** from the **Categories** list.
- Select **Web Application** from the **Projects** list.
- Click **Next** to specify the name and location.

Figure 1.25 shows how to create a project.

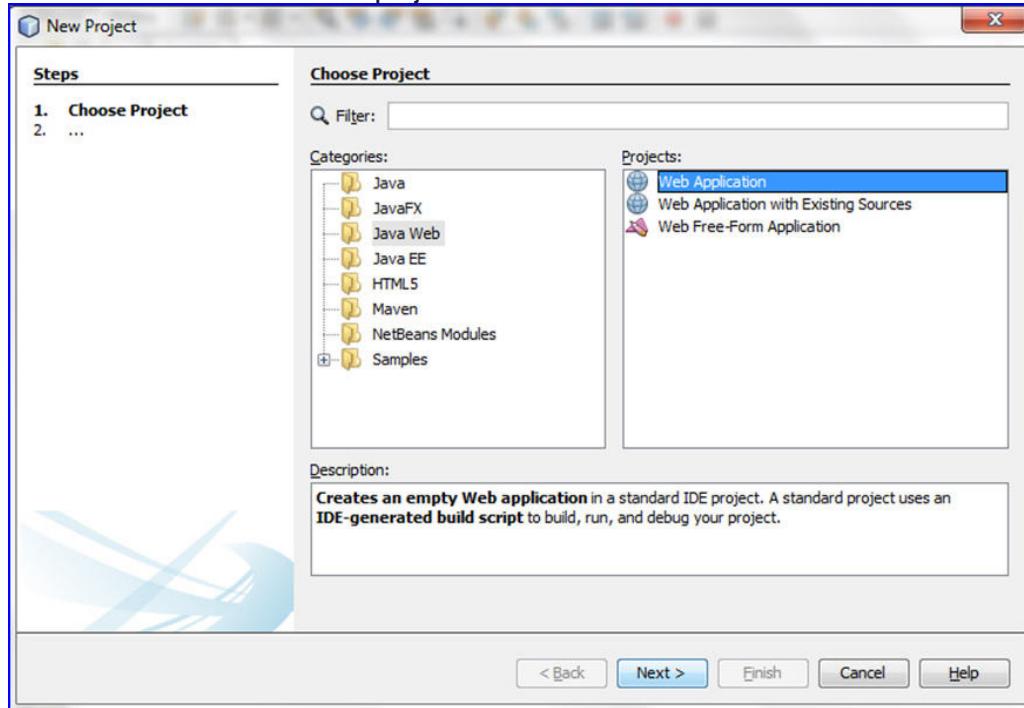


Figure 1.25: Creating Project in NetBeans

1.6.3 Specifying Name and Location for the Project

To specify the name and location for the project:

- Enter the name of the project, for example, **FirstWebApp**.
- Click the **Browse** button and select a location for saving the project.
- Click **Next** to specify the server used for deploying Web application.
- Click **Server** drop-down list and select the Web server for the application. If Web server is not available, then you can add a new one (for current application we will be using Apache Tomcat 7.0).

Figure 1.26 shows how to select a Web server for the project.

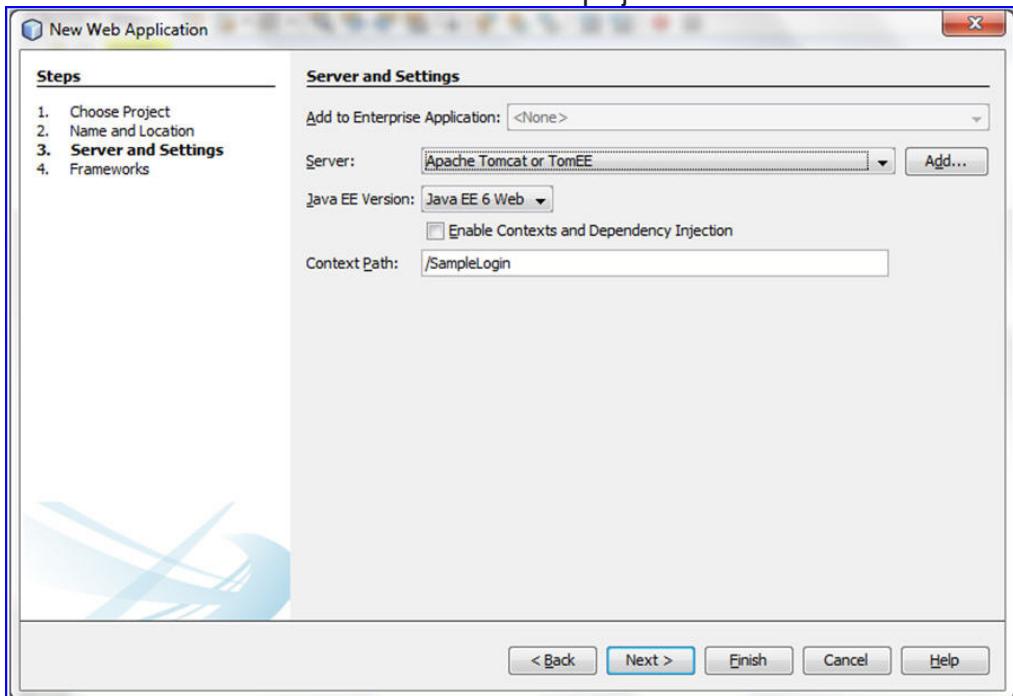


Figure 1.26: Specifying a Web Server

- Click **Next** and select frameworks used in Web application. Currently, the Web application is not using any framework.
- Click the **Finish** button.

Figure 1.27 shows the structure of the **FirstWebApp** project created in NetBeans IDE.

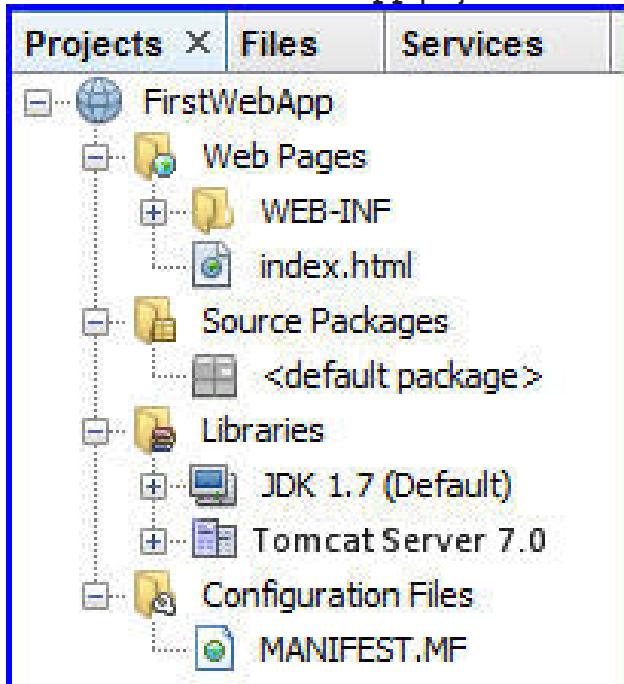


Figure 1.27: Web Project Structure in NetBeans IDE

1.6.4 Writing the Code for Servlet

- Right-click the **Source Packages** and select **Servlet** from the context menu. This opens **New Servlet** window.
- Type the name of the Servlet as **FirstServlet** and package as **com.controller** as shown in figure 1.28.

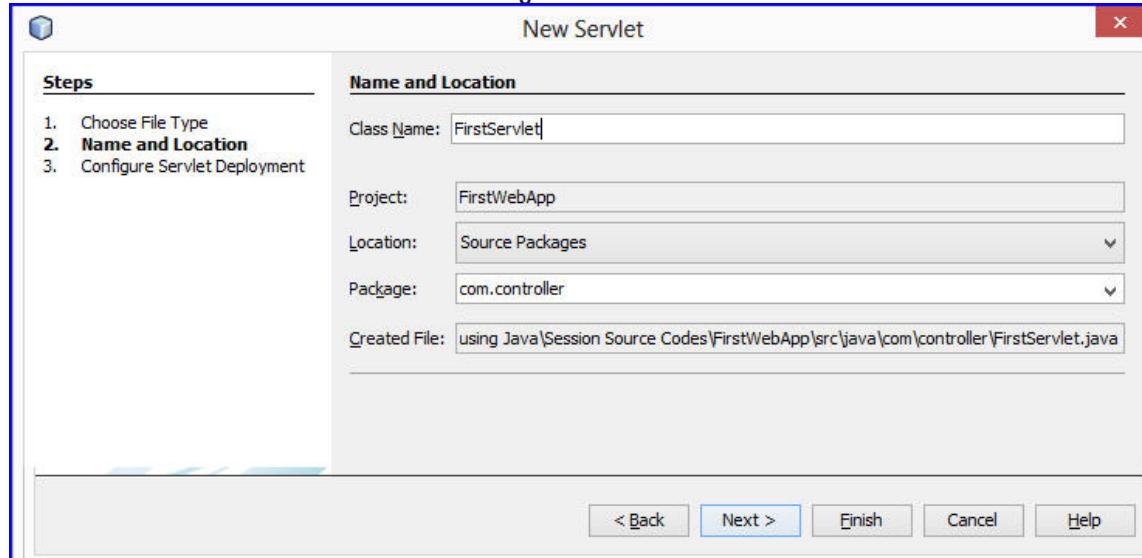


Figure 1.28: Create a Servlet within the Project

- Click the **Finish** button.

Then, a new Java class in the package needs to be created. For example, a class named as 'FirstServlet' is created within the package `com.controller`.

Code Snippet 2 shows the code generated for the 'FirstServlet' class.

Code Snippet 2:

```
import javax.servlet.http.*;
public class FirstServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
```

```
out.println("<title>ServletFirstServlet</title>");  
out.println("</head>");  
out.println("<body>");  
out.println("<h1>ServletFirstServlet at " + request.  
getContextPath() + "</h1>");  
out.println("</body>");  
out.println("</html>");  
}  
...  
  
protected void doGet(HttpServletRequest request,  
HttpServletResponse response) throws ServletException,  
IOException {  
    processRequest(request, response);  
}  
...  
  
protected void doPost(HttpServletRequest request,  
HttpServletResponse response) throws ServletException, IOException  
{  
    processRequest(request, response);  
}  
}
```

As shown in the given code snippet, the `out` object of `PrintWriter` class is created within the `processRequest()` method in NetBean IDE. This method is invoked by `doGet()` and `doPost()` method based on how the data is sent by the client in HTTP request.

The `println()` method of `out` object is used to display the message string on the Web page. Change the `out.println("<h1>Servlet FirstServlet at " + request.getContextPath() + "</h1>");` with this `out.println("<h1> Hello World </h1>");`

1.6.5 Creating the Deployment Descriptor

Code Snippet 3 shows the `web.xml` file within the `Web-INF` directory. The `web.xml` file is updated with the configuration settings of the `FirstServlet`.

Code Snippet 3:

```
 . . .
<?xml version="1.0" encoding="UTF-8"?>
<servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>com.controller.FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FirstServlet</servlet-name>
    <url-pattern>/myservlet</url-pattern>
</servlet-mapping>
</web-app>
```

1.6.6 Build and Execute the Web Application

To run the application, right-click the project and select **Run** from the context menu. When you run the application, it first compile the Servlet code and then deploy the resource to the Web server, if Web server is not running, first start the server and then deploy the Web application to server.

After building and deployment is completed successfully, NetBeans will launch the Servlet page in the browser as shown in figure 1.29.



Figure 1.29: Output of Servlet in Web Browser

Check Your Progress

1. Identify the features of a Web client-server model.

(A)	The architecture of Web client-server model can be one-tier, two-tier, three-tier, or N-tier.
(B)	Web applications have higher maintenance cost.
(C)	Code related to presentation, business, and data access logic are all clubbed together in the one-tier architecture of Web application.
(D)	Web-applications are platform independent.

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

2. Which of the following statements about Request and Response message structure are true?

(A)	Request message structure consists of only header information.
(B)	Header information contains information such as server, last modified date, content-length, and content type.
(C)	Request line returns the User-Agent along with the status code header.
(D)	Response message structure consists of status line and header information.

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

3. Identify the correct feature of the HTTP request GET methods.

(A)	The GET method is used to receive information over the Web.
(B)	Data sent using POST method is not encrypted.
(C)	The POST method is used to receive response over the Web.
(D)	The length of query string is restricted to 240 to 255 characters in GET method.

Check Your Progress

4. Which of the following options is a limitation that is overcome by a Web server?

- | | |
|-----|--|
| (A) | Username and password are retained by the browser. |
| (B) | Each time user is prompted for username and password for accessing subsequent pages. |
| (C) | Dynamic Web content can be managed from the client end. |
| (D) | It is feasible to maintain static Web content from the customer's end. |

5. Which of the options are disadvantages of using CGI?

- | | |
|-----|----------------------------|
| (A) | Reduced Efficiency |
| (B) | Ease of use |
| (C) | Reloading PERL interpreter |
| (D) | Safe and Cheap |

6. Identify the merits and demerits of servlets.

- | | |
|-----|---|
| (A) | Servlets are easy to use. |
| (B) | Servlets are portable. |
| (C) | Servlets are less efficient than other scripting languages. |
| (D) | Servlets are safe and less costly to develop. |

- | | | | |
|-----|------|-----|---------|
| (A) | A, C | (C) | A, B, C |
| (B) | B, D | (D) | A, B, D |

7. Which of the following statements, about the deployment descriptor are true?

- | | |
|-----|--|
| (A) | Instructs a deployment tool to create specific configuration requirements for deployment of a servlet. |
| (B) | Written using HTML syntax. |
| (C) | Placed under the WEB-INF directory. |
| (D) | Stored in web.xml file. |

Answer

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

Summary

- An application is a collection of programs designed to perform a particular task. Different types of applications are designed for different purposes.
- A Web application is a software application that runs on a Web server. The Web application basically has three components: the presentation layer, the application layer, also known as business layer, and the data access layer.
- The most common technologies for communication on the Web are HTML and HTTP. HTML is a presentation language which enables Web designer to create visually attractive Web pages.
- The requests and responses sent to a Web application from one computer to another computer are sent using HTTP. HTTP does not store any connection information about the page and hence, it is referred to as a stateless protocol.
- The HTTP request messages uses the HTTP methods for transmitting request data over the Web.
- Java Web application comprises servlets, JSP pages, images, HTML files, JavaBeans, Applets, and Java classes.
- The Web applications are packaged in the .war file which allows the content of the Web application are accessed from the application's context root.
- A deployment descriptor describes how the Web application should be deployed. It is written using XML with name web.xml and placed under the WEB-INF folder, that is, WEB-INF/web.xml.
- Java Web applications are developed in NetBeans IDE which provides fully-integrated environment for building and executing the Web applications.



Welcome to the Session, **Java Servlet**.

This session explains the Java Servlet and its architecture. It describes the Java Servlet API and also explains the life cycle of a Servlet. Further, the session introduces RequestDispatcher interface that help in Servlet communication. It also explains how to work with the Servlet initialization parameters and handle errors in the Servlet.

In this Session, you will learn to:

- Explain the Servlet API
- Explain the servlet architecture and life cycle of Servlet
- Explain the methods of ServletRequest and HttpServletRequest interfaces
- Explain the methods of ServletResponse and HttpServletResponse interfaces
- Describe the use of response headers
- Explain how to read text and binary data from a request
- Explain the ServletConfig and ServletContext interface
- Explain redirection of client requests
- Explain RequestDispatcher interface
- Explain error handling in Servlet

2.1 Introduction

A Servlet is a server-side program written in Java programming language. A Servlet is a Web component that processes the client requests and generates dynamic content in response. Normally, the generated response contains HTML content that is sent back to the client.

Servlet is protocol independent which means different types of clients can access Servlet. The different types of clients to a Servlet can be a Web client which send request through HTTP protocol or it can be a Web service endpoint sending request data through SOAP protocol.

2.2 Servlet API

All the classes related to developing and managing Servlets are provided in two packages namely,

- javax.servlet
- javax.servlet.http

- javax.servlet

The `javax.servlet` package contains generic interfaces and classes that are implemented and extended by all Servlets. This package basically provides a framework for the Servlet operations.

It includes an interface named `Servlet` which defines the life cycle methods for a Servlet. Every Servlet must implement the `javax.servlet.Servlet` interface directly or indirectly (by extending the class which already implements this interface).

- javax.servlet.http

The `javax.servlet.http` package contains classes and interfaces used for developing a Servlet that works with HTTP protocol.

Figure 2.1 displays the Servlet API hierarchy.

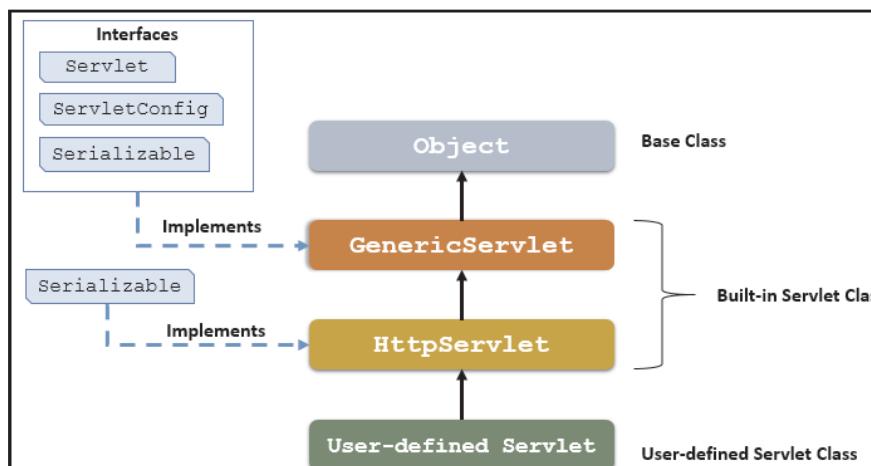


Figure 2.1: Servlet Hierarchy

The `javax.servlet` package contains an abstract class named `GenericServlet` which inherits from `Object` class and helps to design a protocol independent Servlet. You can inherit the `GenericServlet` class to design your own Servlet.

The `GenericServlet` implements three interfaces namely,

- `Servlet` – This interface defines the life cycle methods for a Servlet.
- `ServletConfig` – This interface defines methods that are used by Servlet container to pass information to a Servlet instance during its initialization.
- `Serializable` – This interface is defined in `java.io` package to serialize the state of an object.

Consider a situation where you need to develop a login Servlet to validate the user's information. Code Snippet 1 demonstrates how to create a protocol-independent login servlet.

Code Snippet 1:

```
public class LoginServlet extends GenericServlet{  
    ...  
}
```

In the given code, the class named `LoginServlet` is a generic servlet.

To develop a Servlet which is accessible only through a Web browser, you can create an `HTTP Servlet` by extending `HttpServlet` class. The `HttpServlet` class is a subclass of `GenericServlet` and enables to create an HTTP-based Servlet as part of a Web application.

Code Snippet 2 shows the creation of login servlet to be accessed on the Web.

Code Snippet 2:

```
public class LoginServlet extends HttpServlet{  
    ...  
}
```

In the given code, the `LoginServlet` is defined as HTTP servlet, so it will be accessed through HTTP protocol.

2.3

Servlet Life Cycle

The life cycle define how the servlet is loaded, initialized, handles requests from clients and is taken out of service.

- Instantiation**

In this stage, the servlet container creates an object of the servlet.

- **Initialization**

When the Servlet is instantiated, it is in initialization stage. In this stage, the servlet container calls the `init()` method which begins the life cycle of the servlet.

- **Service**

In this stage, the servlet processes the request given by the client and generates the dynamic response to be sent back to the client.

- **Destroy**

In this stage, the servlet container destroys the servlet instance. Some of the conditions where a Servlet instances is destroyed includes: an explicit request from administrator, Web server shutting down, or a Servlet instance is idle and not processing any request.

- **Unavailable**

In this stage, the servlet container frees up the memory occupied by the servlet.

Figure 2.2 depicts the servlet life cycle.

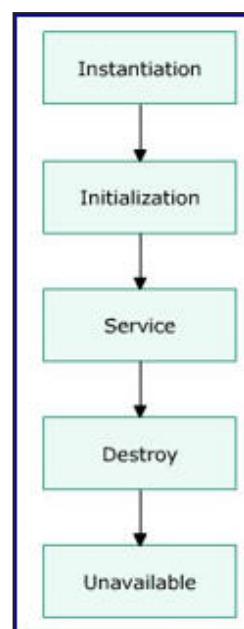


Figure 2.2: Servlet Life Cycle

2.3.1 Servlet Life Cycle Methods

The servlet life cycle is defined by the following methods:

- **init()**

The method initializes the servlet. The life cycle of the servlet begins with the `init()` method. The `init()` method is called only once by the server and not again for each user request.

service()

The method processes the request made by the client. It passes the `ServletRequest` and `ServletResponse` objects which collect and pass information such as the values of the named attributes, the IP address of the agent or the port number on which the request was received.

destroy()

The method destroys the servlet interface, if there are no more requests to be processed. A server calls this method after all requests have been processed or a server-specific number of seconds have passed, whichever comes first.

2.3.2 Servlet Architecture

The Servlet container is responsible for handling Servlet execution which is based on multithreaded model. This means, when the container receives the request for the same Servlet, it creates a new thread that handles the execution of the Servlet and dies once the execution is completed.

Figure 2.3 shows the handling of multiple clients request by a Servlet.

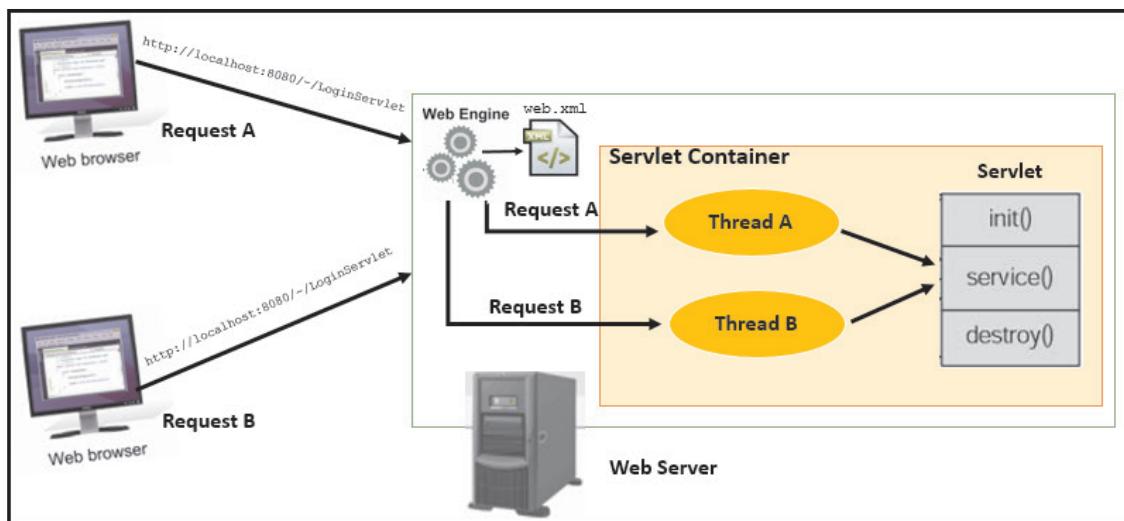


Figure 2.3: Servlet Multithreaded Model

As shown in figure 2.3, the flow of information in a Servlet is as follows:

1. The Web browser sends an HTTP request to the Web server through a Uniform Resource Locator (URL). For example, a URL, `http://localhost:8080/.../LoginServlet` is sent to the Web Server.
2. On receiving the request, the Web engine looks into the `web.xml` deployment descriptor file to match the URL with the registered Servlets.
3. Once the requested Servlet is found, the Web engine forwards it to the Servlet container which is responsible for instantiating the Servlet instance.

4. The container first locates the servlet class, loads the servlet, and creates an instance of the servlet in the memory.
5. After the Servlet is loaded by the Servlet container, it is initialized completely and then the client request is processed.

To service multiple clients' request, the servlet container creates multiple threads for the same Servlet instance. Each instance handles request received from the client and generates response that is sent to the Web engine which in turn sends the response to the client.

2.4 Handling Servlet Request

When a servlet is requested to handle a request, it needs specific information about the request so that it can process the request appropriately. The `ServletRequest` interface provides access to this specific information.

When a servlet handles a request, the server passes a request object, which implements the `ServletRequest` interface. With this object, the servlet can find out information about the actual request such as protocol, Uniform Resource Locator (URL) and type. It can access parts of the raw request such as headers and input stream. It can also retrieve client-specific request parameters such as data entered in online forms.

Figure 2.4 depicts a Servlet request.

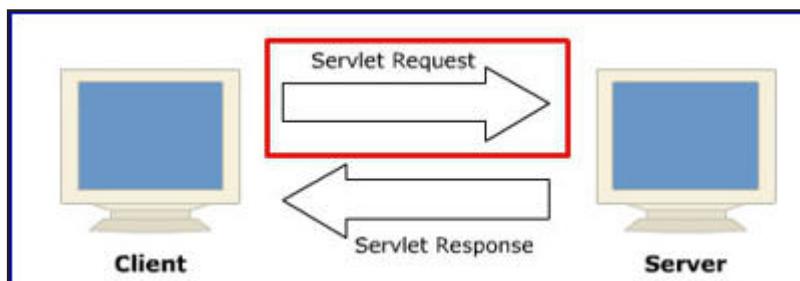


Figure 2.4: Servlet Request

2.4.1 ServletRequest Interface

The `ServletRequest` interface defines an object that makes client request information available to servlets. The `ServletRequest` object is passed as an argument to the `service()` method of the servlet.

Some of the methods defined in the `ServletRequest` interface are as follows:

```
public String getParameter(String s)
```

Returns the value of a specified parameter that is sent along with the request information. For example, the statement, `String name1 = request.getParameter("names1");` retrieves the value from the parameter `names1` passed in the request.

- `public Object getAttribute(String name)`

The method retrieves the value of an attribute specified by name, that was set using the `setAttribute()` method and returns null when no attribute with the specified name exists. For example, `Object cobj1 = cnt1.getAttribute("obj1");`

- `public int getContentLength()`

Returns the length of content in bytes and returns -1, if the length is not known.

Code Snippet 3 shows the use of `getContentLength()` method.

Code Snippet 3:

```
// The length of the content is passed to len1 and compared with the
// size of file limit

if((len1 = request.getContentLength()) > fileLimit) {
    System.out.println ("Name1 "+login+" Id1 "+id +
        "Expected Upload size is more than specified file Limit");
}
```

- `public ServletInputStream getInputStream() throws IOException`

Returns the binary data of the body of request, requested by the client and stores it in a `ServletInputStream` object. For example, to read the binary data, the following statement can be used:

```
ServletInputStream inStream1 = request.getInputStream();
```

- `public String getServerName()`

Returns the host name of the server to which the client request was sent. For example,
`String serverName = request.getServerName();`

2.4.2 HttpServletRequest Interface

The `HttpServletRequest` defines an `HttpServletRequest` object, which is passed as an argument to the `service()` method.

The methods defined by the `HttpServletRequest` interface are as follows:

- `public Cookie[] getCookies()`

Returns an array containing the entire `Cookie` objects and returns null if no cookies were found. For example, `Cookie[] cookies = request.getCookies();`

- `public String getHeader (String name)`

Returns the value of the specified request header as a String and returns null if the request did not include a header of the specified name.

For example, `out.println(request.getHeader("host") + "
");` returns the value of the header 'host'.

- `public String getMethod()`

Returns the name of the HTTP method used to make the request. For example, GET, POST, and PUT. For example, `out.println("<td align=center>" + request.getMethod() + "</td></tr>");`

- `public String getPathInfo()`

Returns the path information associated with a URL. For example, `String ses_id1 = req.getPathInfo();`

- `public String getAuthType()`

Returns the basic authentication scheme used to protect the servlet from unauthorized users. For example, `out.println(request.getAuthType());`

2.4.3 Reading Parameters from Request

Request parameters are strings sent by the client to a servlet container as a part of its request. These parameters are stored as a set of name-value pairs. Following methods of the `ServletRequest` interface are available to access parameters:

- `public java.lang.String getParameter(java.lang.String name)`

Returns the value of a request parameter as a String, or null if the parameter does not exist. For HTTP servlets, parameters are contained in the query string or posted form data. For example, `String name= request.getParameter("names");`

- `public java.util.Enumeration getParameterNames()`

Returns an enumeration of string objects containing the names of the parameters of this request. If the request has no parameters, the method returns an empty enumeration. For example, `Enumeration Books = req.getParameterNames();`

- `public java.lang.String[] getParameterValues(java.lang.String name)`

Returns an array of string objects containing all of the parameter values or null if parameters do not exist. For example, `String[] value = request.getParameterValues("interests");`

2.4.4 Http Request Headers

The `request-header` fields allow the client to pass additional information about the request, and about the client itself, to the server. Few of the request headers available are as follows:

- `Accept` - It is used to specify types of headers acceptable by the client.
- `Accept-Charset` - The character sets acceptable by the response is specified by this field.
- `Accept-Encoding` - Restriction of content-coding which is accepted by response is mentioned in the field.

- Accept-Language - Restriction of a group of natural languages which is used by response is set by this field.
- Authorization - Authentication of a user agent with a server is done by this field.

2.4.5 Reading Headers form Request

A servlet can access the headers of an HTTP request through the following methods of the `HttpServletRequest` interface:

- `public String getHeader (String name)`
Returns the value of the specified request header as a String. For example, `out.print ("<TD>" + request.getHeader(headerName));`
- `public java.util.Enumeration getHeaders(java.lang.String name)`
Returns all the values of the specified request header as an Enumeration of String objects. For example, `Enumeration headers = request.getHeaders();`
- `public java.util.Enumeration getHeaderNames()`
Returns an enumeration of all the header names this request contains. If the request has no headers, this method returns an empty enumeration. For example, `Enumeration strHeaderName = request.getHeaderNames();`

2.4.6 HTML Forms

HTML forms are used to collect user input data. These input data are stored as form fields and passed through browser URL for further processing in the Servlet.

Several input types are provided by HTML to be used with `<form>` element. Some of them are as follows:

Text input – Allows the user to enter a single line of text.

Drop-down list input – Provides a list of options to the user to be selected.

Submit button – Allows the user to send the data to the server for processing.

The HTML form has the two following major attributes:

- **Methods** – Defines the way of data transfer is done to the Web server. The methods such as `Get`, `POST`, and so on are used to forward the parameters with the HTTP request.
 - **GET** - With the `GET` method, all form data is encoded into the URL, appended to the action URL as query string parameters. The `GET` method is taken as the default if the `method` attribute is not specified in the HTML form.
 - **POST** - With the `POST` method, form data appears within the message body of the HTTP request.

- **Action** - Includes the servlet class name. For example, <form method="POST" action="Login">

Code Snippet 4 shows the HTML page for accepting the user choice.

Code Snippet 4:

```
<form method="GET" action="Login">  
    <Username:> <input type="text" name="username"/>  
    <br/>  
  
    <select name="color" size="1">  
        <option> light  
        <option> amber  
        <option> brown  
        <option> dark  
    </select>  
  
    <center>  
        <input type="SUBMIT"/>  
    </center>  
</form>
```

2.4.7 Handling Form Data Using Request Object

In Servlet, `doGET()` and `doPOST()` are the methods that handle form data. The request object allows the servlet to obtain the user data from the form. To handle the form data, call the data reading methods of the request object.

Code Snippet 5 shows the use of `getParameter()` method in handling data.

Code Snippet 5:

```
public void doGet(HttpServletRequest request, HttpServletResponse
response)
throws IOException, ServletException
{
String name = request.getParameter("username");
String selectedColorValue = request.getParameter("color");

// data processing code here
}
```

The code invokes the `getParameter()` method to gets value of the field `name` and `color` and stores them as `String`.

2.5 Handling Servlet Response

Handling Servlet response defines an object to help a Servlet in sending a response to the client using different interfaces.

2.5.1 'ServletResponse' Interface

Servlet response is the response sent by the servlet to the client. The `ServletResponse` and `HttpServletResponse` interfaces include all the methods needed to create and manipulate a servlet's output.

Figure 2.5 depicts servlet response.

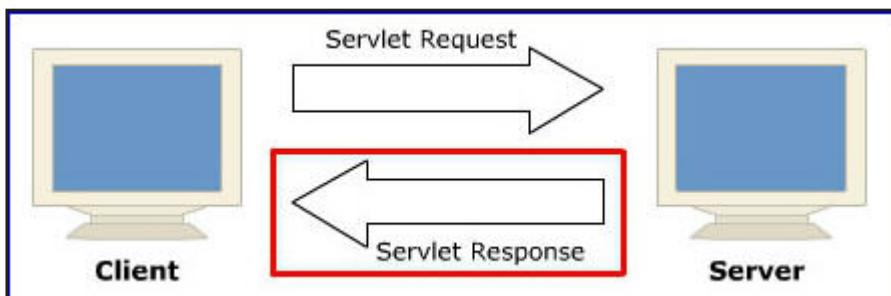


Figure 2.5: Servlet Response

The `ServletResponse` interface defines methods that allows to retrieve an output stream to send data to the client, decide on the content type, and so on. The `ServletResponse` interface defines an object, which is used to provide response to a client.

This object is passed as an argument to `service()` method of a servlet.

The methods defined by the `ServletResponse` interface are as follows:

- `public java.lang.String getContentType()`

Returns the Multipurpose Internet Mail Extensions (MIME) type of the request body or null if the type is not known. For example, `out.println ("Message content type:" + msg.getContentType () + "
");`

- `public PrintWriter getWriter() throws IOException`

Returns an object of `PrintWriter` class that sends character text to the client. For example, `PrintWriter out1= response.getWriter(); out1.println("Some text and HTML");`

- `public ServletOutputStream getOutputStream() throws IOException`

Uses `ServletOutputStream` object to write response as binary data to the client. For example, `ServletOutputStream out2 = response.getOutputStream(); out2.write(twoBytarray);`

- `public void setContentType(java.lang.String str)`

Used to set the format in which the data is sent to the client, either in normal text format or html format. For example, `response.setContentType ("text/html");`

2.5.2 **HttpServletResponse Interface**

The `HttpServletResponse` interface extends `ServletResponse` interface and provides information to an `HttpServlet`. It defines an `HttpServlet` object, which is passed as an argument to the `service()` method of a servlet.

Using the `HttpServletResponse` interface, you can set HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL or add cookies to the response.

Figure 2.6 depicts HTTP servlet response.

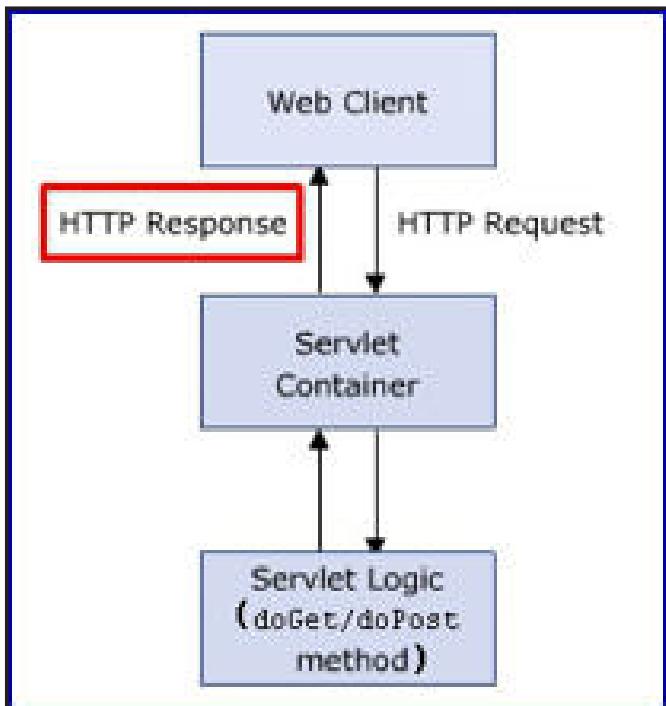


Figure 2.6: HTTP Servlet Response

2.5.3 Methods in `HttpServletResponse` Interface

The methods defined by the `HttpServletResponse` interface are as follows:

- `public void addCookie(Cookie cookie)`

Adds the specified cookie to the response, sent to the client.

Code Snippet 6 shows the use of `addCookie()` method.

Code Snippet 6:

```
// Adds a cookie named cookie2 with color red to the existing  
cookies  
Cookie cookie2 = new Cookie("color", "red");  
response.addCookie(cookie2);
```

- `public void addHeader(java.lang.String name, java.lang.String value)`

Used to add name and value to the response header. For example, `response.addHeader("Refresh", "15")`;

- `public boolean containsHeader(String name)`
Used to verify if the response header contains any values. It returns true if the response header has any values. It returns false otherwise. For example, `res.containsHeader("Cache");`
- `public void sendError(int sc) throws java.io.IOException`
Sends an error response to the client using the specified status code and clearing the buffer. Status codes are used to indicate the reason, a request is not satisfied or that a request has been redirected. For example, `httpResp.sendError(HttpServletRequest.SC_FORBIDDEN, "Good Bye for now!");`

2.5.4 Response Headers

The response header is attached to the files being sent back to the client. It contains the date, size, and type of file that the server sends back to the client and also data about the server itself.

Response headers can be used to specify cookies to supply the modification date. It is also used to instruct the browser to reload the page. In addition, it specifies how big the file is, to determine how long the HTTP connection needs to be maintained.

2.5.5 Sending Headers

The methods of `HttpServletResponse` interface that are used to send header information to the client are as follows:

- `addHeader()` - Adds a response header with the given name and value. For example, `response.addHeader("Cache", 3.14);`
- `addDateHeader()` - Adds a response header with the given name and date value. For example, `res.addDateHeader("Cache", 20-02-2002)`
- `addIntHeader()` - Adds a response header with the given name and integer-value. For example, `res.addIntHeader("Cache", 3)`
- `containsHeader()` - Returns a boolean value to indicate if response header has already been set.

2.5.6 Sending Data from Servlet Using Response Object

Let us create a servlet to send the jar file data using `response` object to the client.

The `Response` object should perform the following steps:

- Inform the browser, the type of file/data that is getting transferred.
- Convert the object data to stream.

Code Snippet 7 shows how to send data using response object.

Code Snippet 7:

```
public class CodeReturnJARfile extends HttpServlet {  
  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException {  
  
        response.setContentType("application/jar");  
        ServletContext ctx = getServletContext();  
  
        InputStream is = ctx.getResourceAsStream("/JAVAcompactV3.jar");  
  
        int read = 0;  
        byte[] bytes = new byte[1024];  
  
        OutputStream os = response.getOutputStream();  
        while ((read = is.read(bytes)) != -1) {  
            os.write(bytes, 0, read);  
        }  
  
        os.flush();  
        os.close(); } }
```

In the given code snippet, `response.setContentType("application/jar");` tells the browser to recognize that the response object carries a JAR. The statement, `OutputStream os = response.getOutputStream();` reads the JAR bytes and writes the bytes to the output stream that is received from the response object.

2.6

Reading Binary Data

Some Web applications such as e-mail need to send files such as documents and images, from one system to another as attachment. In this case, data transmitted from a client to a Web application will be of binary type. In such cases, the user will specify the reference to the binary files in online form.

Multipart/form-data is an Internet media type that returns a set of values as the result of a user filling out a form. These set of values are transmitted as binary data. Each request handled by a servlet has an input stream associated with it.

You use `getInputStream()` to retrieve the input stream as a `ServletInputStream` object. The syntax of the method `getInputStream()` is as follows:

Syntax:

```
public ServletInputStream ServletRequest.getInputStream() throws  
IOException
```

2.6.1 Producing Text and Binary Data

Two methods belonging to `ServletResponse` interface for producing output streams are as follows:

```
public ServletOutputStream getOutputStream() throws java.  
io.IOException
```

The `getOutputStream()` returns a `ServletOutputStream`, which can be used for binary data.

```
public java.io.PrintWriter getWriter() throws java.io.IOException
```

The `getWriter()` returns a `java.io.PrintWriter` object, which is used only for textual output. The `getWriter()` method examines the content-type to determine what character set to use, so `setContentType()` should be called, before `getWriter()`.

2.6.2 Sending Text and Binary Data

Whenever any file sent to the client has to be encoded, for example, images or text attachments, it is sent as binary data. `ServletOutputStream` provides an output stream for sending binary data to the client.

The methods supported by the `ServletOutputStream` class are as follows:

- `public void print(boolean b) throws java.io.IOException`

Writes a boolean value to the client with no Carriage Return-line Feed (CRLF) character at the end.

```
public void println(char c) throws java.io.IOException
```

Writes a character value to the client, followed by a Carriage Return-line Feed (CRLF).

2.7

Redirecting Requests

Whenever the client makes a request, the request goes to the servlet container. The Servlet container decides whether the concerned servlet can handle the request or not. If the servlet cannot handle the request, it decides that the request can be handled by another servlet or JSP.

Then, the servlet calls the `sendRedirect()` method and sends the response to the browser along with the status code. The browser makes a new request, with the name of the servlet that can now handle the request and displays the result on the browser. This redirecting is done by the following methods:

- `public void sendRedirect(java.lang.String location) throws java.io.IOException`
Sends a redirect response to the client using the specified redirect location URL. For example, `response.sendRedirect(response.encodeRedirectURL(contextPath + "/maps"));` redirects the request to the specified location.
- `public java.lang.String encodeRedirectURL(java.lang.String url)`
Encodes the specified URL for use in the `sendRedirect()` method or, if encoding is not needed, returns the URL unchanged. All URLs sent with the `sendRedirect()` method should be run encoded using this method.

2.7.1

Request Dispatcher

The servlets sometimes need to access network resources to satisfy client requests. The resources can be another servlet, a JSP page, or a CGI script. A servlet runs in an environment called the servlet context, which describes various parameters associated with the servlet. A servlet belongs to only one servlet context.

A servlet cannot access resources such as static HTML pages, which are stored in local files using the `RequestDispatcher` objects. The local resource can be accessed using the method `getResource(String path)` of `ServletContext` object that returns a `URL` object for a resource specified by a local Uniform Resource Identifier (URI), for example, '/html'.

The method `ServletRequest.getRequestDispatcher(java.lang.String)` allows using relative path whereas the method `ServletContext.getRequestDispatcher(java.lang.String)` allows only absolute path.

The servlets belonging to same application can share data using the following methods of RequestDispatcher interface are as follows:

□ **forward() method**

The method is used to forward request from one servlet to another resource on the same server.

Syntax:

```
public void forward(ServletRequest request1, ServletResponse  
response1) throws ServletException, IOException
```

where,

- request1 is the request made by the client to the servlet
- response1 is the response made by the servlet to the client

Code Snippet 8 retrieves the servlet context, from its RequestDispatcher instance, and then forwards control to the JSP page.

Code Snippet 8:

```
public class MyServlet extends HttpServlet  
{  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
  
        request.setAttribute("title", "Home Page");  
        ServletContext servletContext = getServletContext();  
        RequestDispatcher dispatcher = servletContext.  
            getRequestDispatcher("/cart.jsp");  
        dispatcher.forward(request, response);  
    }  
}
```

include() method

The method is used to include the contents of another servlet, JSP page or an HTML file.

Syntax:

```
public void include(ServletRequest request1, ServletResponse  
response1) throws ServletException, IOException void  
println(char c) throws java.io.IOException
```

where,

- `request1` is the object that contains client's request.
- `response1` is the object that contains servlet's response.

Code Snippet 9 demonstrates the code that includes the content from the specified path.

Code Snippet 9:

```
// Include static page or servlet/JSP page from the specified path,  
...  
RequestDispatcher dispatcher = getServletContext().  
getRequestDispatcher("/contactus.jsp");  
  
if (dispatcher == null)  
    out.println(path + " not found");  
else  
    dispatcher.include(request, response);  
...
```

2.8**Initializing Servlets**

In a Web application, the database connection information, such as SQL username and password are not sent to the servlet every time a client makes a request. Hence, this information needs to be passed to the servlet before beginning the servlet life cycle.

To pass the parameters from the client-side to the servlets for executing the first time and retrieve the data required as specified by the user the servlet needs to be initialized. To set up an uninterrupted client-server communication at the beginning of a process initialization is done. After setting up the first connection, the process is carried out by auto refreshing the page, which is managed by the servlet itself.

2.8.1 init Parameters

To pass as an argument during initialization, the servlet container uses an object of `ServletConfig` interface. The `ServletConfig` interface provides various methods to configure a servlet, before processing the data requested by the client.

The Servlet has the direct access to Servlet initialization parameters using the `getInitParameter()` method.

The syntax of using the `getInitParameter()` method is as follows:

```
public String getInitParameter(String name)
```

This method retrieves the value of the initialization parameter. This method returns null if the specified parameter does not exist.

Code Snippet 10 demonstrates the code that retrieves the parameters of the servlet.

Code Snippet 10:

```
//Retrieves the parameters user and password  
super.init(config);  
String user1 = getInitParameter("user");  
String password1 = getInitParameter("password");  
...
```

The same method is available in the `ServletConfig` interface to read the initialization parameters.

Figure 2.7 depicts servlet initialization.

```

18  public class InitServlet extends HttpServlet {
19
20      protected void processRequest(HttpServletRequest request,
21                                     HttpServletResponse response)
22          throws ServletException, IOException {
23      response.setContentType("text/html;charset=UTF-8");
24      PrintWriter out = response.getWriter();
25      /* TODO output your page here */
26      out.println("<html>");
27      out.println("<head>");
28      out.println("<title>Servlet InitServlet</title>");
29      out.println("</head>");
30      out.println("<body>");
31      out.println("<h1>Servlet InitServlet at " + request.getContextPath()
32                  () + "</h1>");
33      String initParam=getServletConfig().getInitParameter("MyParam");
34      out.println("Value of the init parameter is :" + initParam);
35      out.println("</body>");
36      out.println("</html>");
37      out.close();
38  }

```

Figure 2.7: Servlet Initialization Using ServletConfig

2.8.2 Reading Parameters in init() Method

□ `public void init() throws ServletException`

Code Snippet 11 illustrates how a servlet reads initialization parameters.

Code Snippet 11:

```

int count;
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    String initial=config.getInitParameter("initial");
    try {
        count=Integer.parseInt(initial);
    } catch (NumberFormatException e) {
        count=0;
    }
}

```

2.8.3 ServletContext

The container first locates the servlet class, loads the servlet and creates an instance of the servlet. It then invokes `init()` method to initialize the servlet. A `ServletConfig` interface object is passed as an argument to the `init()` method, which provides the servlet with access to `ServletContext` interface.

2.8.4 'ServletContext' Interface

The servlet context provides access to various resources and facilities, which are common to all servlets in the application. The servlet API is used to set the information common to all servlets in an application.

Servlets running in the same server sometimes share resources, such as JSP pages, files, and other servlets. This is required when several servlets are bound together to form a single application, such as a chat application, which creates single chat room for all users.

The servlet API also knows about resources to the servlets in the context. The attributes of `ServletContext` class can be used to share information among a group of servlets.

2.8.5 'ServletContext' Methods

The `ServletContext` methods are used to get and set additional information about the servlet.

Following are some commonly used methods of `ServletContext` interface:

- `public String getInitParameter(String name)`: Returns the parameter value for the specified parameter name.
- `public void setAttribute(String name, Object object)`: Sets the given object in the application scope.
- `public Object getAttribute(String name)`: Returns the attribute for the specified name.
- `public Enumeration getInitParameterNames()`: Returns the names of the context's initialization parameters as an Enumeration of String objects.
- `public void removeAttribute(String name)`: Removes the attribute with the given name from the servlet context.

Code Snippet 12 shows the web.xml file for specifying the context parameters and servlet initialization parameters.

Code Snippet 12:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"  
 . . .  
<context-param>  
   <param-name>Global</param-name>  
   <param-value>ValueToAll</param-value>  
</context-param>  
  
<!-- other stuff including servlet declarations -->  
<servlet>  
  <servlet-name>ParamServletTests</servlet-name>  
  <servlet-class>TestInitParams</servlet-class>  
  
<init-param>  
  <param-name>ServletSpecific</param-name>  
  <param-value>ServletValue</param-value>  
</init-param>  
</servlet>  
</web-app>
```

Code Snippet 13 demonstrates the code to read the context parameters from the `web.xml` file.

Code Snippet 13:

```
 . . .
public class ContextParameters extends HttpServlet{
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();

    // Creating ServletContext object
    ServletContext context = getServletContext();

    //Getting the value of the initialization parameter and printing it
    String str = context.getInitParameter("Global");
    pw.println("driver name is=" + str);
    pw.close();
}
}
```

2.9 Error Handling in Servlets

There are many situations where a Web server can give an error. A requested page may be moved from one location to another. The address may be wrongly typed. The requested page may be forbidden, may be temporarily deleted or correct HTTP version might not have found. There are other situations where an error may be generated.

2.9.1 Status Codes

Errors may occur due to one reason or the other but the status code for each type of error, makes it easy for identifying the error.

Table 2.1 lists some of the error codes along with their associated message and meaning.

Status Code	Associated Message	Meaning
301	Moved Permanently	Document is moved to a separate location as mentioned in the URL. The page is redirected to the mentioned URL, to find the document.
302	Found	Temporary replacement of file from one location to the other as specified.
400	Bad Request	The request placed is syntactically incorrect.
401	Unauthorized	Authorization is not given to access a password protected page.
404	Not Found	Resource not found in the specified address.
408	Request Timeout	Time taken by client is very long to send the request (only available in HTTP 1.1).
500	Internal Server Error	Server is unable to locate the requested file. The servlet has been deleted or crashed or has been moved to a new location without informing.

Table 2.1: Status Codes

Figure 2.8 depicts an example of status code.



Figure 2.8: Status Code 404

2.9.2 Error Methods in 'HttpServlet' Class

Errors during the execution of a Web application are reported using the following methods:

□ **sendError()**

This method checks for the status code and sends to the user the specified response message. After sending the error message the buffer is cleared.

Syntax:

```
public void sendError(int sc1) throws java.io.IOException
```

where,

- sc1 is a status code passed as an integer value to the method.

The `sendError()` method sends a predefined error message as a response for the file not found error.

Code Snippet 14 shows the use of `sendError()` method.

Code Snippet 14:

```
// Return the file file1
try {
    ServletUtils.returnFile(file1, out);
}
catch (FileNotFoundException err) {
    res.sendError(res.SC_NOT_FOUND);
}
```

□ **setStatus()**

This code is specified earlier so that on receiving the `setStatus()` method, the error message is thrown or redirected to another default Web page.

Syntax:

```
public void HttpServletResponse.setStatus(int sc1)
```

where,

- sc1 is the status code.

Code Snippet 15 shows the use of `setStatus()` method.

Code Snippet 15:

```
/* If the client sent an If-Modified-Since header equal or after  
the servlet's last modified time, send a short "Not Modified" status  
code Round down to the nearest second since client headers are in  
seconds */  
  
if (servletLastMod == -1) {  
    super.service(req, res);  
}  
else if ((servletLastMod / 1000 * 1000) <=  
    req.getDateHeader("If-Modified-Since")) {  
    res.setStatus(res.SC_NOT_MODIFIED);  
}
```

2.9.3 Logging Errors

Servlets can store the actions and errors through the `log()` method of the `GenericServlet` class. The message along with the stack's trace is thrown to a servlet log.

The `log()` method also assists in debugging by describing all the details about an error. The error can be viewed by using `log()` to record in a server.

❑ log()

The `log()` method can be used by passing either a single argument or two arguments as given here:

Syntax:

```
public void log(String msg1)
```

where,

- `msg1` is the specified message to be written to the servlet log file.

Code Snippet 16 shows the use of `log()` methods.

Code Snippet 16:

```
// Return the file
try {
    ServletUtils.returnFile(file, out);
} catch (FileNotFoundException e) {
    log("Could not find file: " + e.getMessage());
    res.sendError(res.SC_NOT_FOUND);
}
catch (IOException e) {
    log("Problem sending file", e);
    res.sendError(res.SC_INTERNAL_SERVER_ERROR);
}
```

The method writes an explanatory error message and a stack trace for specified throwable object into the servlet log file.

Figure 2.9 depicts a log file.

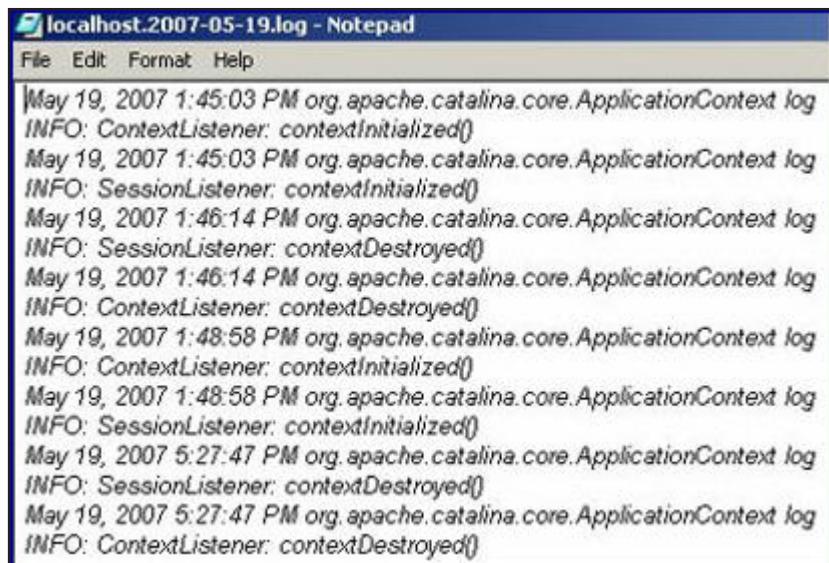


Figure 2.9: Log File

2.9.4 Forwarding to Error Page

A `RequestDispatcher` object is used to redirect the client request to a Web page on receiving an error message on the server. The `RequestDispatcher` object is created by the servlet container.

Following methods creates object for the `RequestDispatcher` interface:

- `'ServletContext.getRequestDispatcher(String path)'`

This method takes a string argument as a path, which is within the scope of the `ServletContext`. The path, relative to root is passed as an argument. This path, in turn, is used to locate the servlet to which it is redirected.

- `'ServletContext.getNamedDispatcher(String name)'`

The method takes a string argument indicating the name of servlet to the `ServletContext`. It returns a `RequestDispatcher` object for the named servlet.

- `'ServletRequest.getRequestDispatcher(String path)'`

This method uses relative path which is the location of the file relative to current path. This is similar to the method in the same name as in the `ServletContext`. Transforming the current servlet path relative to a complete path.

Code Snippet 17 shows the dispatching of error object.

Code Snippet 17:

```
public class Dispatcher extends HttpServlet {  
    public void doGet(HttpServletRequest req1,  
                      HttpServletResponse res1) {  
        RequestDispatcher dispatcher1 =  
            request.getRequestDispatcher("/error_page.  
            jsp");  
        if (dispatcher1 != null)  
            dispatcher1.forward(req1, res1);  
    }  
}
```

The `Dispatcher` class requests for '`error_page.jsp`' file and wait for the response. If the `dispatcher1` is not null, then the file is forwarded as response to the request received from the `dispatcher` class.

Check Your Progress

1. Identify the methods present in life cycle of a servlet.

(A)	service()
(B)	destroy()
(C)	authorize()
(D)	init()

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

2. Match the methods of the `ServletRequest` interface with their corresponding description.

	Description		Method
(A)	Returns the actual length of the request sent by the client.	(1)	getInputStream()
(B)	Returns the binary data requested by the client and stores it in a <code>ServletInputStream</code> object.	(2)	getAttribute()
(C)	Gives additional information about the servlet.	(3)	getContentLength()
(D)	Used to get the additional information that is sent along with the request information.	(4)	getServerName()
(E)	Returns the name of the server to which the request was sent.	(5)	getParameter()

(A)	A-2, B-3, C-4, D-5, E-1	(C)	A-4, B-1, C-2, D-3, E-5
(B)	A-3, B-1, C-2, D-5, E-4	(D)	A-5, B-4, C-2, D-1, E-3

Check Your Progress

3. Which of the following request-header fields allow the client to pass additional information about the request?

(A)	Accept-Numeric
(B)	Accept-Encoding
(C)	Accept-Language
(D)	Accept-Charset
(E)	Accept

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

4. Match the methods of the `ServletResponse` interface with their corresponding description.

Description		Method	
(A)	Returns an object of <code>PrintWriter</code> class that sends character text to the client.	(1)	<code>setContentType()</code>
(B)	Uses <code>ServletOutputStream</code> object to write response as binary data to the client.	(2)	<code>getContentType()</code>
(C)	Returns the MIME type of the request.	(3)	<code>getWriter()</code>
(D)	Used to set the format in which the data is sent to the client.	(4)	<code>getOutputStream()</code>

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

Check Your Progress

5. Match the methods used to send text and binary data to a client with their corresponding description.

Function		Method	
(A)	Writes a boolean value to the client	(1)	getWriter()
(B)	Writes a character value to the client	(2)	getOutputStream()
(C)	Method used to send binary data	(3)	print(boolean b)
(D)	Method used for textual output only	(4)	println(char c)

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

6. Which of the following methods is used for redirecting requests?

(A)	getHeader()
(B)	redirectRequest()
(C)	sendRedirect()
(D)	RequestDispatcher()

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

Check Your Progress

7. Which of the following error code is associated with the message 'Resource Not Found'?

(A)	301
(B)	302
(C)	404
(D)	408

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

8. Match the following methods with their corresponding functions.

Function		Method	
(A)	The method returns a RequestDispatcher object for the named servlet.	(1)	ServletRequest. getRequestDispatcher (String path)
(B)	The method transforms the current servlet path relative to a complete path.	(2)	Servlet. getRequestDispatcher (String path)
(C)	The method assists in debugging by describing all the details about an error.	(3)	ServletContext. getNamedDispatcher (String name)
(D)	The path passed into the method is used to locate the servlet to which it is redirected.	(4)	log()

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

Answer

1.	B
2.	B
3.	C
4.	B
5.	C
6.	D
7.	A
8.	B

Summary

- The init(), service(), and destroy() methods are the servlet's lifecycle methods. The init() method intimates the servlet that it is being placed into service.
- The service() method is called by the servlet container to allow the servlet to respond to a request. The servlet container calls the destroy() method after all threads within the servlet's service method have exited or after a timeout period has passed.
- A GenericServlet class defines a servlet that is not protocol dependent. To have better control over the required servlets HttpServlet is extended to GenericServlet.
- A servlet request contains the data to be passed between a client and the servlet. All requests implement the ServletRequest interface, which defines the methods for accessing the relevant information.
- A servlet response contains data to be passed between a server and the client. All responses implement the ServletResponse interface. This interface defines various methods to process response.
- You use getInputStream() to retrieve the input stream as a ServletInputStream object. ServletOutputStream provides an output stream for sending binary data to the client.
- Resources are included to a servlet by forwarding request from one servlet to another by using the forward() and include() methods along with RequestDispatcher interface. Inter-servlet communication can be used by the servlets to gain access to other currently loaded servlets and perform some tasks on other servlets.
- Errors may occur due to one reason or the other but the status code for each type of error, makes it easy for identifying the error. Errors are reported using sendError() and setStatus() methods. Errors are logged using the log() method of ServletContext and forwarded to an error page using the RequestDispatcher interface.



Login to www.onlinevarsity.com



Welcome to the Session, **Session Tracking**.

This session explains the concept of session tracking that allows the server to keep a track of successive requests made by the same client. The session explains various session tracking techniques, such as URL rewriting, hidden fields, cookies, and HttpSession that are used to maintain the client's information.

In this Session, you will learn to:

- Describe the stateless nature of HTTP protocol
- Explain the need of tracking client identity and state
- Explain the URL rewriting method for session tracking
- Explain how to use hidden form fields
- Explain the use of Cookie class and its methods
- Explain how to store and retrieve information in a session
- Describe the use of HttpSession interface and its methods
- Explain how to invalidate a session

3.1 Introduction

A protocol is a set of rules, which governs the syntax, semantics, and synchronization of communication.

When the configuration setting, transaction, and information are not tracked by a protocol, then it is called a stateless protocol. For such a protocol connections last for only one transaction.

The HTTP protocol is a stateless protocol that is based on a client-server model. An HTTP client, such as a Web browser, opens a connection and sends a request message to an HTTP server asking for a resource. The server returns a response message with the requested resource. Once the resource is delivered, the server closes the connection. Thus, no connection information is stored, and hence HTTP is referred to as a stateless protocol.

The stateless nature of HTTP protocol has its own advantages and disadvantages.

Advantages

The advantage of a stateless protocol is that hosts do not need to retain information about users between requests. This simplifies the server design because it does not need to dynamically allocate storage to manage conversations in progress or worry about freeing it when a client connection dies in mid-transaction.

Disadvantages

A disadvantage of HTTP being stateless is that it may be necessary to include more information in each request and this extra information will need to be interpreted by the server each time. Another disadvantage is that the acknowledgement that the information has reached the client is absent. Hence, the loss of data, if any, is not known.

Figure 3.1 depicts client-server model using HTTP protocol.

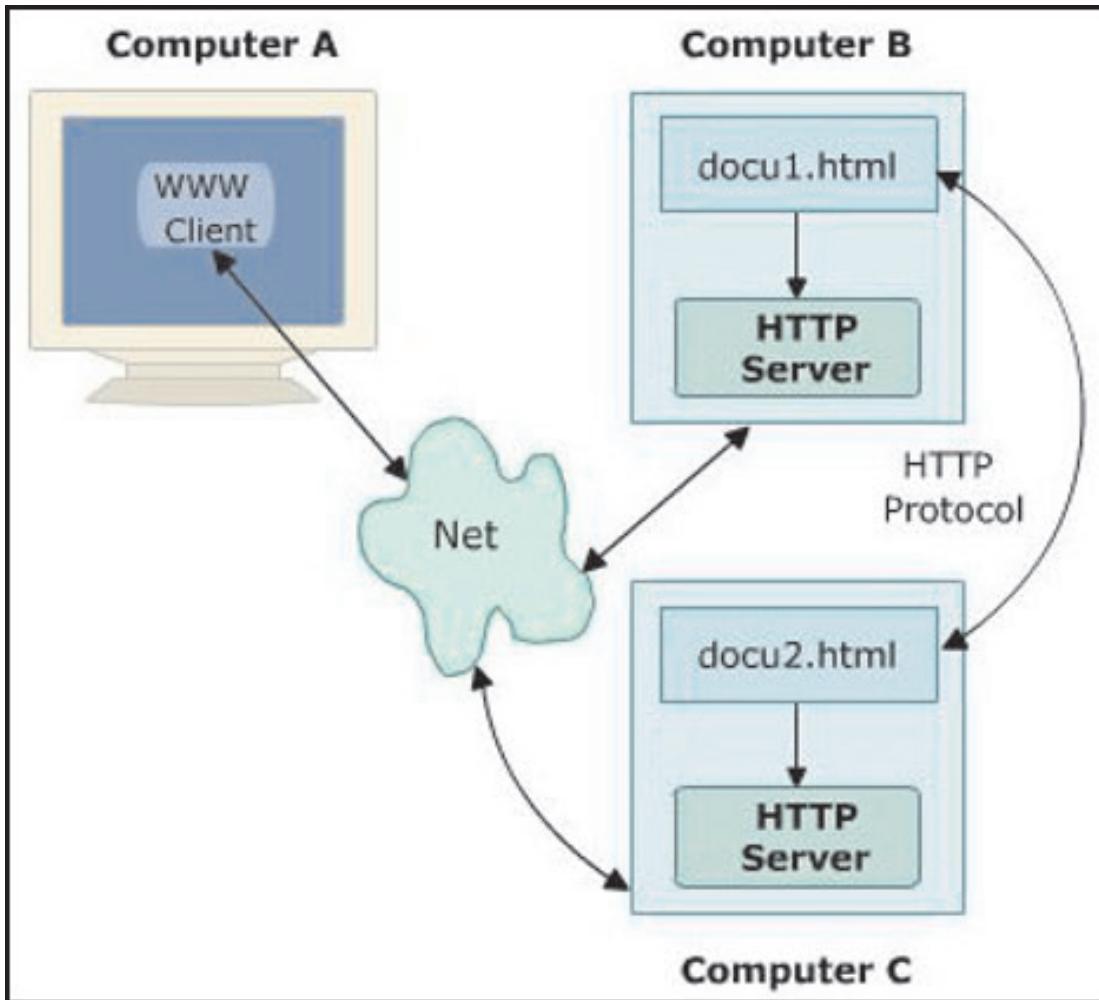


Figure 3.1: Client-Server Model Using HTTP Protocol

3.1.1 Session Tracking

Consider a scenario of an online shopping Web site where items are displayed on different pages based on classifications, such as stationery, hardware, music, books, and so on. When a customer is doing online shopping, he/she may select items from various pages and put it in the shopping cart. However, due to HTTP being a stateless protocol, when the customer clicks a new page, the information about the previously selected items is lost. As a result, the server may not keep track of the selected items on the pages.

To resolve this problem, it is necessary for Web applications to adopt a mechanism that can be used to keep a track of successive requests made by the same user.

The tracking mechanism allows the Web application to maintain information with the server as long as the customer does not log out from the Web site. This tracking mechanism is known as session tracking.

The session tracking mechanism serves the purpose of tracking the client identity and other state information required throughout the session.

Figure 3.2 depicts session tracking.

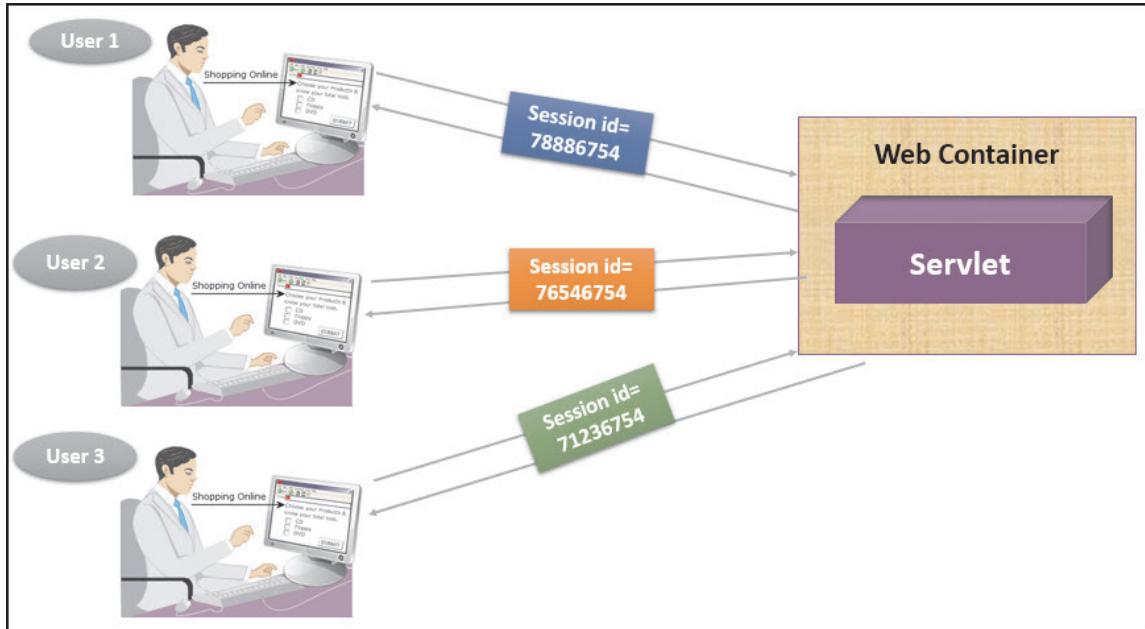


Figure 3.2: Session Tracking

3.1.2 Session Tracking Techniques

There are various techniques that can be used by a Web application to maintain a session which keeps track of data between the multiple requests from a client.

Some of these techniques are as follows:

- **URL rewriting** – This technique is used to pass data from one page to another by appending a text string at the end of the request URL.
- **Hidden field** – This technique is used to pass values from HTML form hidden fields to Web resources.
- **Cookie** – This technique is used to store the information on the client's machine in a text file.

These are the most common techniques used by Web applications for maintaining clients' data. Apart from these, Java Servlet API specification provides a session tracking mechanism through `javax.servlet.http.HttpSession` object. The object is used by Servlet to store or retrieve information related to the user for maintaining session on the server.

3.2 URL Rewriting

Uniform Resource Locator (URL) is the address of a resource located on the Web. The URL rewriting technique adds some extra data at the end of the URL to identify the session.

The extra information can be in the form of extra path information or added parameters. The user does not see extra information on the surface as such but when he/she clicks a link, the data from the page is appended after the '?' in the URL.

Figure 3.3 shows the URL rewriting technique used for session management between the pages.

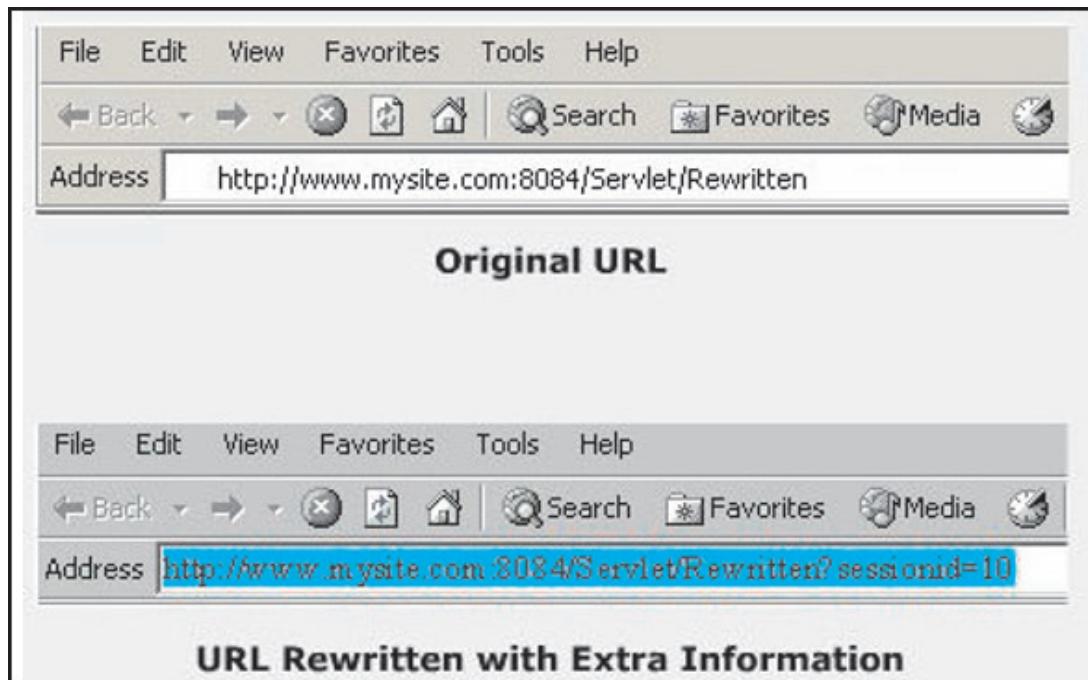


Figure 3.3: URL Rewriting

As shown in figure 3.3, the original URL,

`http://www.mysite.com:8084/Servlet/Rewritten` is appended with the parameter `sessionid=10` at the end of the URL. The `sessionid` parameter is sent to the server as part of the client's request and helps the server to identify the client.

The URL rewriting is the lowest priority technique used for session management and is used as an alternative for cookies. In case, if the setting to receive cookies at the client-side is disabled in the Web browser, the URL rewriting is used.

3.2.1 Information in URL

A parameter or token is attached at the end of the query string in the request sent from the Web browser. The token consist of name-value pair.

Figure 3.4 shows the token in the URL.



Figure 3.4: URL Rewriting Technique

Code Snippet 1 demonstrates the working of URL rewriting technique using two Servlets.

Code Snippet 1:

```
/* Servlet1.java */
. . .
public class Servlet1 extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        String sessionID = "Session1";
        PrintWriter pw = response.getWriter();
        pw.println("<html>");
        pw.println("<head></head>");
        pw.println("<body>");
        pw.println("Please click the below link:<br>");
        pw.println("<a href=/MyWebApp/Servlet2?sessionID=" + sessionID +
        ">View Report</a><br>");
        pw.println("</body>");
        pw.println("</html>");
    }
}
```

```
/* Servlet2.java */
...
public class Servlet2 extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html; charset=UTF-8");
        String sessionID = request.getParameter("sessionID");
        PrintWriter pw = response.getWriter();
        pw.println("<html>");
        pw.println("<head></head>");
        pw.println("<body>");
        pw.println("This is the Session ID of the last page: <br>");
        pw.println("SessionID=" + sessionID + "<br>");
        pw.println("</body>");
        pw.println("</html>");
    }
}
```

In the code, the **Servlet1** generates a hyperlink element which when clicked by the user will send the request to **Servlet2**. A name-value parameter **sessionId=Session1** is appended with the URL link specified with the **<a>** element. The **Servlet2** access the parameter using **getParameter()** method and displays its value on the page.

Figure 3.5 shows the output of Code Snippet 1.

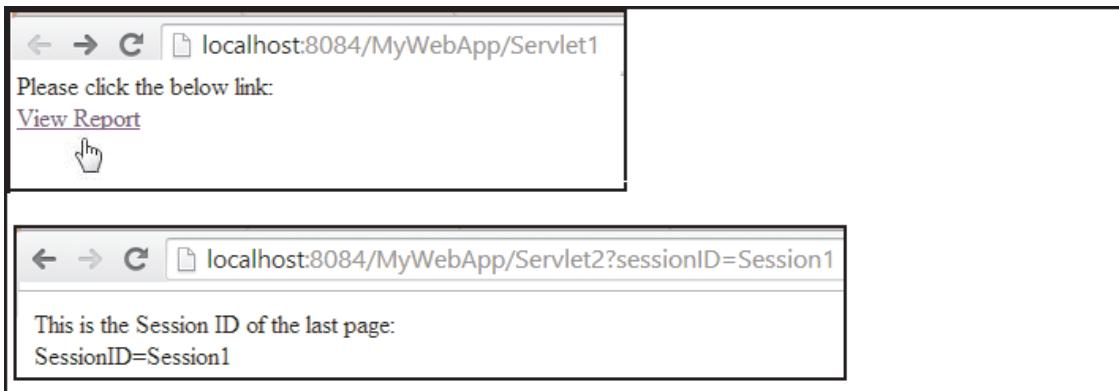


Figure 3.5: Output – URL Rewriting

3.2.2 Advantages of URL Rewriting

- URL can be appended when sending the data from the form as well as it can be sent along with the dynamic generated content from a Servlet.
- URL is a preferred way to maintain the session when the browser doesn't support cookies or user disables the support for cookies.

3.2.3 Disadvantages of URL Rewriting

- The only way to send the URL is through hyperlinks on the Web page.
- Sometimes, the URLs are very long and cannot store that much information because of the URLs length limitation.
- URLs containing data information is visible, so it is not safe to be shared with others.

3.3 Hidden Form Fields

This technique is similar to URL rewriting, instead of appending values to the URL, hidden fields are placed within an HTML form. Hidden form fields are either a part of the static HTML form or dynamic form generated through Servlets. They can be used to hold any kind of data.

The hidden fields are not visible to the user and hence are not interpreted by the browser. Each time the Servlet receives the form data, it would read all the parameters passed to it from the form along with the values in the hidden fields received from the HTML page.

The advantage of hidden field technique over URL rewriting is that you can pass much more information to the server and even character encoding is not necessary.

The syntax for adding the hidden field element in the HTML form is as follows:

Syntax:

```
<INPUT TYPE="HIDDEN" NAME="..." VALUE="...">>
```

Code Snippet 2 demonstrates the use of hidden field in the Web application.

Code Snippet 2:

```
<!-- index.html -->
...
<body>
<form action="MyServlet" method="POST">
<input type="hidden" name="job" value="Developer"/>
Name:<input type="text" name="firstname"/>
<input type="submit" name="Submit"/>
</form>
</body>
...

```

```
/* MyServlet.java */
```

```
...
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html; charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        // Retrieves the name entered on the form
        String name = request.getParameter("firstname");
        // Retrieves the value from the hiddenfield
        String job = request.getParameter("job");
        out.println("<h3>Welcome: " + name + "<br>");
        out.println("\n Your Job: " + job + "</h3>");
        out.close();
    } finally {
        out.close();
    }
}
...

```

Figure 3.6 shows the output of Code Snippet 2.



Figure 3.6: Output - Hidden Form Field

3.3.1 Advantages of Hidden Form Fields

- Hidden form fields are supported in all browsers.
- Hidden form fields have no special server requirements from clients.
- Hidden form fields are not visible directly to the user, but can be viewed using **View page source** option from the browsers.
- Hidden form fields works with or without cookies, even if the cookies are disabled, they will not have any impact.

3.3.2 Disadvantages of Hidden Form Fields

- Hidden form fields works only when the page receives request through a submission of a form. This means if the target page receives request by clicking a hyperlink, it will not result in form submission.

3.4 Cookies

A cookie is a small piece of information sent by a server to the client Web browser. The cookies are stored on client's machine and are read back by the server on receiving request for the same page.

A cookie contains one or more name-value pairs which are exchanged in request and response headers. The HTTP request header contains the request made by the client. It contains information such as method, URL path, and HTTP protocol version.

The HTTP response header contains the date, size, and type of the file that server is sending back to the client. When the server sends a cookie, the client receives the cookie, saves and sends it back to the server each time the client accesses a page on that server.

Cookies are stored for a limited life span on the client's machine and once the specified time period is completed, they are automatically deleted.

The major drawback with cookies is that most browsers allow the users to deactivate (not to accept) cookies.

As the value of the cookie can uniquely identify a client, cookies are commonly used in session tracking.

Figure 3.7 depicts the concept of cookie.

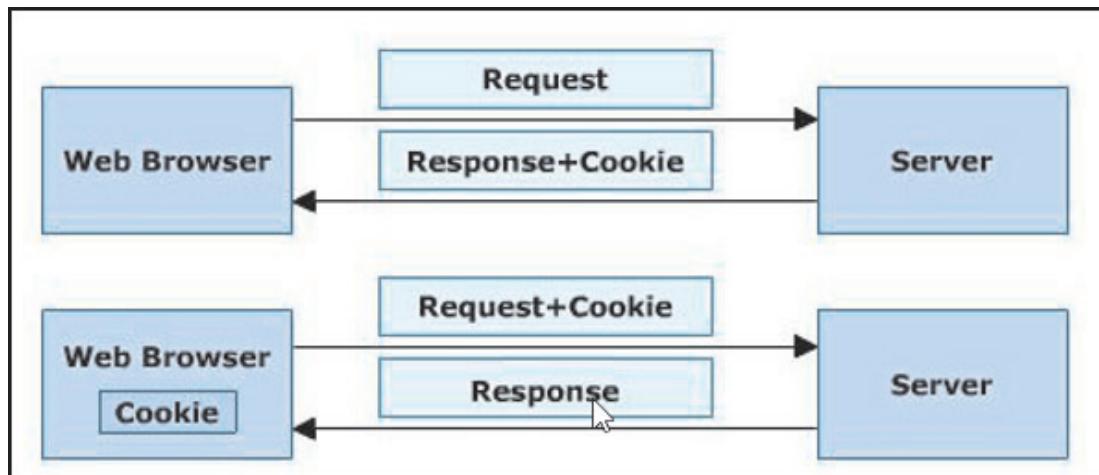


Figure 3.7: Concept of Cookie

Code Snippet 3 demonstrates how to create and add cookie in the Servlet response.

Code Snippet 3:

```
// This snippet remember an added item by adding to a cookie
...
public void doGet (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    // If the user wants to add an item in a cookie
    if (values != null) {
        ItemId=values[0];
        Cookie getitem=new Cookie ("Buy", ItemId);
```

```
getItem.setComment ("User has indicated a desire " +  
    "to buy this book from the bookstore.");  
  
response.addCookie (getItem);  
  
}  
}
```

3.4.1 Cookie API

The Servlet API provides `javax.servlet.http.Cookie` class for creating and working with cookies. These cookies are small amount of information sent by a servlet to the Web browser. The value of cookie is unique and can be used to identify a client thereby helping in session management.

The `Cookie` class provides several methods which help in cookie management. These are as follows:

- **`public void setMaxAge(int expiry)`**

This method sets the maximum age of the cookie in seconds. If the value is positive, then the cookie will expire after that many seconds which is specified by the expiry. For example, `demoCookie.setMaxAge(600);`

- **`public int getMaxAge()`**

This method returns the maximum age of the cookie. This method returns an integer which specifies the maximum age of the cookies in seconds.

Code Snippet 4 demonstrates how to get the cookie age.

Code Snippet 4:

```
/* Prints the cookie age */  
  
PrintWriter out = response.getWriter();  
  
Cookie demoCookie = new Cookie ("FavColor", "Blue");  
  
demoCookie.setMaxAge (600);  
  
int result = demoCookie.getMaxAge ();  
  
out.println ("Cookie Age: " + result);  
  
...
```

The code returns the maximum age of the cookie, which is specified in seconds.

- **public void setValue(java.lang.String newValue)**

This method assigns a new value to a cookie after the cookie is created. In case if binary value is used, base 64 can be used for encoding.

Code Snippet 5 demonstrates how to set the value of the cookie.

Code Snippet 5:

```
// Sets the value of the cookie  
  
public void setCookieValue(String value) {  
    if (value == null || (value.equals("")))  
        throw new IllegalArgumentException("Invalid cookie  
value  
set in: " + getClass().getName());  
    if (cookie != null)  
        cookie.setValue(value);  
}
```

- **public java.lang.String getValue()**

This method returns the value of the cookie. The method returns a string containing the cookie's present value.

- **public java.lang.String getName()**

This method returns the name of the cookie. Once, the cookie has been created its name cannot be changed.

Code Snippet 6 demonstrates how to retrieve the name and value of the cookie.

Code Snippet 6:

```
// Retrieves the name of the cookie and its value  
  
for (int i = 0; i < cookies.length; i++) {  
    String name = cookies[i].getName();  
    String value = cookies[i].getValue();  
    out.println("name = " + name + "; value = " + value);
```

□ `public void setPath(String uri)`

This method sets the path for the cookie. The cookie is available to all the pages specified in the directory and its subdirectories. A cookie's path must have the Servlet which sets the cookie.

Code Snippet 7 demonstrates how to set the path for the cookie.

Code Snippet 7:

```
// Sets the path for the cookie
cookie = new Cookie("sessionId", "erased");
cookie.setPath("/servlet/SessionCookie");
resp.setHeader("Set-Cookie", cookie.toString());
```

□ `public java.lang.String getPath()`

This method returns the path on the server to which the client returns the cookie. The cookie is available to all sub paths on the server. This method returns a string specifying a path that contains a servlet name, for example, /AptechDemo.

□ `public Cookie[] getCookies()`

Cookies can be read from a request by using the `HttpServletRequest.getCookies()` method. The `getCookies()` method returns an array containing all of the `Cookie` objects that the client sends with the request.

Code Snippet 8 demonstrates how to read cookies received in the client request.

Code Snippet 8:

```
// Retrieves cookies from the request object
Cookie[] cookies = request.getCookies();
// Iterates through the array
for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    // Print cookie details
    out.println("Cookie Name: " + cookie.getName());
    out.println("Cookie Value: " + cookie.getValue());
}
```

- **void addCookie (Cookie cookie)**

The cookies are sent by the servlet to the browser by using the `HttpServletResponse.addCookie (javax.servlet.http.Cookie)` method.

This method adds field to the HTTP response headers to send cookies to the browser, one at a time.

This method adds specified cookie to the response. This method can be called multiple times to set more than one cookies. For example, `response.addCookie(new Cookie("cookiename", "cookievalue"));`

3.4.2 Securing Cookies

To secure the cookies from hackers on the Web, you can configure cookies with two security settings namely, `secure` and `HttpOnly`.

The `secure` flag informs the Web browser that cookies should be sent only on the SSL connection. This means any page of the Web application that is not secured will not be able to access cookies.

The JavaScript can also be used to access the cookies from the machine. This means the cookies are available across scripts and may be accessed by hackers for manipulation.

Servlet 3.0 provides an `HttpOnly` flag which informs the browser that the content of the cookie are not accessible within JavaScript. It is included in the HTTP response header and prevent the cookies from certain kinds of attacks.

The cookie class provides the following methods:

- **public void setSecure()**

This method informs the browser to send the cookie only through secured protocol, such as HTTPS.

- **public void setHttpOnly(boolean)**

This method can be used to mark or unmark the cookie. If set as true, then cookie is marked as `HttpOnly` and are not exposed to client-side scripting code.

- **public boolean isHttpOnly()**

This method is used to check whether the cookie has been marked as `HttpOnly`.

Code Snippet 9 shows how to set the cookies as HttpOnly and secure flag programmatically in Servlet.

Code Snippet 9:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {  
    . . .  
    Cookie cookie = new Cookie("Color", "Cyan");  
    Response.addCookie(cookie);  
    cookie.setHttpOnly(true);  
    cookie.setSecure(true);  
    . . .  
    boolean status = cookie.isHttpOnly();  
    out.println("<br>Status of Cookie - Marked as HttpOnly = " +  
    status);  
}
```

Alternatively, you can provide the declaration settings in the `web.xml` configuration file.

Code Snippet 10 shows the `web.xml` file with security setting for a cookie.

Code Snippet 10:

```
    . . .  
<session-config>  
    <cookie-config>  
        <http-only>true</http-only>  
        <secure>true</secure>  
    </cookie-config>  
</session-config>  
    . . .
```

3.5

HttpSession

HttpSession helps in identifying user in a multi-page request scenario and information about that user. The HttpSession interface is used to create a session between the client and server. The session is created between an HTTP client and an HTTP server by the Servlet container using this interface.

When users make a request, the server assigns it a session object and a unique session ID. The session ID matches the user with the session object in subsequent requests. The session ID and the session object are then passed along with the request to the server.

The methods of HttpSession interface used to create a session are:

□ **public Object getAttribute(String name)**

This method returns the object which is bound with the specified name in the session. It returns null in case there is no object bound under the name.

□ **public String getId()**

This method returns a string containing the unique identifier assigned to this session. The servlet container assigns the identifier and it is implementation independent.

□ **public int getMaxInactiveInterval()**

This method returns the maximum time interval, in seconds, for which the Servlet container will keep the session alive between the client accesses. Once, this interval is over session is invalidated by the servlet.

□ **public ServletContext getServletContext()**

This method returns the ServletContext object to which the current session belongs.

□ **public void invalidate()**

This method invalidates the session and the objects bound to the session are unbound. This method throws IllegalStateException if called on already invalidated session.

□ **public boolean isNew()**

This method returns true if the client is unaware about the session or chooses not to be part of the session. For example, if only cookie-based session is used and the client had disabled the use of cookie, then each request would be considered as a session.

□ **public void setAttribute(String name, Object value)**

This method binds the object to the current session by using the specified name. In case of repetition of name of the object bound to the session, the object is replaced.

This method throws IllegalStateException in case of being called on an invalidated session.

□ **public void removeValue(String name)**

This method removes the object bound with the specified name from the session. In case of absence of object bound to specified name, this method does nothing.

This method throws `IllegalStateException` in case of being called on an invalidated session.

- `public void setMaxInactiveInterval(int interval)`

This method specifies the time, in seconds, between the client requests before the servlet container invalidates the current session.

3.5.1 Storing Information in a Session

The data can be stored in an `HttpSession` object using the name-value pairs. The data which is stored is available throughout the current session. To store the data in a session, the method `setAttribute()` is used.

The `setAttribute()` method sets the value of the attribute which can be retrieved later.

Code Snippet 11 demonstrates how to create a new session object and set object in it.

Code Snippet 11:

```
public void doGet(HttpServletRequest request, HttpServletResponse
) throws IOException, ServletException
{
    HttpSession httpSession = request.getSession(true);
    // Gets current session or create a new one if not exist
    if (httpSession isNew())
    {
        // Set the maximum interval for the session
        httpSession.setMaxInactiveInterval(60);
        // Sets the name attribute name as Jami
        httpSession.setAttribute("name", "Jenny");
        // Sets the attribute background color to "#FFFFFF"
        httpSession.setAttribute("age", new Integer(20));
    }
}
```

In the code, `getSession()` returns the current session object associated with the request. If the session does not exists, then the boolean value `true` indicates to creates a new session.

The `isNew()` returns a boolean value indicating whether it is a new session or not. If it returns `true`, then the objects are bounded to the new session through `setAttributes()` method.

The method `setMaxInactiveInternal()` specifies the time between the requests from the client before servlet container invalidates the session.

3.5.2 Retrieving Information Stored in Session

The server, using the `HttpSession`, does session tracking. The previously stored value can be retrieved using `getAttribute()` method. Since, the return type is an object, typecasting of data associated with that attribute name in the session is done.

If the return value is null, it can be concluded that no such attribute exists. Therefore, it is required to check for null before calling the methods on the object.

Code Snippet 12 explains the procedure for retrieving name and age stored in the session.

Code Snippet 12:

```
// Retrieves name and age from session  
String myText = (String) httpSession.getAttribute("name");  
int myNumber = ((Integer) httpSession.getAttribute("age")).  
intValue();
```

3.5.3 Invalidate a Session

Sometimes an unauthorized user may hack into the Web application. To avoid the hacker from causing any harm, the `invalidate()` method is called.

The `invalidate()` method destroys the data in a session that another servlet or JSP might require in future. Therefore, invalidating a session should be done cautiously as sessions are associated with the client, not with individual servlets or JSP pages.

Code Snippet 13 demonstrates invalidating the session.

Code Snippet 13:

```
// Returns current session or a new session if it does not exist  
HttpSession session = request.getSession (true);  
  
// Checks the session  
  
if (session.isNew () == false) {  
  
    // Invalidates the session if it is not a new session  
    session.invalidate ();  
  
    // Creates a new session  
    session = request.getSession (true);  
}
```

The code directs the session to invalidate itself if it is not created newly. To invalidate the session manually, the `invalidate()` method is being called.

3.5.4 Session Timeout

Session timeout is necessary as a session utilizes the memory locations to store information and long period of inactivity will occupy the memory unnecessarily. After a certain time period of inactivity the session is destroyed to prevent the number of sessions increasing infinitely. The data stored in the session disappears. Session timeout happens if the user remains inactive for a period greater than the set inactive time period.

The session timeout period can be set either in the `web.xml` file or can be set by the method `setMaxInactiveInterval()`. This method is used to specify the time between the requests from the client before servlet container invalidates the session.

Syntax:

```
<session-config>  
    <session-timeout>N</session-timeout>  
</session-config>
```

where,

- N in the fragment is session timeout period.

The setting for `session-timeout` should be written in `web.xml` file.

Check Your Progress

1. Which of the following statements describing the stateless nature of HTTP protocol and the need of tracking client identity and state are true?

(A)	When the configuration setting, transactions, and information are not tracked by the protocol, then it is called stateless protocol.
(B)	The HTTP protocol is based on a client-server model.
(C)	The HTTP protocol sends an acknowledgement that the information sent by the server has reached the client.
(D)	Session tracking allows the server to keep a track of successive requests made by the same client.
(E)	In HTTP protocol, the loss of data can be easily detected.

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

2. Which among the following statement relating to URL rewriting is incorrect?

(A)	URL is the address of a resource located on the Web.
(B)	The URL rewriting technique adds some extra data at the end of the URL to identify the session.
(C)	Generally, the extra information appended in the URL rewriting is the unique session ID and tracking can be done by retrieving this session ID.
(D)	Only information after ? is sent to the next page and information before ? is detached.

3. Which among the following methods returns a string containing the cookie's present value?

(A)	getMaxAge ()	(C)	setMaxAge ()
(B)	getValue ()	(D)	setValue ()

4. Which among the following syntax does not belong to methods for reading cookies?

(A)	public java.lang.int getValue ()	(C)	public java.lang. String getName ()
(B)	public Cookie[] getCookies ()	(D)	public java.lang. String getValue ()

Check Your Progress

5. Which among the following methods is correct for sending cookie?

(A)	HttpServletResponse. setCookie(javax. servlet.http.Cookie)	(C)	HttpServletRequest. addCookie(javax. servlet.http.Cookie)
(B)	HttpServletResponse. addCookie(javax. servlet.http.Cookie)	(D)	HttpServletResponse. addCookie(javax. servlet.http. addCookie)

6. Which among the following statements related to retrieving information in a session is incorrect?

(A)	The previously stored value can be retrieved using <code>getAttribute()</code> method.
(B)	While retrieving, the return type is a class.
(C)	Type casting of data associated with the attribute name in the session is possible.
(D)	It is required to check for null before calling the methods on the object.

7. Which among the following statements related to session timeout is incorrect?

(A)	Session is automatically timed out when the maximum number of sessions is reached.
(B)	Session timeout is necessary as it utilizes the memory locations to store information.
(C)	The session timeout period can be set in the <code>web.xml</code> file.
(D)	The session timeout period can be set by the method <code>setMaxInactiveInterval()</code> also.

Answer

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

Summary

- Session tracking allows the server to keep track of successive requests made by the same client.
- Some of the session tracking techniques are namely, URL rewriting, hidden field, and cookie.
- Java Servlet specification provides a session tracking mechanism through javax.servlet.http.HttpSession object.
- The URL rewriting technique adds some extra data at the end of the URL to identify the session.
- Hidden form fields are used to pass data to the server-side resource invisibly from the user.
- A cookie is a small piece of information sent by a server to the client Web browser. The cookies are stored on client's machine and are read back by the server on receiving request for the same page.
- To secure the cookies from hackers on the Web, you can configure cookies with two security settings namely, secure and HttpOnly.
- The HttpSession interface is used to create a session between the client and server. The session is created between an HTTP client and an HTTP server by the servlet container using this interface.



Welcome to the Session, **Filters and Annotations**.

This session introduces the concept of filter as a vital Web component. The session explains how to create filters that can manipulate client requests and server responses. Further, the session provides a comprehensive study of annotations supported by Java Servlets. Finally, the session concludes with the explanation on uploading the files using HTML forms and Servlets.

In this Session, you will learn to:

- Explain the concept and working of filters
- List the benefits of filters
- Explain the Filter API interfaces and there methods
- Explain the use of wrapper classes in managing Servlets
- Explain how to alter a request and response using filters
- Describe the basic concept of annotations
- Explain the different types of annotations supported in Servlet API
- Explain the steps to upload the file using HTML form elements
- Explain the mechanism to upload files on the server using Servlet

4.1 Introduction

There are some situations when data sent or received from the Web resources need to be processed with extra functionality. For example, a common scenario for processing the sensitive data sent in response over the network. The data sent in the response from a Servlet must be encrypted to avoid any malfunction threat. This needs an extra processing of encryption to be done on the received response from the Servlet.

Java supports filters objects that performs processing of the request before it acquires the resource. Similarly, before sending the response, filters perform manipulation of the responses sent to the client.

Some of the common scenarios of the Web applications where filters can be configured are authentication of the users, logging of the messages, image conversion, compression of data, encryption of responses, transformation to XML, and so on.

4.1.1 Concept of Filter

Filters are Java classes and are typically used to encapsulate the preprocessing and post processing functionalities that are common to a group of Servlets or JSP pages.

Filters were introduced as a Web component in Java Servlet specification. They reside in the Web container along with the other Web components, such as Servlet or JSP pages.

Figure 4.1 depicts filters in a Web application.

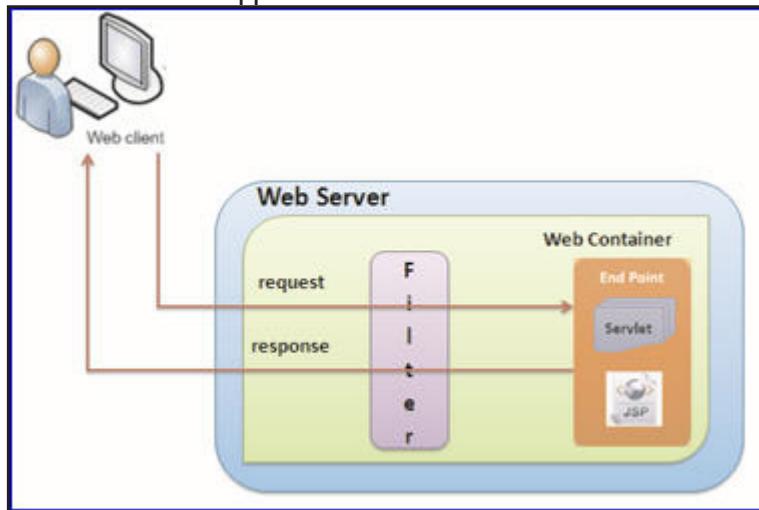


Figure 4.1: Filters

Figure 4.1 shows the filter acting as interceptors for processing the request and response between the client and Web resources or end points, such as Servlet and JSP.

Some of the important features provided by the filters to the Web applications are as follows:

- Optimization of the time taken to send a response back to a client.
- Compression of the content size sent from a Web server to a user over the Internet.

- Conversion of images, texts, and videos to the required format.
- Optimization of the available bandwidth on the network thereby reducing the network traffic and in turn, increasing the efficiency of all Web-based services.
- Authentication of the users in the secured Web site.
- Encryption of the request and response header for addressing security concerns.

4.1.2 Filter Channing

There can be more than one filter between the client and the Web resources, thus forming a filter chain. A request or a response is passed through one filter to the next in the filter chain. So each request and response has to be serviced by filters configured in the chain, before the Servlet is called by the Web container and after the Servlet responds.

Figure 4.2 shows the channing of the filters.

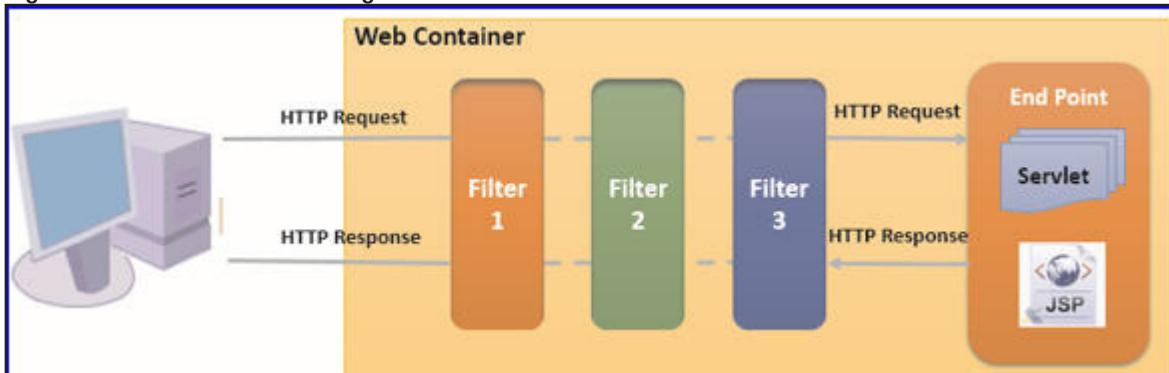


Figure 4.2: Filter Channing

4.1.3 Working of Filters

Filters dynamically access incoming requests from the user before the servlet processes the request. Filters can also access the outgoing response from the Web resources before it reaches the user.

The information about the configured filters is provided when the application is deployed on the server through web.xml.

A typical filter operates in the following way:

1. When the request for the Web resource is sent to the Web server, the Web container instantiates the filters and loads them for preprocessing.
2. The filter intercepts the request from the user to the servlet.
3. After the request is processed, the filter can do the following:
 - a. Generate response and return to the client.

- b. Pass the modified or unmodified request to other filter in the chain.
 - c. If the request is received by the last filter in the chain, it passes the request to the destination Servlet.
 - d. It may route the request to a different resource rather than the associated Servlet.
4. Then, the filter sends the serviced request to the appropriate Servlet.

Similarly, when the response is returned to the client, the response passes through same set of filters in reverse order.

Figure 4.3 depicts working of a typical filter.

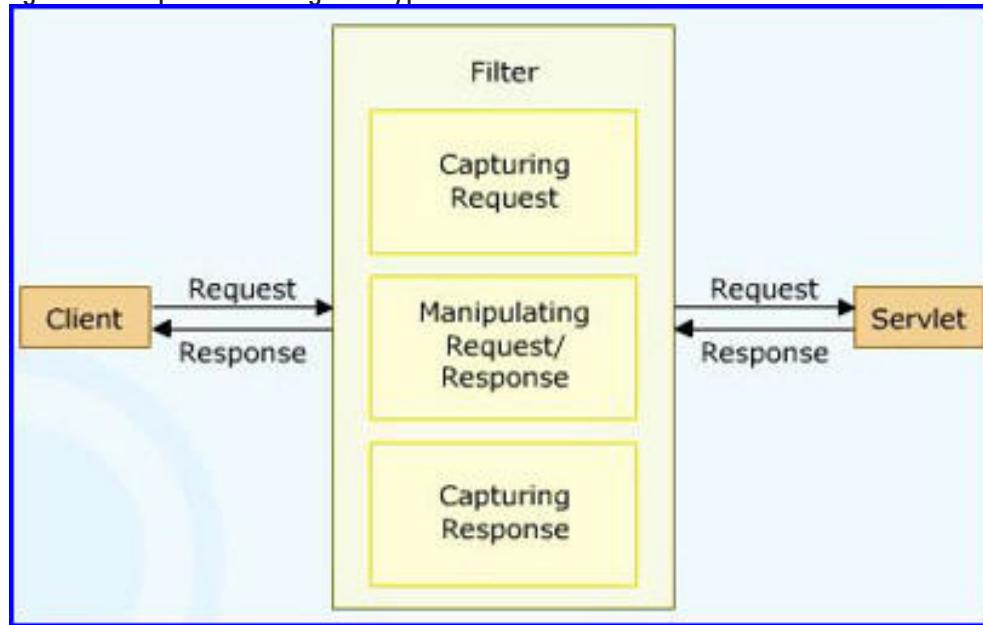


Figure 4.3: Working of a Filter

4.1.4 Usage of Filters

Filters are categorized according to the services they provide to the Web applications. Some of the filter categories are:

Authentication filters

These filters allow the users to access any resource in secured Websites after providing proper username and password.

Logging and auditing filters

These filters track the activities of users on a Web application and log them.

Image conversion filters

These filters scale the image size or change the image type as per requirements.

Data compression filters

These filters help in compressing uploaded or downloaded file size, thereby reducing the bandwidth requirement and time for downloading.

Encryption filters

These filters help in encrypting the request and response header.

Filters that trigger resource access events

These filters help in registering the particular database for a Web application before they retrieve or manipulate data according to the request made by the client.

XML transformation filters

These filters read the XML data from the response and applies an XSLT transformation before sending it to the client browser.

Figure 4.4 depicts a compression filter.

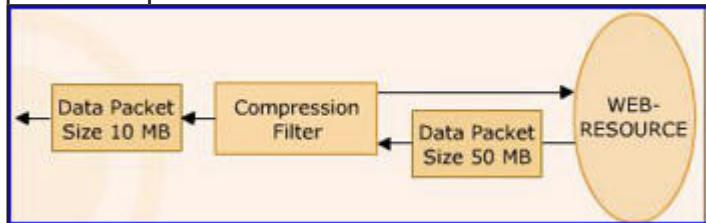


Figure 4.4: Compression Filter



Filters are not Servlets. However, even they have a life cycle for execution.

4.2 Filter API

The filter API brings in all the required interfaces namely `Filter`, `FilterChain`, and `FilterConfig` for creating a filter class into the `javax.servlet` package.

Figure 4.5 depicts hierarchy structure of Filter API.

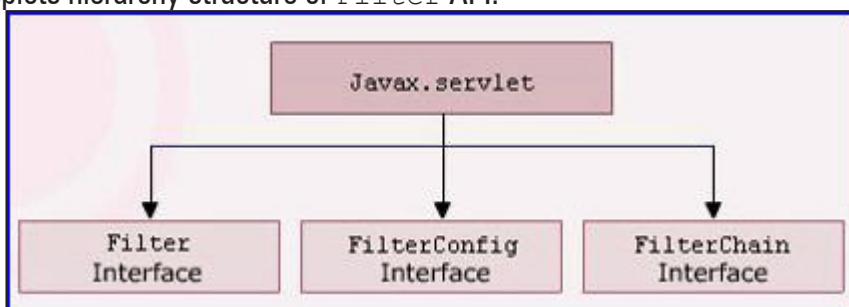


Figure 4.5: Hierarchy of Filter API

Any typical filter needs to implement the `Filter` interface and the methods contained in it.

4.2.1 Filter Interface

The `Filter` interface is part of the Servlet API. The `Filter` interface provides the life cycle methods of a filter. It must be implemented to create a filter class.

The methods of the `Filter` interface are as follows:

`init()`

This is called by the servlet container to initialize the filter. It is called only once. The `init()` method must complete successfully before the filter is asked to do any filtering work.

Syntax:

```
public void init(FilterConfig filterConfig) throws
ServletException
```

where,

- `filterConfig` is the object of `FilterConfig` interface passed as a parameter to the `init()` method.
- `ServletException` is a general exception that can be thrown by a Servlet.

`doFilter()`

The `doFilter()` method of the `Filter` interface is called by the container each time a request or response is processed. It then examines the request or response headers and customizes them as per the requirements. It also passes the request or response through the `FilterChain` object to the next entity in the chain.

Syntax:

```
public void doFilter(ServletRequest req, ServletResponse
res, FilterChain chain) throws IOException, ServletException
```

where,

- `ServletRequest` defines an object to provide client request to a servlet.
- `ServletResponse` defines an object to help the Servlet in sending response to the user.
- `FilterChain` interface creates an object chain to invoke next filter component in the chain.

`destroy()`

This method is called by the servlet container to inform the filter that its service is no more required. This method is also called only once similar to the `init()` method.

Syntax:

```
public void destroy()
```

Figure 4.6 shows the template of a class implementing the `Filter` interface.

```
public class MyGenericFilter implements javax.servlet.Filter {

    public FilterConfig filterConfig;

    public void init(final FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }
    public void doFilter(final ServletRequest request, final
        ServletResponse response, FilterChain chain) throws
        java.io.IOException, javax.servlet.ServletException {
        chain.doFilter(request, response);
    }

    public void destroy() { }

}
```

Figure 4.6: Implementation of Filter Interface

Code Snippet 1 demonstrates the implementation of filter to display a message before the invocation of the configured Servlet.

Code Snippet 1:

```
public class MessageFilter extends Filter {
    private FilterConfig filterConfig;
    public void doFilter(final ServletRequest request,
        final ServletResponse response,
        FilterChain chain)
        throws java.io.IOException, javax.servlet.ServletException {
        System.out.println("Invoking Filter");
        System.out.println("Preprocessing done by filter");
        chain.doFilter(request, response);
    }
}
```

In the code, the `MessageFilter` class implements the `Filter` interface. When the filter will be invoked, the container will pass the `request`, `response`, and `chain` objects to the `doFilter()` method. These objects are passed to the next component in the chain. It can be a filter or a Servlet.

4.3 Configuring Filter

Each filter is a Web component, which along with others forms a complete Web application. To be a part of a Web application, it needs to be configured in the web.xml file. The configuration of a filter inside the `web.xml` file ensures that filter is the part of that application.

In the deployment descriptor, filter is declared by the `<filter>` element. It defines a filter class and its initialization parameters.

Following elements can be defined within a filter element:

- `<display-name>`

This element accepts the short name of the filter, which is intended to be displayed by GUI tools. It is not mandatory inside the `<filter>` element.

- `<description>`

This gives a short description about the filter.

- `<filter-name>`

This defines the name of the filter. This is used to refer to the filter definition somewhere in the deployment descriptor. This element is mandatory.

- `<filter-class>`

It defines the class name of the filter. This is a mandatory element in a filter definition.

- `<init-param>`

It holds the name and value pair of an initialization parameter. It is optional within the `<filter>` element.

4.3.1 Filter-mapping Element

The `<filter-mapping>` element specifies which filter to execute depending on a URL pattern or a Servlet name. This actually sets up a logical relation between the filter and the Web application for sequential control flow of request and response between them.

Following elements are defined inside a filter-mapping element:

- `<filter-name>`

The element inside the deployment descriptor holds the name of the filter to which a URL pattern or a Servlet is mapped. It is a mandatory element for mapping a filter.

- `<url-pattern>`

The element describes a pattern used to resolve URLs. It is a mandatory element for mapping a filter.

- `<servlet-name>`

The element specifies the name of a servlet whose request and response will be modified by the filter. It is a mandatory element.

Code Snippet 2 shows the mapping of MessageFilter with the MyServlet in the web.xml file.

Code Snippet 2:

```
<filter>
    <display-name>Filter</display-name>
    <description>This is my first filter
    </description>
    <filter-name>Message</filter-name>
    <filter-class>
        servlet.filter.MessageFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>Message</filter-name>
    <servlet-name>MyServlet</servlet-name>
</filter-mapping>
...

```

The mapped filter named Message will be executed when the mapped servlet named MyServlet is requested by the client. The <servlet-name> can be replaced with <url-pattern> element to match a specific URL pattern.

4.3.2 Passing Initialization Parameters to Filter

The init() method of a filter initializes the filter. The init() method receives the object of FilterConfig object created by the Web container. Then, the initialization parameters are read in the init() using the methods of FilterConfig.

Table 4.1 lists some of the methods of FilterConfig.

Method	Description
String getFilterName()	This method returns the name of the filter defined in the web.xml or deployment descriptor file in the Web application.
String getInitParameter(String name)	This method returns the value of the named initialization parameter as a string. It returns null if the servlet has no attribute.

Method	Description
Enumeration getInitParameterNames()	This method returns the names of the initialization parameter as an enumeration of String objects. It returns an empty enumeration if the servlet has no attribute.
ServletContext getServletContext()	This method returns the ServletContext object used by the caller to interact with its Servlet container and filter.

Table 4.1: Methods of FilterConfig

Code Snippet 3 exhibits the use of the methods in the `FilterConfig` interface.

Code Snippet 3:

```
public class MessageFilter implements Filter {

    private String message;
    private FilterConfig config;

    public void init(FilterConfig config) throws ServletException {
        this.config = config;
    }
}
```

```

message = config.getInitParameter("message");
}

public void doFilter(final ServletRequest request,
                     final ServletResponse response,
                     FilterChain chain)
    throws java.io.IOException, javax.servlet.ServletException {
    System.out.println("Entering MessageFilter");
    request.setAttribute("messageObj", message);
    chain.doFilter(request, response);
    System.out.println("Exiting MessageFilter");
}
}
}

```

The `init()` method of a filter is called only once, when the filter is instantiated by the container. The `getInitParameter()` method retrieves the value passed as parameter to the filter.

Figure 4.7 shows the configuration of filter with initialization parameters in the `web.xml` file.

```

<filter>
    <filter-name>Message</filter-name>
    <filter-class>servlet.filter.MessageFilter</filter-class>

    <!-- Initialization parameters -->
    <init-param>
        <param-name>message</param-name>
        <param-value>Welcome to Filter</param-value>
    </init-param>

</filter>

<filter-mapping>
    <filter-name>Message</filter-name>
    <servlet-name>MyServlet</servlet-name>
</filter-mapping>

```

Figure 4.7: Filter Initialization

4.3.3 Use of Wildcard in Mappings

The `<url-pattern>` element in the deployment descriptor is used to specify URL pattern for the Web resources. It can either contain a specific URL or a even wild card character (*) can be used to match a set of URLs. All the URLs mapped with filters are applied before the Servlet are invoked.

When a Web application is executed, it creates an instance of each filter that has been declared in the deployment descriptor. The sequence for execution of these filters is in the order, as they are declared in the deployment descriptor.

Code Snippet 4 shows how to include multiple filters with a request to a mapped Servlet.

Code Snippet 4:

```
...
<filter-mapping>
    <filter-name>LogFilter</filter-name>
    <url-pattern>*.abc</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>AuditFilter</filter-name>
    <url-pattern>*.abc</url-pattern>
</filter-mapping>
...
...
```

Consider a situation where a Web page sends a request for a URL,

.../pages/add_customers.abc, then the filters will be applied to the request in the order: LogFilter and then AuditFilter.

Servlet API allows you to specify multiple match criteria in the same entry. This implies that there can be multiple entries for the <url-pattern> element.

You can also have multiple entries for <servlet-name> and <url-pattern> elements inside the <filter-mapping> element.

Code Snippet 5 shows the mapping for multiple patterns with the filter.

Code Snippet 5:

```
...
<filter-mapping>
    <filter-name>DispatchFilter</filter-name>
    <servlet-name>*</servlet-name>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
...
```

In this code, the `<filter-mapping>` element defines a filter named **DispatchFilter** in the `<filter-name>` element to handle all the forward calls to the Servlets. This is achieved using the `<dispatcher>` element having the value set to **FORWARD**.

4.4

Manipulating Requests and Responses

Apart from performing, pre and post processing operations on the request and response objects, the filters can also wrap the request or response objects in custom wrappers. This helps to intercept the request or response, before they can reach their logical destination.

The wrapper classes create the object to capture the request and response, before they reach server and client respectively. Figure 4.8 shows the working of wrapper objects used for processing of request and response in Servlets.

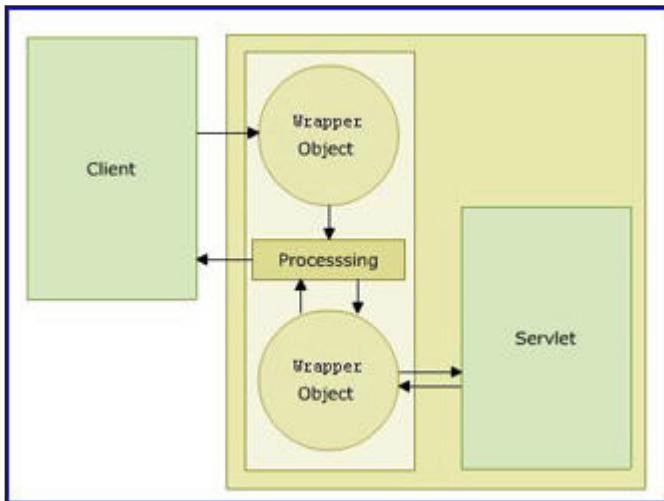


Figure 4.8: Wrapper Objects

The wrapper object generated by the filter implements the `getWriter()` and `getOutputStream()`, which returns a stand-in-stream. The stand-in-stream is passed to the Servlet through the wrapper object. The wrapper object captures the response through the stand-in-stream and sends it back to the filter.

The classes defined by Servlet API for wrapping request and response are as follows:

'ServletRequestWrapper'

This class provides a convenient implementation of the `ServletRequest` interface. The `ServletRequest` can be sub-classed by developers wishing to send the request to a servlet. To override request methods, one should wrap the request in an object that extends `ServletRequestWrapper` or `HttpServletResponseWrapper`.

'ServletResponseWrapper'

This class provides a convenient implementation of the `ServletResponse` interface. The `ServletResponse` can be sub-classed by developers wishing to send the response from a servlet. To override response methods, one should wrap the response in an object that extends `ServletResponseWrapper` or `HttpServletResponseWrapper`.

Code Snippet 6 shows how to extend HttpServletRequestWrapper.

Code Snippet 6:

```
class MyRequestWrapper extends HttpServletRequestWrapper {  
    ...  
    public MyRequestWrapper(HttpServletRequest nested) {  
        super(nested);  
        ...  
    }  
  
    public void setParams(String key, String value) {  
        ...  
    }  
  
    public String getParameter(String name) {  
        String response;  
        ...  
        if (response==null){  
            response=super.getParameter(name);  
        }  
        return response;  
    }  
}
```

Code Snippet 7 shows how to extend `HttpServletResponseWrapper` class.

Code Snippet 7:

```
public class MyResponseWrapper extends HttpServletResponseWrapper {  
  
    public String toString() {  
        ...  
    }  
  
    public MyResponseWrapper (HttpServletResponse response) {  
        super(response);  
        ...  
    }  
  
    public PrintWriter getWriter() {  
        ...  
    }  
}
```

4.5

Introduction to Annotations

There are two approaches adopted in developing Java server-side components namely, imperative programming and declarative programming. The imperative programming approach specifies how to achieve a goal. The declarative programming specifies only the goal, but not the implementation code to achieve that specific goal.

Prior to Java EE 5, the Java EE platform supports the declarative approach where many of the APIs require code to be written in external files such as the deployment descriptors, server specific configurations, and so on. It would be more convenient if the code to be written in the deployment descriptor is maintained as part of the program itself, but in a declarative way. To avoid writing such kind of unnecessary codes, annotations are used.

Annotations are one of the major advancement from Java EE 5.0 that are used to configure the server components. Using annotation in Java EE, makes the standard **application.xml** and **web.xml** deployment descriptors files optional.

4.5.1 Addition or Removal of Excel Add-ins

Annotation can be defined as metadata information that can be attached to an element within the code to characterize it. Annotation simplifies the developer's work to a great extent by significantly reducing the amount of code to be written by moving the metadata information into the source code itself.

Annotation can be added to program elements such as classes, methods, fields, parameters, local variables, and packages. Unlike code, annotations are never executed. Annotations are processed when the code containing it are compiled or interpreted by compilers, deployment tools, and so on.

Processing of annotations can result in the generation of code documents, code artifacts, and so on.

4.5.2 Advantages of Annotations

Some of the advantages of using annotations are as follows:

- **Ease of Use** – Annotations are checked and compiled by the Java language compiler and are simple to use. Many vendors such as IBM and BEA have introduced the annotation feature in attributes for the deployment descriptor.
- **Portability** – Annotations are portable.
- **Type Checking** – Annotations are instances of annotation types and are compiled in their own class files.
- **Runtime Reflection** – Annotations are stored in the class files and accessed for runtime access.

4.5.3 Declaring Annotations

The annotation type is used for defining an annotation and is used when custom annotation needs to be created. An annotation type takes an 'at (@)' sign, followed by the interface keyword and the annotation name.

Alternatively, an annotation takes the form of an 'at' (@) sign, followed by the annotation type. The various standard annotations include @Deprecated, @Override, @SuppressWarnings, @Documented, and @Retention.

Before using annotation in your Web application, there are some rules and guidelines to be followed by a developer. They are as follows:

- A field or a method can be assigned any access qualifier.
- A field or method cannot be static. However, the fields or methods of the main class that have been annotated for injection must be static.
- Resource annotation can be specified in any of the classes or their superclasses.
- Any violation of these rules will generate an error that will result in logging of message or messages.

4.5.4 Support for Annotation

Servlet API supports different types of annotations to provide information. Some annotations can be used as an alternative to XML entries specified in the deployment descriptor, web.xml. Other annotations act as a request for the container to perform tasks that would otherwise be performed by the Servlet.

The `javax.servlet.annotation` package provides annotations to declare Servlets by specifying metadata information in the Servlet class. In addition to this, the `javax.servlet.annotation` package also provides annotations to declare filters and listeners.

Figure 4.9 shows how to configure a Servlet by adding annotations to it.

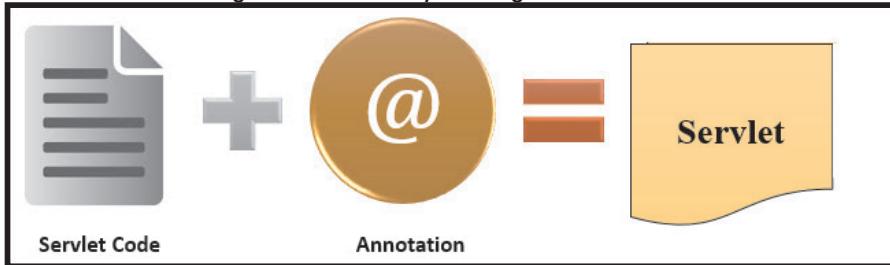


Figure 4.9: Servlet with Annotations

4.5.5 Annotations API

If you use annotations to develop the Web application, there is no need to use the deployment descriptor, `web.xml`. You should use higher version of NetBeans IDE and Web servers to run the Servlets developed using annotations.

The `javax.servlet.annotation` package contains a number of annotations Servlets, filters, and listeners. Using these packages, the developers can develop a complete Web application in annotation.

The `javax.servlet.annotation` package provides different type of annotations.

- Servlet annotations
- Filter annotations
- Listener annotations
- Security annotations

4.5.6 Servlet Annotations

The mapping information of Servlet can be provided in Servlet file itself, instead of `web.xml`, using annotations. The Servlet mapping annotations are supported with the various elements which are optional.

Some of the annotations used by the Servlet are as follows:

- WebServlet**

This annotation is used to provide the mapping information of the Servlet. It is processed by the servlet container at the time of the deployment. The result of processing the annotations provides information about the Servlet to the container such as Servlet class, specified URL pattern to access the Servlet, and so on.

Table 4.2 shows the attributes of `@WebServlet` annotation.

Attribute Name	Description
name	Specifies the Servlet name. This attribute is optional.
description	Specifies the Servlet description and it is an optional attribute.
displayName	Specifies the Servlet display name, this attribute is optional.
urlPatterns	An array of url patterns use for accessing the Servlet, this attribute is required and should register one url pattern.
asyncSupported	Specifies whether the Servlet supports asynchronous processing or not, the value can be true or false.
initParams	An array of <code>@WebInitParam</code> , that can be used to pass servlet configuration parameters. This attribute is optional.
loadOnStartup	An integer value that indicates servlet initialization order, this attribute is optional.
smallIcon	A small icon image for the servlet, this attribute is optional.
largeIcon	A large icon image for the servlet, this attribute is optional.

Table 4.2: `@WebServlet Annotation`

Code Snippet 8 demonstrates how to assign `@WebServlet` annotation to the Servlet class.

Code Snippet 8:

```

    ...
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(
    name = "webServletAnnotation",
    urlPatterns = {" /hello", " /helloWebApp" },
    asyncSupported = false
)

public class HelloAnnotationServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
}

```

```
throws ServletException, IOException {  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    out.write("<html><head><title>WebServlet Annotation</title></head>");  
    out.write("<body>");  
    out.write("<h1>Servlet Hello Annotation</h1>");  
    out.write("<hr/>");  
    out.write("Welcome " +  
        getServletConfig().getInitParameter("name"));  
    out.write("</body></html>");  
    out.close();  
}  
}
```

Note that if the Servlet is already configured in the deployment descriptor, then the deployment descriptor will have higher precedence than annotations. However, this is time consuming since the container has to find all the annotation. In other words, the container has to scan all the files including the jar files present in the classpath.

To avoid this wastage of time, there is an option to disable annotation using the `metadata-complete` attribute in the `web.xml` file. If you set the `metadata-complete` attribute value to true, then the container will ignore all annotations in the Servlet files. In case, the attribute value is set to false or not specified, then by default, it will search for the annotation declaration.

Code Snippet 9 shows the code to disable annotation in the Servlet files with version 2.5.

Code Snippet 9:

```
...  
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"  
    metadata-complete="true">  
</web-app>  
...
```

In this code, the version of Servlet files is specified as 3.1 and the `metadata-complete` attribute value is set to true, so that the container will disable and ignore all annotations applied in the Servlet files.

WebInitParam

This annotation is used to specify the initialization parameters. This can be used with Servlets as well as filters.

Table 4.3 shows the attributes of `@WebInitParam` annotation.

Attribute Name	Description
name	Specifies the names of the initialization parameters.
value	Specifies the value for the initialization parameters.

Table 4.3: Methods of @WebServlet Annotation

Code Snippet 10 shows how to apply the `@WebInitParam` annotation to the servlet.

Code Snippet 10:

```
@WebServlet(
    name = "webServletAnnotation", urlPatterns = {"/hello", "/helloWebApp"},
    asyncSupported = false,
    initParams = {
        @WebInitParam(name = "name", value = "admin"),
        @WebInitParam(name = "param1", value = "paramValue1"),
        @WebInitParam(name = "param2", value = "paramValue2")
    }
)
public class HelloAnnotationServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        ...
        out.write("Welcome " + getServletConfig().getInitParameter("name"));
        ...
    }
}
```

The `initParams` attribute of `@WebServlet` is used that can be used to pass Servlet configuration parameters.

4.5.7 Listener Annotation

The Servlet API provides different types of listener interfaces to handle different events. For example, to handle HttpSession life cycle events, the HttpSessionListener can be used. To declare a class as a listener class, the @WebListener annotation can be used.

The @WebListener annotation is used to register the following types of listeners:

- **Context Listener** - javax.servlet.ServletContextListener
- **Context Attribute Listener** - javax.servlet.ServletContextAttributeListener
- **Servlet Request Listener** - javax.servlet.ServletRequestListener
- **Servlet Request Attribute Listener** - javax.servlet.ServletRequestAttributeListener
- **Http Session Listener** - javax.servlet.http.HttpSessionListener
- **Http Session Attribute Listener** - javax.servlet.http.HttpSessionAttributeListener

Code Snippet 11 shows the implementation of the class that will log the session created or destroyed.

Code Snippet 11:

```
...
...
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class MySessionListener implements HttpSessionListener{

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("Session Created:: ID="+ se.getSession().
getId());
    }
}
```

```
@Override  
public void sessionDestroyed(HttpSessionEvent se) {  
    System.out.println("Session Destroyed:: ID=" + se.getSession().  
    getId());  
}  
}
```

Code Snippet 12 shows the Servlet class to create the HttpSession object.

Code Snippet 12:

```
@WebServlet("/MyServlet")  
public class MyServlet extends HttpServlet {  
    private static final long serialVersionUID = 1L;  
  
    protected void doGet(HttpServletRequest request,  
    HttpServletResponse response) throws ServletException, IOException  
    {  
        HttpSession session = request.getSession();  
        session.invalidate();  
  
    }  
}
```

The code creates an HttpSession object to store the session object. The session object is also destroyed to check the invocation of listeners.

4.5.8 Filter Annotations

In a Web application, the @WebFilter annotation is used to define a filter. It is specified on a class and contains metadata regarding the filter being declared. Atleast one URL pattern must be specified with the annotated filter. The urlPatterns or value attribute on the annotation accomplishes this. The other attributes are optional and have default settings.

Code Snippet 13 shows the creation of filter using @WebFilter annotation.

Code Snippet 13:

```
...
@WebFilter (value="/hello",
    initParams=({
        @WebInitParam(name="message", value="Servlet says: ")
    }))
public MyFilter implements Filter {
    private FilterConfig _filterConfig;

    public void init(FilterConfig filterConfig)
        throws ServletException
    {
        _filterConfig = filterConfig;
    }

    public void doFilter(ServletRequest req,
        ServletResponse res,
        FilterChain chain)
        throws ServletException, IOException
    {

        PrintWriter out = res.getWriter();
        out.print(_filterConfig.getInitParameter("message"));
    }

    public void destroy() {
        // destroy
    }
}
```

4.5.9 Security Annotations

Security annotations can be used to specify permission on the methods of the Servlet class. Some of the security annotations are as follows:

- **ServletSecurity**

This annotation is used on a Servlet class to specify security constraints to be enforced by a servlet container on HTTP protocol messages.

- **HttpConstraint**

This annotation is used within the `ServletSecurity` annotation to represent the security constraints to be applied to all HTTP protocol methods.

For example, to restrict the user from accessing any other method in the Servlet class, when logged in the role as "User", then the annotation will be:

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "User"))
```

- **HttpMethodConstraint**

This annotation is used within the `ServletSecurity` annotation to represent security constraints on specific HTTP protocol messages.

For example,

```
@ServletSecurity(httpMethodConstraint=@  
HttpMethodConstraint(value = "POST"), emptyRoleSemantic = Emp-  
tyRoleSemantic.DENY)
```

4.5.10 Dependency Injection Annotations

Dependency injection (DI) is a mechanism that allows the container to inject the dependent objects in the application components. The application components can be dependent on many resources to perform the necessary operations. For example, datasource object to connect to a database, a JavaBean object, an enterprise bean object, and so on.

The dependent objects are injected before the life cycle methods are invoked and before any reference is made to the dependent object.

Servlet API supports various dependency injection annotations for different types of resources on which Servlet can be dependent. These are as follows:

- `@Resource`
- `@RJB`
- `@PersistenceUnit`
- `@PersistenceContext`
- `@WebServiceRef`

Code Snippet 14 demonstrates the code that can be used to obtain a reference for the data source in the Servlet class.

Code Snippet 14:

```
// Injecting the Datasource named EmployeeDB  
@Resource(name="jdbc/EmployeeDB")  
private javax.jdbc.DataSource myDB;  
  
// Creating connection with the data source  
Connection con;  
  
// Obtaining the connection  
con = myDb.getConnection();
```

The `@Resource` annotation is used for accessing data source registered in JNDI service on the server.

4.6 Uploading Files with Servlet

A very common requirement of a Web application is to upload the files on the server. File can be uploaded on the sever using the following two ways:

1. Client-side file upload
2. Server-side file upload

4.6.1 Client-side Uploading

To upload the files on Web pages, there are some changes required in the HTML form. The requirements are as follows:

- a. Create the input element with the attribute `type="file"`.
- b. The form method supported for file upload will be `POST`, as `GET` method is not supported in file uploading.
- c. Use the attribute named `enctype` with the `form` element. This element specifies how the form data will be encoded when the form is submitted for processing on the server.

Table 4.4 shows the values that can be assigned to the `enctype` attribute.

Value	Description
application/x-www-form-urlencoded	This is the default value if not specified. The value ensures that all characters are encoded before they are sent to the server.
multipart/form-data	The value is provided when the form is using a file upload control. It does not encode the characters before sending them in the request.
text/plain	The value converts the spaces with the "+" symbols, however no other special characters are encoded.

Table 4.4: Values of enctype Attribute

Code Snippet 15 shows the `upload.jsp` page containing the HTML form element to upload the file on the server.

Code Snippet 15:

```
<html>
  <head>
    <title>File Uploading Form</title>
  </head>
  <body>
    <h3>File Upload:</h3>
    Select a file to upload: <br />
    <form action="UploadServlet" method="post"
          enctype="multipart/form-data">
      <input type="file" name="file" size="50"/>
    <br />
    <br />
    <input type="submit" value="Upload File" />
  </form>
</body>
</html>
```

The `file` element displays the **Browse** button on the form which allows the user selects the file from the system. When the user click `Upload File`, the HTTP request will pass the uploaded file to the Servlet, which will write the uploaded file to the folder on the server.

Figure 4.10 shows the output of the JSP page.

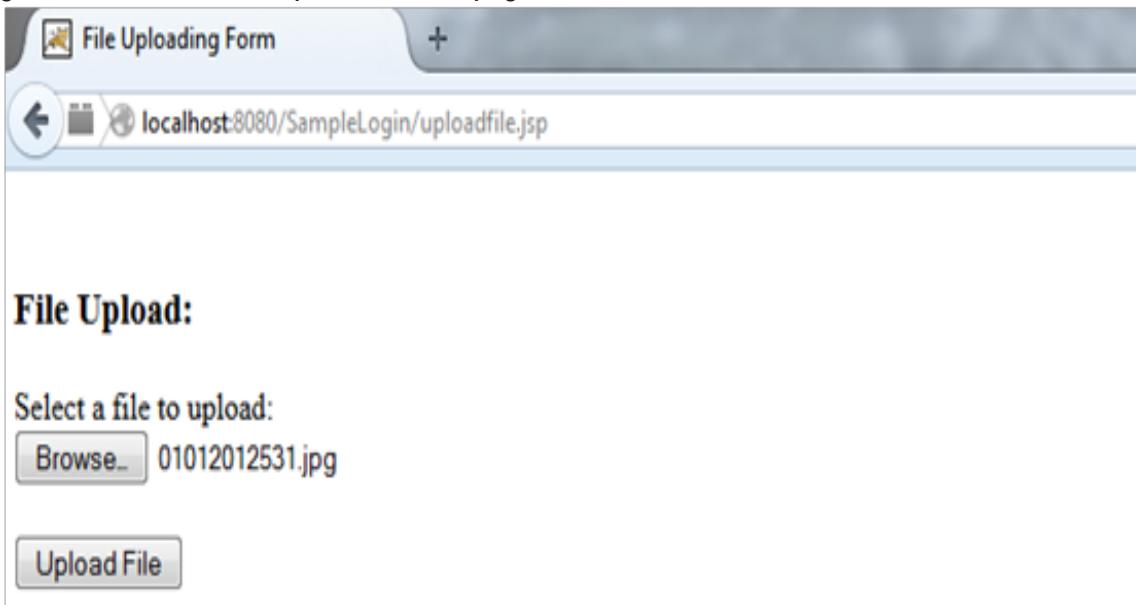


Figure 4.10: JSP File with Single File Upload

4.6.2 Server-side File Upload

In server-side file upload, the files are uploaded using Servlet class. The Servlet specification provides the classes that are used to handle encrypted from data. The classes that are involved in handling the file uploading in the Servlet are `HttpServletRequest`.

The `HttpServletRequest` class includes two methods for handling the file upload. These are

- `Part getPart(String name)` – This method extracts the individual parts of the file and returns each as a `Part` object. The `Part` object belongs to the `Part` interface and provide methods that can be used to extract information from the returned `Part` object. The information includes: the name, the size, and the content-type of the file returned as a part. It also provide methods for querying the header content submitted with the `Http request` object, writing the part to the external disk, and deleting the part.
- `Collection<Part> getParts()` – This method returns a collection of `Part` objects which can be iterated to extract individual `Part` object from the collection.

Code Snippet 16 demonstrates the Servlet code that uploads the file on the server.

Code Snippet 16:

```
public class UploadServlet extends HttpServlet {  
    public void doPost (HttpServletRequest request,  
    HttpServletResponse response) throws IOException {  
        response.setContentType ("text/html");  
        PrintWriter out = response.getWriter();  
        boolean isMultipartContent = ServletFileUpload.  
        isMultipartContent (request);  
        if (!isMultipartContent) {  
            out.println ("No file uploaded<br/>");  
            return;  
        }  
  
        out.println ("You are trying to upload:<br/>");  
        FileItemFactory factory = new DiskFileItemFactory ();  
        ServletFileUpload upload = new ServletFileUpload (factory);  
        try {  
            List<FileItem> fields = upload.parseRequest (request);  
            out.println ("Number of fields: " + fields.size () + "<br/><br/>");  
            Iterator<FileItem> it = fields.iterator();  
            if (!it.hasNext ()) {  
                out.println ("No fields found");  
                return;  
            }  
  
            while (it.hasNext ()) {  
                FileItem fileItem = (FileItem) it.next();  
                boolean isFormField = fileItem.isFormField();  
            }  
        }  
    }  
}
```

```
        if (!isFormField) {  
  
    String fileName = fileItem.getName();  
  
    File file = new File(fileName);  
  
    fileItem.write(file);  
  
    out.println("Uploaded Filename: " + fileName +  
    "<br>");  
    }  
}  
}  
}  
}  
catch (Exception e) {  
    e.printStackTrace();  
}  
}  
}  
}
```

Code Snippet 17 shows the corresponding `web.xml` file for configuring the Servlet.

Code Snippet 17:

```
<servlet>  
    <servlet-name>UploadServlet</servlet-name>  
    <servlet-class>com.Upload.UploadServlet</servlet-class>  
</servlet>  
  
<servlet-mapping>  
    <servlet-name>UploadServlet</servlet-name>  
    <url-pattern>/UploadServlet</url-pattern>  
</servlet-mapping>  
  
<welcome-file-list>  
    <welcome-file>uploadfile.jsp</welcome-file>  
</welcome-file-list>  
</web-app>
```

4.6.3 MultipartConfig Annotation

The `javax.servlet.annotation.MultipartConfig` annotation is specified on a Servlet class. It provides file information to the server while uploading its content.

The information provided by the `MultipartConfig` annotation is as follows:

- The maximum size of the data that can be uploaded on the server
- The default location where the file will be uploaded on the server

Table 4.5 lists the attributes of the `MultipartConfig` annotation.

Attribute	Description
<code>fileSizeThreshold</code>	Specifies the size of the file in bytes. The default size of the file is 0.
<code>maxFileSize</code>	Specifies the maximum file limit allowed to upload the file on the server. If the size of the file to be uploaded is greater than the specified limit, then throws <code>IllegalStateException</code> .
<code>maxRequestSize</code>	Specifies the maximum file size in bytes to be accepted for encrypted file data sent in the request.
<code>location</code>	Specifies the default location for the file upload on the server.

Table 4.5: Methods of `MultipartConfig` Annotation

Code Snippet 18 shows the attributes of @multipartconfig annotation that are applied to the Servlet.

Code Snippet 18:

```
@WebServlet(name="UploadServlet", urlPatterns={"/Upload"})
@MultipartConfig(
    fileSizeThreshold=100,
    maxFileSize=400,
    maxRequestSize=50,
    location="C:/tmp")

public class UploadServlet extends HttpServlet{
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws IOException {
        . . .
        . . .
    }
}
```

Check Your Progress

1. HTTP is a _____ layer protocol.

(A)	Filter was introduced as a Web component in Java servlet Specification version 2.3.
(B)	A filter optimizes the time it takes to send back a response from your Web server.
(C)	A logging and auditing filter allows users to log in to a Web application.
(D)	Filter optimizes the available bandwidth on the network thereby reducing the network traffic and in turn increasing the efficiency of all Web-based services.
(E)	An authentication filter tracks the activities of users of a Web application.

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

2. Match the methods to their appropriate descriptions.

Description		Method	
(A)	This method returns the names of the initialization parameter as an enumeration of String objects.	(1)	getInitParameter(name)
(B)	This method returns the name of the filter defined in the web.xml or deployment descriptor file inside the Web application.	(2)	getInitParameterNames()
(C)	This method returns the Servlet Context object used by the caller to interact with its servlet container and filter.	(3)	getFilterName()
(D)	This method returns the value of the named initialization parameter as a string.	(4)	getServletContext()

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

Check Your Progress

3. Which of these statements is incorrect?

(A)	The FilterChain interface provides an object through the Web container.	(C)	The object invokes the previous filter in a filter chain starting from the first filter from a particular end.
(B)	The doFilter(request, response) method invokes the next filter in the filter chain.	(D)	If the calling filter is the last filter in the chain, it will invoke the Web resource, such as JSP and servlet.

4. Match the elements to their appropriate features.

Feature		Element	
(A)	This element inside the deployment descriptor holds the name of the filter to which a URL pattern or servlet is mapped.	(1)	<icon>
(B)	This specifies the location of a small or a large image used to represent the filter in a GUI tool within a Web application.	(2)	<init-param>
(C)	It holds the name and value pair of an initialization parameter of the filter.	(3)	<filter-name>
(D)	This element specifies the name of a servlet whose request and response will be modified by the filter.	(4)	<display-name>
(E)	This is the short name of the filter, i.e. intended to be displayed by GUI tools.	(5)	<servlet-name>
(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-5, C-2, D-3, E-1
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-3, B-1, C-2, D-5, E-4

Check Your Progress

5. Which of these statements is correct about altering a response using filters?

(A)	The request wrapper object is generated by the filter by extending to <code>ServletResponseWrapper</code> class.	(C)	The stand-in-stream allows the servlet from closing the response generated by it.
(B)	The stand-in-stream is then passed to the servlet through the response wrapper object.	(D)	The request object overrides the <code>getWriter()</code> and <code>getOutputStream()</code> methods.

6. Which of the following annotation will help to create a Servlet?

(A)	@WebListener	(C)	@MultipartConfig
(B)	@WebServlet	(D)	@WebInitparam

7. Which type of input type we can use on the client-side to upload the file on the server?

(A)	text	(C)	button
(B)	file	(D)	option

8. Which of the following annotation is used to create the filter?

(A)	@WebFilter	(C)	@MultipartConfig
(B)	@WebServlet	(D)	@WebInitparam

Answer

1.	C
2.	B
3.	C
4.	D
5.	B
6.	B
7.	B
8.	A

Summary

- Filter acts as an interface between client and servlet inside the server. It intercepts the request or response to and from the server. More than one filter can also exist between a client and a servlet. This is called a filter chain.
- The Filter API is a part of the servlet package. It contains interfaces namely, Filter, FilterConfig, and FilterChain.
- All the information about the filter is described within the <filter> element inside the web.xml file. The filter also needs to be mapped to the servlet for which it will function. This is done by assigning URL name and servlet name inside the <filter-mapping> element.
- The request and response to and from the servlet is manipulated by the filter. The wrapper object of RequestWrapper or ResponseWrapper classes created by the filter intercepts the request or response and sends it to the filter.
- Annotation can be defined as metadata information that can be attached to an element within the code to characterize it. Annotation can be added to program elements such as classes, methods, fields, parameters, local variables, and packages.
- The javax.servlet.annotation package contains a number of annotations for Servlets, filters and listeners. Apart from this, Servlet API also supports various dependency injection annotations.
- File can be uploaded on the sever using the following two ways namely, Client-side file upload and server-side file upload.
- The javax.servlet.annotation.MultipartConfig annotation is specified on a Servlet class to provide file information to the server while uploading its content.



Welcome to the Session, **Database Access and Event Handling**.

This session explains the concept of establishing the database connection in the Servlet using the Java Database Connectivity (JDBC) API. The session also explains the concept of Object Relational Mapping (ORM) and persist the entities in the database using Java Persistence (JPA) API. Further, the session explains the event-handling mechanism in Servlets.

In this Session, you will learn to:

- Explain database handling using JDBC
- Describe JDBC and its role
- Explain connecting database using JDBC
- Describe JPA and its role
- Describe connecting database using JPA
- Explain the significance of session handling and session events
- Explain different types of listener interfaces used in Servlets

5.1 Introduction

The most common operation performed by the Servlet is storing and retrieving database information.

Java provides various mechanisms using which data can be accessed from the database. These are as follows:

- Using Java Database Connectivity (JDBC) API - It is a part of Java SE platform that enables a Web application to interact with a Database Management System (DBMS).
- Using Object Relational Mapping (ORM) API – It is a persistence mechanism used by enterprise application to access relational databases. It maps the Java objects to database tables through XML configuration. The Java Persistence API (JPA) is one such specification that is based on ORM.

5.2 JDBC API

The JDBC API provides classes and interfaces that allow a Web application to access and perform operations on the databases. The operations that can be performed on the databases include:

- Connecting to the databases
- Processing SQL statements
- Building result sets
- Executing the parameterized statements
- Invoking callable statements

Figure 5.1 shows the layers of the JDBC architecture.

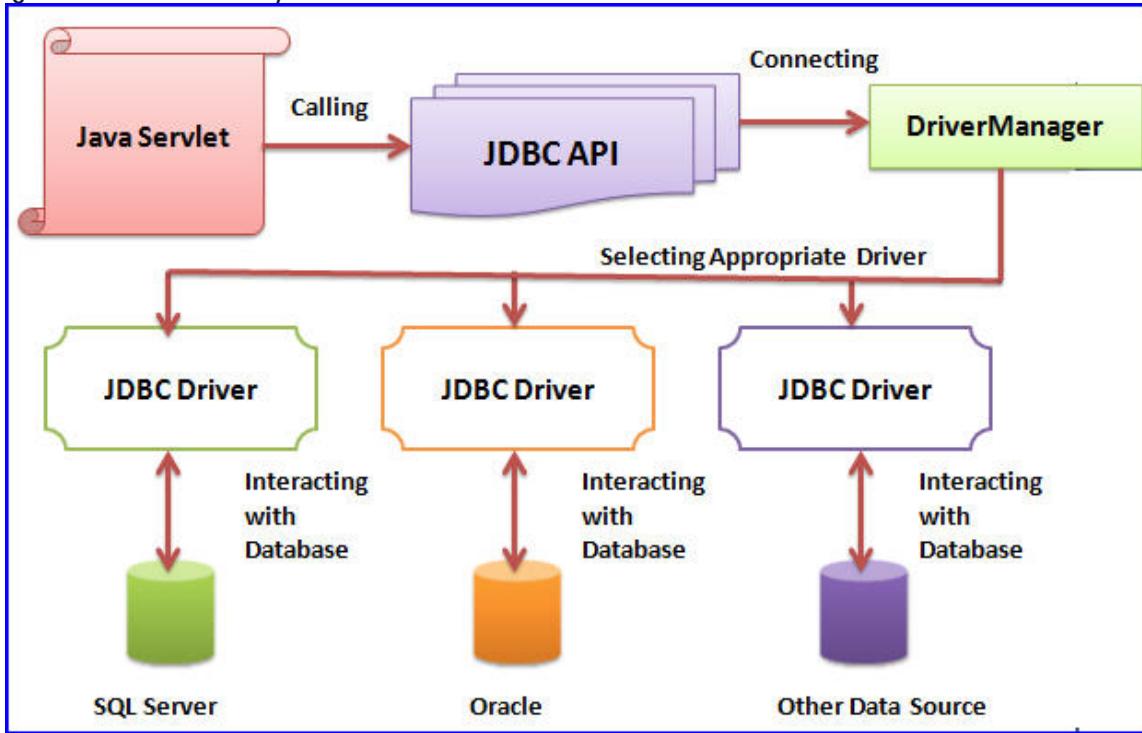


Figure 5.1: JDBC Architecture

The Servlet uses JDBC APIs to access database. The JDBC API uses a driver manager, which ensures that the correct driver is used for accessing a specific database. A JDBC driver translates standard JDBC calls into a client's API specific calls. These API calls facilitate communication with the database.

The application developer using JDBC API need not bother about repackaging the application, if the database is changed or upgraded.

Note, that the JDBC drivers are database specific.

5.2.1 JDBC Driver Types

There are four different types of JDBC drivers that are as follows:

- **Type 1: JDBC-ODBC Bridge Plus ODBC Driver** – The JDBC-ODBC bridge provides access to the database through standard ODBC libraries. The driver translates the standard JDBC calls to the corresponding ODBC calls.
This driver is a part of the `sun.jdbc.odbc` package. Its major limitation is that the JDBC driver's capabilities depend on the capabilities of the ODBC driver.
- **Type 2: Native API Partly Java Driver** – The native APIs partly Java driver calls the native database library that accesses the database. This driver also requires the native database libraries to be installed and configured on the client's machine.

- **Type 3: JDBC-Net Pure Java Driver** – This is a pure Java driver that communicates with the JDBC middle tier on the server through a proprietary network protocol. The Java Servlet sends a JDBC call to the middleware through the JDBC driver without translation. The middleware then sends the request to the database.
- **Type 4: Native-Protocol Pure Java Driver** – The type 4 drivers implement a proprietary database driver. It provides database connectivity through the standard JDBC APIs and do not require any native libraries on the client machine. These drivers are the fastest.

5.3 Accessing Database Using JDBC

The Servlet interacts with the database using JDBC APIs and follows certain steps in sequence:

1. Registering the JDBC driver
2. Establishing a database connection
3. Executing an SQL statement
4. Processing the results
5. Closing the database connection

5.3.1 Registering the JDBC Driver

Registering the driver notifies the driver manager that a JDBC driver is available. The JDBC automatically registers itself with the driver manager as soon as it is loaded.

The JDBC driver can be loaded at run time using the `forName()` method.

Syntax:

```
Class.forName (String driver-name)
```

The `forName()` is a static method that instructs the JVM to locate, load, and link the specified driver classes.

5.3.2 Establishing a Database Connection

The database connection can be established when a valid driver is loaded. The JDBC connection is identified by a database URL.

The syntax for specifying the URL is as follows:

Syntax:

```
jdbc:<subprotocol>:<database_identifier>
```

The first part specifies that JDBC is used for establishing the connection. The <subprotocol> is the name of any valid database driver or any other database connectivity that is being requested. The <database_identifier> is the logical name of the database connection that maps to the physical database. It is basically the Data Source Name (DSN) that holds the information to establish a connection.

The database connection is established using the `getConnection()` method of the `DriverManager` class in JDBC API.

Code Snippet 1 demonstrates the connecting to a SQL Server database from the Servlet.

Code Snippet 1:

```
public class JDBCServlet implements HttpServlet
{
    ...
    Connection conn = null;
    try {
        // Loads Type 4 driver for connecting to SQL Server 2012 database
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        // Obtains a connection to the database
        conn = DriverManager.getConnection("jdbc:sqlserver://
localhost:1433;" +
+ "database=ProductStore;user=sa;password=sa;");
        ...
    } catch (ClassNotFoundException ex) {
        ...
    }
}
```

In the given code snippet, the `Conn` object is created using the `getConnection()` method of the `DriverManager` class.

5.3.3 Executing an SQL Statement

Once the database connection is established, several SQL statements such as retrieving, inserting, updating, or deleting of data can be performed on the database. In order to query the database, the `Statement` interface of JDBC API can be used.

The `Statement` object reference can be created using the `createStatement()` method on the obtained connection.

For example, the following statement is used to create the `Statement` object:

```
Statement stat = conn.createStatement();
```

The `Statement` interface provides methods to perform various database operations including SQL query execution. The `executeQuery()` method of `Statement` interface accepts SQL query as argument and returns a `ResultSet` object.

For example, the following statement shows the creation of query using `Statement` object.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Product");
```

In case of any error, the `executeQuery()` method throws an exception – `SQLException`.

The `ResultSet` object contains all the rows returned by the SQL query.

5.3.4 Processing the ResultSet

`ResultSet` is an interface used to describe `ResultSet` object. This object represents data in a tabular form that is retrieved by executing an SQL query.

Some of the features of `ResultSet` object are as follows:

- `ResultSet` follows the iterative pattern.
- `ResultSet` object maintains a cursor which points to the current row in the result table.

There are three types of `ResultSet` as follows:

- `TYPE _ FORWARD _ ONLY`

In this, the cursor moves only in forward direction in the resultset.

- `TYPE _ SCROLL _ INSENSITIVE`

In this, the cursor scrolls forward and backward. The resultset is not sensitive to changes, which have been made by other users to the database.

- `TYPE _ SCROLL _ SENSITIVE`.

In this, the cursor scrolls forward and backward. The resultset is sensitive to changes, which have been made by other users to the database.

Code Snippet 2 demonstrates the code that iterates through the resultset.

Code Snippet 2:

```
// Access column values
while(rs.next()) {
    rs.getInt ("product_id");
    rs.getString ("product_name");
}
```

Iterating the `ResultSet` includes `next()` method which moves the cursor forward one row. This returns true, if the cursor is now positioned on a row and false, if the cursor is positioned after the last row.

Code Snippet 3 demonstrates the code that is used to update the resultset.

Code Snippet 3:

```
rs.updateString("product_name", "Beverages");
rs.updateRow();
```

5.3.5 Closing the Connection

It is particularly important to close database-related connections. In case we do not close them, there will be unclosed connections to the database and it could eventually run out of connections.

The statement, `conn.close()` closes the connection explicitly allows garbage collector to recollect memory as early as possible.

5.4 Java Persistence API (JPA)

Java Persistence API or JPA is a lightweight, POJO-based Java framework. It helps to persist the Java Objects to the relational database. JPA is an ORM technology that persist the entities in the database. The main benefit of using an ORM technology is that it supports the mapping to object-oriented software to the database schema through configuration mapping. The configuration mapping information about the entity in the application domain to the database table is specified either in XML file or through annotations.

JPA provides the persistence providers in the JPA application. JPA converts data between incompatible type in object-oriented programming to the appropriate underlying database types. This provides the database and schema independence. Generally, converting an object-oriented domain schema into relational schema is a time consuming process. ORM software provides this mapping in less time and requires less or almost no coding. It supports a simple mapping of Java objects to database tables.

Figure 5.2 shows the ORM technology such as JPA support for Java objects to database tables.

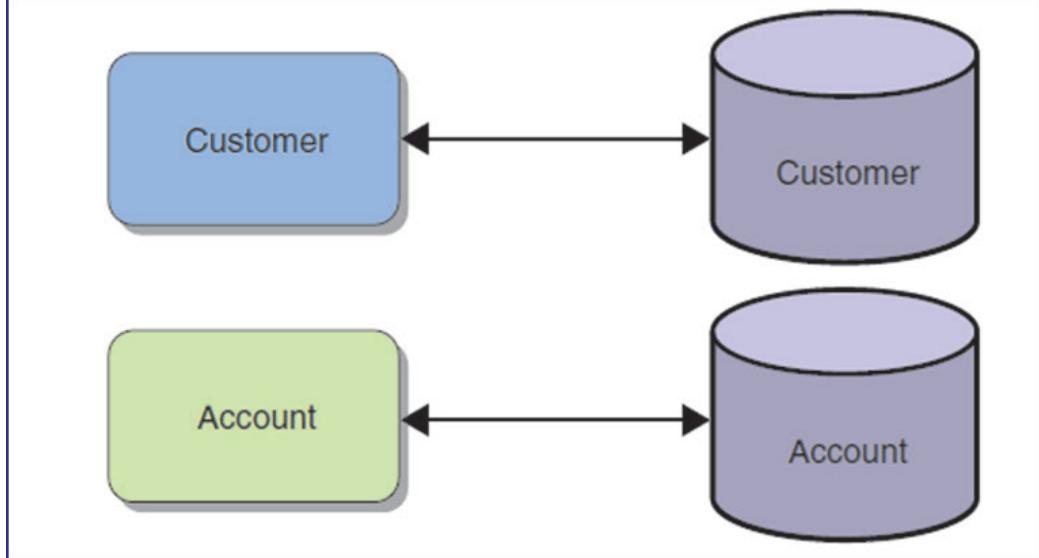


Figure 5.2: Mapping of Java Objects to Tables

Entities are annotated Plain Old Java Objects (POJOs). They are mapped to the tables in a relational database. While working with entities, it is required to declare an entity explicitly. This is done to distinguish it from other regular Java objects that might be used in the application. Java annotations or XML is used to define the mapping of entity to existing database tables. The JPA specification provides the persistence manager which manages the life cycle of an entity in the persistence context. It also provides the interfaces to execute both static and dynamic queries.

5.4.1 JPA Specification

Sun Microsystems has laid the JPA specification which is light weight and implemented by different ORM products on Java EE platform, such as Hibernate, TopLink, and so on.

The features of JPA specification are as follows:

- It supports POJO programming model for persisting objects into the database. The POJOs persisted in the database by JPA are also referred as entities.
- It provides a standard ORM API which incorporates concepts present in third-party persistence frameworks such as Hibernate, Toplink, Java Data Object (JDO), and so on.
- It defines a service provider interface implemented by different persistence providers. Thus, any change in persistence provider does not affect the entities developed in the application.

5.4.2 JPA Components

Figure 5.3 shows the primary components of JPA API.

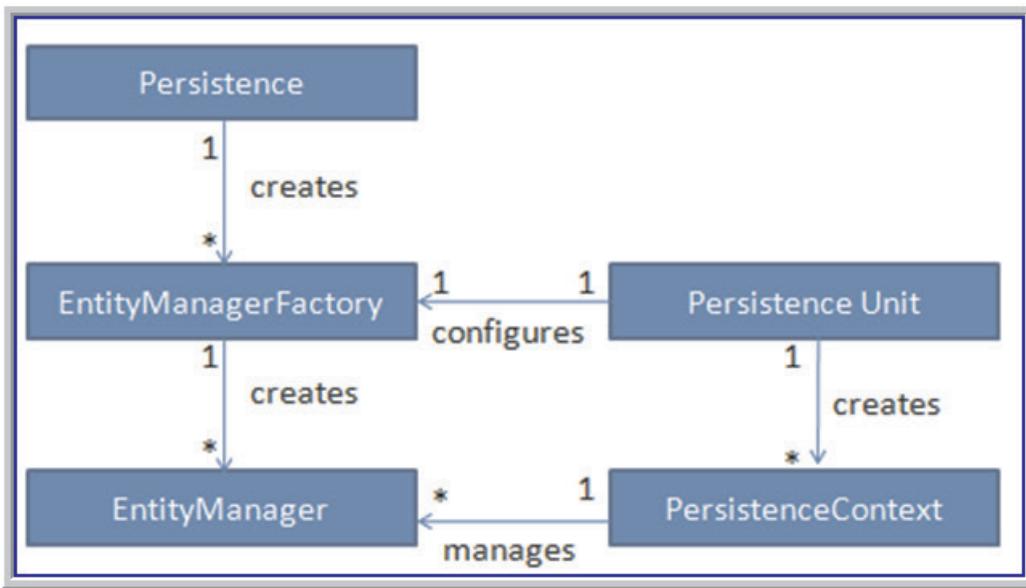


Figure 5.3: Primary Components of JPA API

Components of JPA specification are as follows:

□ Persistence

The persistence is an identity which is a unique value used by the container to map the entity object to the corresponding record in the database.

□ EntityManager

The EntityManager interface is providing the API for interacting with the Entity. The function of EntityManager API is to manage entity life cycle instances. It is the core component of the JPA API is used to create, retrieve, update, and delete data from a database.

□ EntityManagerFactory

The EntityManagerFactory is used to create an instance of EntityManager. In your application, when there is no use of EntityManagerFactory or application shuts down, then it is necessary to close the instance of EntityManagerFactory. Once the EntityManagerFactory is used to create an instance of EntityManager. In your application, when there is no use of EntityManagerFactory or application shuts down, then it is necessary to close the instance of EntityManagerFactory. Once the EntityManagerFactory is closed, all its EntityManager instances are also closed.

- **Persistence Unit**

Persistence unit consists of XML configuration information and a bundle of classes that are controlled by the JPA provider.

- **Persistence Context**

Persistence context is associated with entity manager. A persistence context is a set of entities such that for any persistent identity there is a unique entity instance. It can be considered as the working copy of persistence unit.

5.4.3 Entities in JPA

In JPA, persistent objects are referred as Entities. Entities are plain old Java objects that are persisted to relational databases or legacy systems.

Typically, an entity represents a table in the database and each instance of an entity maps to a row in that table. The entities store data as fields or properties and have getter/setter methods associated with each of the fields. For example, the employee details can be stored in an entity named Employee entity.

Some of the requirements of entity class are as follows:

- The entity class must be annotated with `@Entity` annotation defined in `javax.persistence` package.
- The class must have a no-argument constructor with public or protected access modifier.
- The properties or methods of the entity class should not be declared as final. The class also should not be declared as final.
- If the entity instance is passed as parameter to a Session Bean remote interface, then it must implement the `Serializable` interface.
- Entities can be inherited from entity or non-entity classes.
- The instance variables of entity class must be declared as private or protected and are accessed only by the methods present in the class.
- Each entity instance must be identified by a unique identifier or a primary key. The primary key helps the client to access the entity instance. The `@Id` annotation applied on the entity property specifies that the property is a primary key.

Creating instances of this class using the `new()` keyword will not create an `Employee` class object in the database. An instance of `EntityManager` is required to create the entity in the database.

5.4.4 Managing Entities in the Persistence Context

A persistence context represents a set of managed entity instances that exist in a particular data store. It is a connection between the entity instances present in the memory and the database. When the entity instance is newly created, it does not have any persistence context associated with it. At this stage, even the database does not have information about them.

When entities are persisted or retrieved from the database, they are associated with the persistence context. To manage the entity instances in the persistence context, an interface named `EntityManager` API is used. The `EntityManager` interface is defined in `javax.persistence` package and is used to interact with the persistence context. The `EntityManager` interface provides methods to synchronize the state of the entity instance to the database.

Figure 5.4 shows the working of `EntityManager` in JPA.

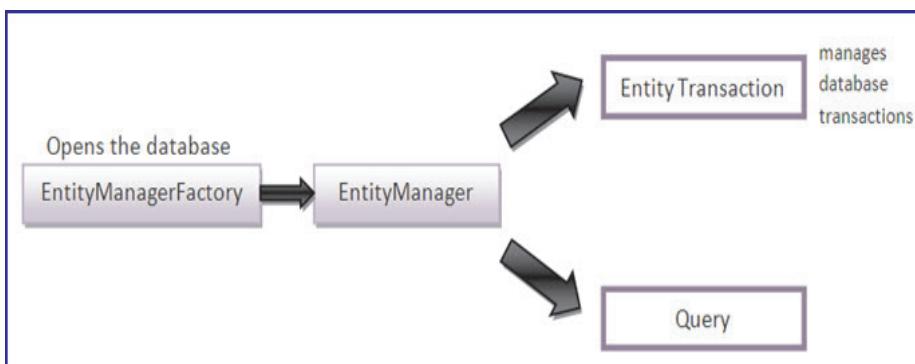


Figure 5.4: EntityManager in JPA

Table 5.1 lists some of the methods defined in the `EntityManager` interface.

Methods	Description
<code>public void persist (Object entity)</code>	The method saves an entity into the database and makes the entity managed.
<code>public <T> T merge (T entity)</code>	The method merges an entity to the EntityManager's persistent context and returns the merged entity.
<code>public void remove (Object entity)</code>	The method removes an entity from the database.
<code>public void flush()</code>	The method synchronizes the state of an entity in the EntityManager's persistent context with the database.

Table 5.1: Methods in EntityManager Interface

5.4.5 Packaging and Deploying Entity Classes

The packaging and deploying of an entity class is done using persistence units. A persistence unit is a set of class mapped to a particular database. This persistence unit is defined in a special descriptor file named `persistence.xml`. This file contains logical grouping of entity classes, their metadata mappings, and configurations related to a database.

The configuration details in the `persistence.xml` are used while obtaining the `EntityManager` instance in the client program. The `EntityManager` instance accordingly persists the entity instances in the database.

Code Snippet 4 displays the code for a `persistence.xml` file developed as part of the enterprise application.

Code Snippet 4:

```
...
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence">
<persistence-unit name="intro" />
</persistence>
...
```

In the given code snippet, the `persistence.xml` file can contain one or more `<persistence-unit>` elements under the root element, `<persistence>`. Each `<persistence-unit>` element defines the configuration for a single data source. Each `<persistence-unit>` element is identified by the name attribute. The name attribute is mapped to the `unitName` attribute of the `@PersistenceContext` annotation used in the client program.

Some of the attributes defined in the `<persistence-unit>` element are as follows:

- `<description>` - It describes the persistence unit and is optional.
- `<provider>` - It must be present in the J2SE environment or when the application requires a provider specific behavior.
- `<transaction-type>` - It has a value either as Java Transaction API (JTA) or `RESOURCE_LOCAL` or by default, the value is JTA.
- `<jta-data-source>/<non-jta-data-source>` - It is used for specifying the Java Naming and Directory Interface (JNDI) name of the data source. JNDI is used by the persistence provider.
- `<mapping-file>` - It contains a list of one or more additional XML files that are used for O/R mapping. The mapping file is used to list the entity classes that are available in the persistence unit.
- `<properties>` - It specifies the configuration properties that are vendor-specific for the persistence unit. Any properties that are not recognized by the persistence provider are ignored.

5.4.6 Accessing Database Using JPA

Let us consider an example for inserting and displaying of records of the players from the database. The Web application containing JSP pages, Entities, and Servlet classes is created for performing operations on the players details stored in the database.

Code Snippet 5 shows a JSP file with a form to add new records to the list.

Code Snippet 5:

```
<!-- addPlayers.jsp -->
<body>

    <h1>New Player record</h1>
    <form id="addNewPlayersForm" action="addPlayers" method="post">
        <table>
            <tr><td>Rank</td><td><input type="text" id = "rank" name="rank" /></td></tr>
            <tr><td>Player</td><td><input type="text" id = "playerName" name="playerName" /></td></tr>

            <tr><td>Team</td><td><input type="text" id = "team" name="team" /></td></tr>
        </table>
        <input type="submit" id="CreateRecord" value="CreateRecord" />
    </form>
    <a href="DisplayPlayers"><b>Complete List</b></a>
</body>
</html>
```

Code Snippet 6 shows the code to display the records which are also accessible from other pages.

Code Snippet 6:

```
<!-- DisplayPlayers.jsp -->
<body>

    <h1>ICC <b>ODI</b> Players Ranking List</h1>

    <table id=" PlayerTBLList" border="2" bgcolor="#B0B0B0">
<tr >
    <th>Rank</th>
    <th>Player</th>
    <th>Team</th>
</tr>
<c:forEach var="cricketer" begin="0" items="${requestScope.playerList}">
<tr>
    <td>${cricketer.rank}&nbsp;&nbsp;</td>
    <td>${cricketer.playerName }&nbsp;&nbsp;</td>
    <td>${cricketer.team}&nbsp;&nbsp;</td>
</tr>

</c:forEach>

</table>
<a href="addPlayers.jsp"><b>Add New Player Record</b></a>
</body>
</html>
```

Code Snippet 7 shows the index.jsp page.

Code Snippet 7:

```
<!-- index.jsp -->
<html>
    <head>
    </head>
    <body>
        <jsp:forward page="DisplayPlayers" />
    </body>
</html>
```

Code Snippet 8 shows the entity class named Players.

Code Snippet 8:

```
/** Players.java **/



@Entity
@Table(name = "PLAYERS")
public class Players {
    // column mapping
    @Rank
    @Column(name = "RANK")
    private String rank;
    @Column(name = "PLAYER")
    private String playerName;
    @Column(name = "TEAM")
    private String team;
    public Players() { }
    public Players(String rank, String playerName, String team) {
        this.rank = rank;
        this.playerName = playerName;
        this.team = team;
    }
}
```

```
public String getRank() {  
    return this.id;  
}  
public String getPlayer() {  
    return this.playerName;  
}  
  
public String getTeam () {  
    return this.team;  
}  
}
```

The code snippet defines a Java Persistence API entity class. To define an Entity class means the variable is declared and annotated.

Code Snippet 9 shows the Servlet is for adding new players.

Code Snippet 9:

```
/** addNewPlayersServlet.java **/  
@WebServlet(name="addNewPlayersServlet", urlPatterns={"/  
addPlayers"})  
  
public class addNewPlayersServlet extends HttpServlet {  
  
    @PersistenceUnit  
    //The eManagerFactory corresponding to  
    private EntityManagerFactory eManagerFactory;  
  
    @Resource  
    private UserTransaction uTransaction;  
  
    /** Processes requests for both HTTP <code>GET</code> and  
    <code>POST</code> methods.  
     * @param request servlet request  
     * @param response servlet response  
     */
```

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException {
    assert eManagerFactory != null;
    EntityManager eManager = null;
    try {

        // the data from user's form
        String rank = (String) request.getParameter("rank");
        String playerName = (String) request.getParameter("playerName");
        String team = (String) request.getParameter("team");

        // player instance
        Players player = new Players(rank, playerName, team);
        // begin transaction uTransaction
        uTransaction.begin();
        // since the eManager is created inside a transaction, it is
        // associated with the transaction
        eManager = eManagerFactory.createEntityManager();
        //persist the player entity
        eManager.persist(player);

        // commit transaction uTransaction
        uTransaction.commit();
        request.getRequestDispatcher("DisplayPlayers").forward(request, response);
    } catch (Exception ex) {
        throw new ServletException(ex);
    } finally {
        //close the eManager to release any resource
        if(eManager != null) {
            eManager.close();
        }
    }
}
```

EntityManagerFactory opens the database when it is constructed. An EntityManagerFactory supports the instantiation of EntityManager instances. EntityManagerFactory forms construct multiple EntityManager instances for a specific database by managing resources efficiently and should be closed when the application ends.

The EntityManager instance represents the following:

- A remote connection to remote database server in client server mode.
- A local connection to a local database file in embedded mode.

Database transactions are managed by the EntityTransaction interface. As shown in the code snippet, every EntityManager holds a single attached EntityTransaction instance.

Code Snippet 10 shows the code for displaying players data in the Servlet.

Code Snippet 10:

```
/** DisplayPlayersServlet.java **/  
  
@WebServlet(name="    DisplayPlayersServlet    ",    urlPatterns={"/  
DisplayPlayers"})  
public class DisplayPlayersServlet extends HttpServlet {  
  
    @PersistenceUnit  
    private EntityManagerFactory eManagerFactory;  
  
    /** Processes requests for both HTTP <code>GET</code> and  
     * <code>POST</code> methods.  
     * @param request servlet request  
     * @param response servlet response  
     */  
  
    protected void processRequest(HttpServletRequest request,  
HttpServletResponse response)  
        throws ServletException, IOException {  
        assert eManagerFactory != null;  
        EntityManager eManager = null;
```

```
try {  
    eManager = eManagerFactory.createEntityManager();  
    //query for all the players in database  
    List players = eManager.createQuery("select p from  
Player p").getResultList();  
    request.setAttribute("playerList",players);  
  
    //Forward to the jsp page for rendering  
    request.getRequestDispatcher("DisplayPlayers.jsp") .  
forward(request, response);  
} catch (Exception ex) {  
    throw new ServletException(ex);  
} finally {  
    //close the eManager to release any resources held up  
    by the persistence provider  
    if(eManager != null) { eManager.close();} } }
```

Figure 5.5 displays the JSP page with the index page and other pages in the players application.

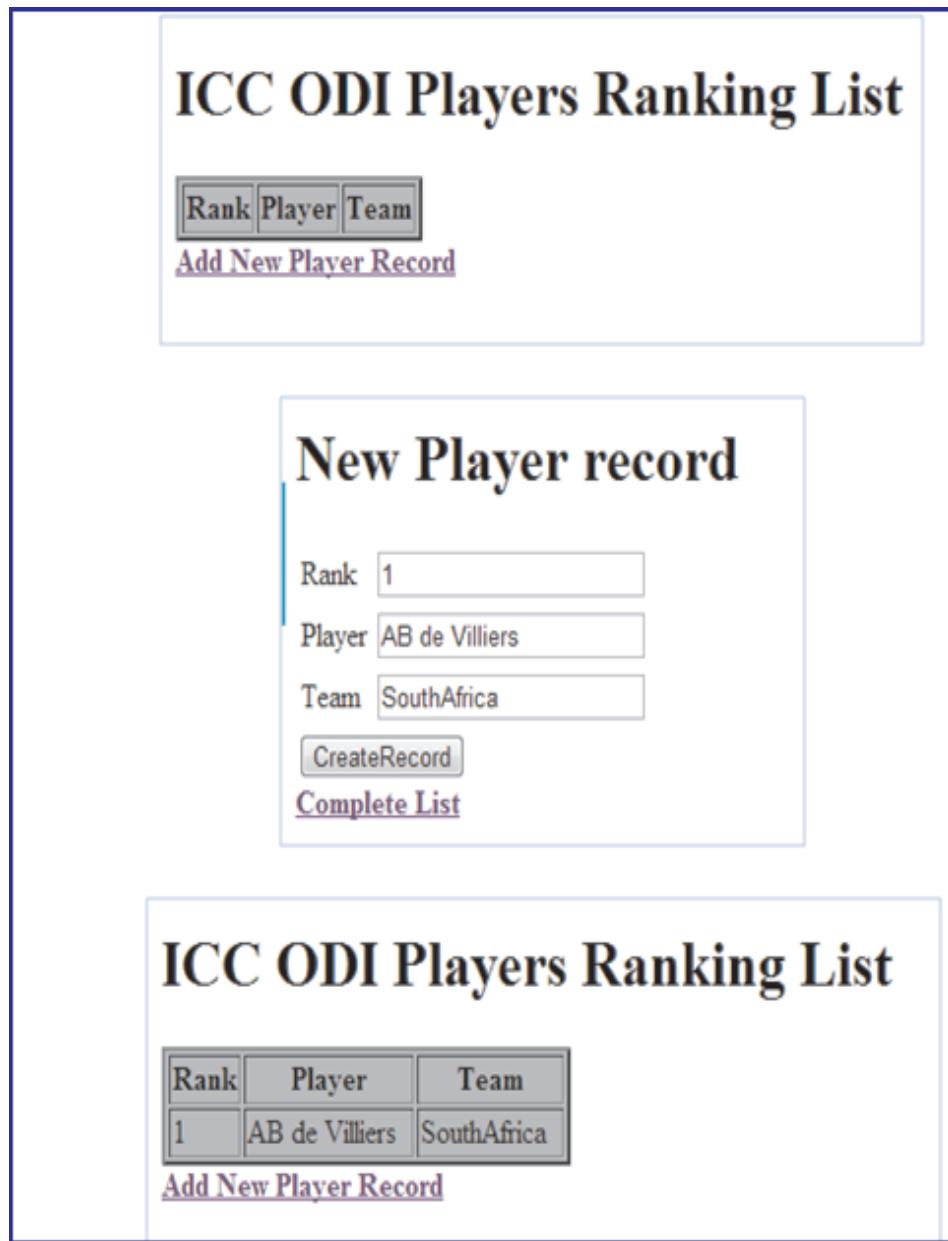


Figure 5.5: Execution of Players Web Application

5.5 DAO Pattern for Databases

The DAO pattern provides the easy maintenance of the applications by separating the business logics from the database access. In DAO pattern, the implementation of data access based on JDBC API is encapsulated in the DAO classes. Thus, providing the independence and increasing flexibility between the layers in the application development.

The business tier in the Web application includes business service classes, domain object classes, and the DAO classes. The business service uses the DAO to perform the data access

functions such as inserting or retrieving an object from the database.

The advantages of DAO pattern are as follows:

- Separation of domain objects and persistence logic.
- Achieving flexibility and reusability in changing the data access.
- Different types of clients such as Servlets or normal Java classes can use the data access logic provided in the DAO classes.
- Provides independence to change the front-end technologies as well as backend database systems.

5.6 Session Event Handling

A session is a collection of HTTP requests, between a client and a Web Server. Session handling is used to maintain the state of a user till the lifetime of the session.

Operating Systems are programmed to run in event driven manner. Interrupt handlers are used as event handlers by them. These are used for hardware events generated by the CPU.

Event driven programs are made up of functions, which listen and handle events. For example, act of clicking a mouse button. In Java EE, the Web container transfer control to event handlers i.e. event listeners to listen events related to Servlets, JSP, and so on.

For example, when you visited an Online Shopping Web site, wherein you added some items to the shopping cart and logged off to pay later. When you again login in the shopping Web site, you'll desire that your cart shows the previously selected items. In such a condition, the server has to keep track of the previous session to show the selected items. Now, you just have to pay for the selected items to get them delivered. This is called session handling or tracking.

5.6.1 Listener Types

The javax.servlet package defines the listener interfaces. These interfaces are of three types as shown in figure 5.6.

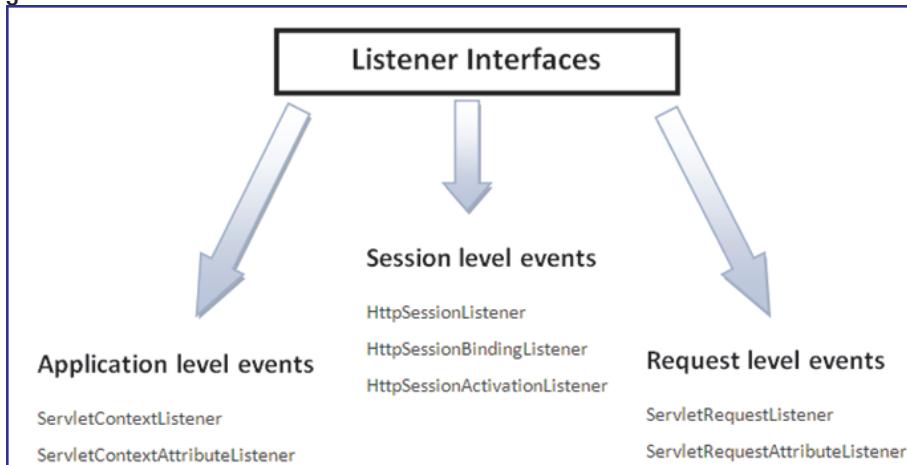


Figure 5.6: Listener Interfaces

5.6.2 Handling Servlet Lifecycle Events

In Web Development, listeners assist by recognizing application events. These are also known as observers or event handlers. This helps developers in controlling life cycle of `ServletContext`, `ServletRequest`, and `HttpSession` objects.

The events in a servlet life cycle is monitored by defining listener objects. These objects methods get invoked when life cycle events occur.

For using listener objects, follow these steps:

- Define the listener class.
- Mention the listener class in `web.xml`, that is, the Web application deployment descriptor.

The servlet context provides access to various resources and facilities, which are common to all Servlets in the application. The Servlet API is used to set the information common to all servlets in an application.

Servlets running in the same server sometimes share resources, such as JSP pages, files, and other servlets. This is required when several servlets are bound together to form a single application, such as a chat application, which creates single chat room for all users.

The Servlet API also knows about resources to the Servlets in the context. The attributes of `ServletContext` class can be used to share information among a group of Servlets.

5.6.3 ServletContextListener

The interface `ServletContextListener` extends `java.util.EventListener` interface. The function of this interface is to receive notifications about changes to the Servlet context of the Web application. The implementation class needs to be configured in the deployment descriptor of the Web application to receive notification about events.

The methods present in the class are as follows:

- `contextInitialized()`

The method returns the notification that a Web application initialization has started.

Syntax:

```
public void contextInitialized(ServletContextEvent  
sce1)
```

where,

- `sce1` is an instance of `ServletContextEvent`

contextDestroyed()

The method returns the notification about closing of the servletcontext. All servlets and filters are closed before notification.

Syntax:

```
public void contextDestroyed(ServletContextEvent sce1)
```

where,

- sce1 is an instance of ServletContextEvent

Code Snippet 11 shows the use of ServletContext methods.

Code Snippet 11:

```
public void contextInitialized(ServletContextEvent event){  
    ServletContext context = event.getServletContext();  
    String IP = "10.1.1.142";  
    context.setAttribute("DefaultAddress", IP);  
}  
  
public void contextDestroyed(ServletContextEvent event){  
    ServletContext context = event.getServletContext();  
    String IP = context.getAttribute("DefaultAddress");  
}
```

5.6.4 ServletContextEvent

The ServletContextEvent class extends the java.util.EventObject class. This event class notifies about changes made to a Servlet context in a Web application. The method present in ServletContextEvent class is as follows:

getServletContext()

The method returns the modified servlet context.

Syntax:

```
public ServletContext getServletContext()
```

Code Snippet 12 shows the use of the method `getServletContext()`.

Code Snippet 12:

```
// Save and get an application-scoped value
getServletContext().setAttribute("app-param", "app-value1");
value1 = getServletContext().getAttribute("app-param");
```

Figure 5.7 depicts the hierarchy for `ServletContextListener`.

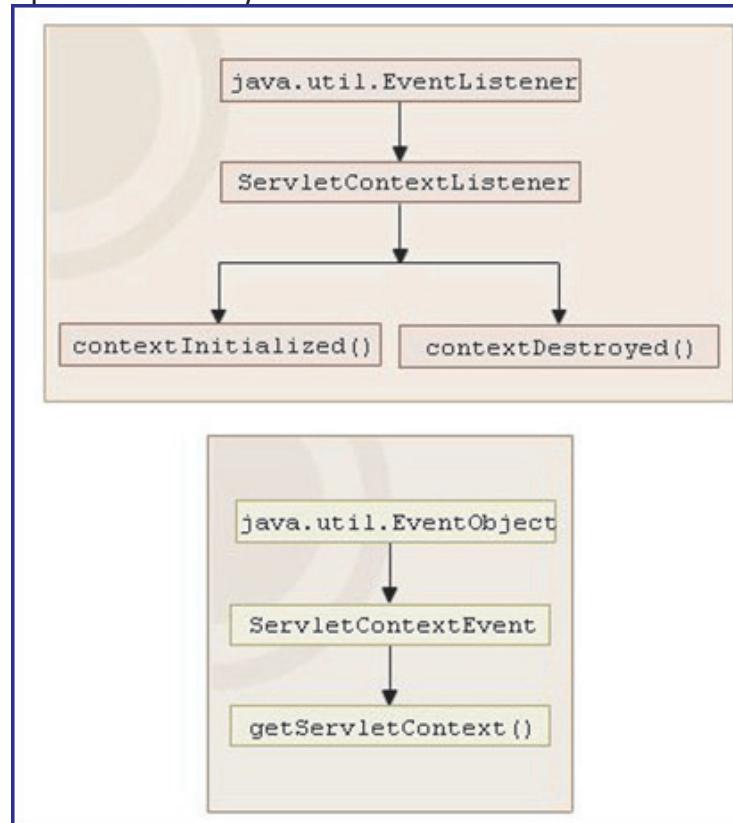


Figure 5.7: ServletContextListener Hierarchy

5.6.5 ServletContextAttributeListener

The `ServletContextAttributeListener` interface extends `java.util.EventListener` interface. The interface receives a notification about any modifications made to the attribute list on the servlet context of a Web application. The implementation class must be configured in the deployment descriptor for the Web application to receive notification events. The methods present in the interface are as follows:

- **attributeAdded()**

The method notifies that a new attribute is added to the `ServletContext`. The method is invoked after an attribute has been added.

Syntax:

```
public void attributeAdded(ServletContextAttributeEvent sca1)
```

where,

- sca1 is an instance of ServletContextAttributeEvent

□ attributeRemoved()

The method notifies that an attribute is removed from ServletContext. The method is invoked after an attribute has been removed.

Syntax:

```
public void attributeRemoved(ServletContextAttributeEvent sca1)
```

where,

- sca1 is an instance of ServletContextAttributeEvent

Code Snippet 13 shows the method of the ServletContextAttributeListener.

Code Snippet 13:

```
public void attributeAdded(ServletContextAttributeEvent event1) {
    log.append(event1.getName() + "," + event.getValue() +
               "," + new Date() + "\n");
}

public void attributeRemoved(ServletContextAttributeEvent event) {
    log.append(event.getName() + "," + event.getValue() + value1
               =      "," + new Date() + "\n");
}
```

□ attributeReplaced()

The method notifies that an attribute of ServletContext has been replaced. The method is invoked after an attribute has been replaced.

Syntax:

```
public void attributeReplaced(ServletContextAttributeEvent sca1)
```

where,

- `sca1` is an instance of `ServletContextAttributeEvent`

The code snippet shows replacement of an attribute.

Figure 5.8 depicts the methods of `ServletContextAttributeListener`.

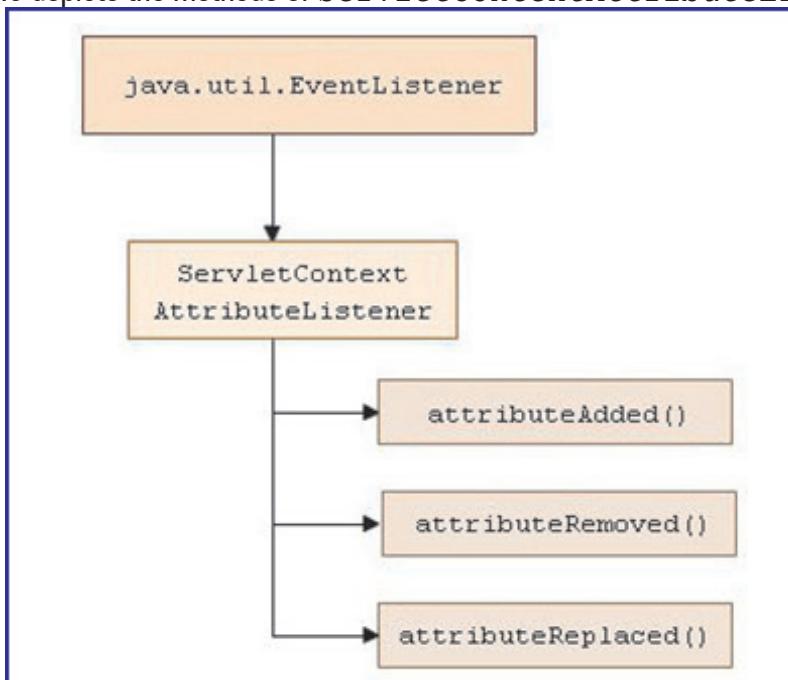


Figure 5.8: Methods of `ServletContextAttributeListener`

5.6.6

ServletContextAttributeEvent

The `ServletContextAttributeEvent` class extends `ServletContextEvent` class. This is the event class for notifications about changes to the attributes of the `ServletContext` of a Web application.

The methods present in the class are as follows:

getName()

The name of the altered attribute in the `ServletContext` is returned by the method.

Syntax:

```
public java.lang.String getName()
```

Code Snippet 14 gets the name of the attribute and checks whether the attribute is altered.

Code Snippet 14:

```
public void attributeRemoved(ServletContextAttributeEvent scael) {
    if (scae1.getName().equals("heading")) {
        ServletContext context1 = scae1.getServletContext();
        String sHeading1=(String) context1.getAttribute("heading");
        context.log("Heading Replaced="+sHeading1);
    }
}
```

□ getValue()

The attribute value that has been added, removed, or replaced is returned by the method.

Syntax:

```
public java.lang.Object getValue()
```

Code Snippet 15 gets the value of the attribute and checks whether the value is not equal to null.

Code Snippet 15:

```
HttpSession session = request.getSession(true);
ShoppingCart1 previousItems1 =
    (ShoppingCart1)session.getValue("previousItems1");
if (previousItems1 != null) {
    doSomethingWith(previousItems1);
} else {
    previousItems1 = new ShoppingCart1(...);
    doSomethingElseWith(previousItems1);
}
```

5.6.7 HttpSessionAttributeListener

`HttpSessionAttributeListener` is an interface that extends the `java.util.EventListener` class. It is called whenever some changes are made to the attribute list on the servlet session of a Web application.

`HttpSessionAttributeListener` is used to notify these changes. The listener is used to notify when an attribute has been added, removed, or replaced by another attribute.

The methods belonging to this interface are as follows:

□ **attributeAdded()**

This method notifies whenever there is an addition of new attributes to a session. It is called after the addition of attribute.

Syntax:

```
public void attributeAdded(HttpSessionBindingEvent se)
```

where,

- se is the object of HttpSessionBindingEvent.

□ **attributeRemoved()**

This method notifies whenever there is a removal of attributes from a session. It is called after the removal of attribute.

Syntax:

```
public void attributeRemoved(HttpSessionBindingEvent se)
```

where,

- se is the object of HttpSessionBindingEvent.

□ **attributeReplaced()**

This method notifies whenever attributes are replaced in a session. It is called after the attribute is replaced.

Syntax:

```
public void attributeReplaced(HttpSessionBindingEvent se)
```

where,

- se is the object of HttpSessionBindingEvent.

Code Snippet 16 implements HttpSessionAttributeListener interface and its methods.

Code Snippet 16:

```
public class UserAttributeListener implements
HttpSessionAttributeListener{
// declaring attributeAdded() method
public void attributeAdded(HttpSessionBindingEvent Demoevent)
{
    // Logging the event details
    HttpSession session = Demoevent.getSession();
    ServletContext sc = session.getServletContext();
    sc.log(Demoevent.getName() + "," + Demoevent.getValue());
}
// Declaring attributeRemoved() method
public void attributeRemoved(HttpSessionBindingEvent Demoevent){}
// Declaring attributeReplaced() method
public void attributeReplaced(HttpSessionBindingEvent Demoevent){}
}
```

5.6.8 HttpSessionBindingListener

The HttpSessionBindingListener notifies the object when it is being bound to or unbound from a session. This notification can be the result of a forced unbinding of an attribute from a session by the programmer, invalidation of the session or due to timing out of the session. The implementation class do not require any configuration within the deployment descriptor of the Web application.

The methods belonging to this interface are as follows:

□ valueBound()

The method notifies the object on being bound to a session and is responsible for identification of the session.

Syntax:

```
public void valueBound(HttpSessionBindingEvent event)
```

where,

- event is the object of HttpSessionBindingEvent.

valueUnBound()

The method notifies the object on being unbound from a session and is responsible for identification of the session.

Syntax:

```
public void valueUnbound.HttpSessionBindingEvent event)
```

where,

- event is the object of HttpSessionBindingEvent.

Code Snippet 17 implements the valueBound and valueUnbound methods of HttpSessionBindingListener interface.

Code Snippet 17:

```
public class UserInfo implements HttpSessionBindingListener{
    private String uName;
    public User(String userName) {
        uName = userName;
    }
    public void valueBound(HttpSessionBindingEvent ev) {
        System.out.println(uName+" user bound");
    }
    public void valueUnbound(HttpSessionBindingEvent ev) {
        System.out.println(uName+" user unbound");
    }
}
```

5.6.9 HttpSessionBindingEvent

The HttpSessionBindingEvent event is sent on occasions to an object that implements HttpSessionBindingListener when being bound to or unbound from the session. It is also sent when HttpSessionAttributeListener is configured during the deployment descriptor while being bound, unbound, or replaced in a session.

The object is bound to a session by a call to HttpSession.setAttribute and unbounded from a session by a call to HttpSession.removeAttribute.

The methods belonging to this class are as follows:

getName()

This method returns a string specifying the name with which the attribute is bound to or unbound from the session.

Syntax:

```
public java.lang.String getName()
```

Code Snippet 18 gets the name string specifying the name with which the attribute is bound to or unbound from the session.

Code Snippet 18:

```
private void checkAttribute(HttpServletRequestBindingEvent event,
String orderAttributeName, String keyItemName, String mes-
sage) {

    String demoCurrentAttributeName = event.getName();
    String demoCurrentItemName = (String)event.getValue();
    if (demoCurrentAttributeName.equals(orderAttributeName)
&&
        demoCurrentItemName.equals(keyItemName)) {
        ServletContext context =
        event.getSession().getServletContext();
        context.log("Customer" + message + keyItemName +
".");
    }
}
```

□ `getSession()`

This method returns the session object that has changed.

Syntax:

```
public HttpSession getSession()
```

Code Snippet 19 returns the session object that has changed.

Code Snippet 19:

```
public void sessionCreated(HttpServletRequestBindingEvent event)
{
    // Retrieves modified session object
    HttpSession session = event.getSession();
    session.getServletContext().log("SessionListener01:
sessionCreated(" +
        session.getId() + ")");
}
```

□ `getValue()`

This method returns the value of the attribute that has been added, removed, or replaced.

Syntax:

```
public java.lang.Object getValue()
```

Code Snippet 20 returns the value of the attribute that has been added, removed, or replaced.

Code Snippet 20:

```
private void checkAttribute(HttpSessionBindingEvent event,
String orderAttributeName, String keyItemName, String
message) {

    String demoCurrentAttributeName = event.getName();
    String demoCurrentItemName = (String)event.getValue();
    if (demoCurrentAttributeName.equals(orderAttributeName)
&&
        demoCurrentItemName.equals(keyItemName)) {
        ServletContext context =
        event.getSession().getServletContext();
        context.log("Customer" + message + keyItemName + ".");
    }
}
```

5.6.10 HttpSessionListener

The changes to the list of active sessions in Web application are notified to the implementations of HttpSessionListener. The deployment descriptor for the Web application must have the configured implementation class to receive the notification events.

Various session events are session creation and destruction. The method belonging to this interface are as follows:

□ **sessionCreated(HttpSessionEvent se)**

This method provides notification that a session was created.

Syntax:

```
public void sessionCreated(HttpSessionEvent se)
```

where,

- se is the object of HttpSessionEvent.

Code Snippet 21 implements HttpSessionListener interface and its methods.

Code Snippet 21:

```
public class SessionTracker extends HttpServlet implements  
HttpSessionListener{  
  
    static private int sessionCount; // count of session  
  
    // Implementing sessionCreated()  
  
    public void sessionCreated(HttpSessionEvent event) {  
  
        sessionCount++;  
    }  
}
```

□ sessionDestroyed(HttpSessionEvent se)

This method provides notification that a session is about to be invalidated.

Syntax:

```
public void sessionDestroyed(HttpSessionEvent se)
```

where,

- se is the object of HttpSessionEvent.

Code Snippet 22 implements the method SessionDestroyed() of the HttpSessionListener interface.

Code Snippet 22:

```
public void sessionDestroyed(HttpSessionEvent se) {  
    HttpSession session = se.getSession();  
    try{  
        session.invalidate();  
    }  
    catch(IllegalStateException e) { }  
}
```

The `SessionTracker` class must be configured in the deployment descriptor of the Web application as shown in Code Snippet 23.

Code Snippet 23:

```
<listener>
    <listener-class> SessionTracker </listener-class>
</listener>
```

5.6.11 HttpSessionActivationListener

Sometimes to manage the size of its working set, idle stateful session bean instance is transferred temporarily to the secondary storage. The transfer from the working set to secondary storage is called instance passivation. The transfer back is called activation.

The objects bounded to a session might listen to container events at the same time notifying them about the probability of session being passivated or activated.

The `HttpSessionActivationListener` is implemented when a container migrates the sessions between virtual machines or persists sessions, if required, to provide notification to all attributes bounded sessions. The implementation class do not require any configuration within the deployment descriptor of the Web application.

The methods present in the listener are as follows:

- **`sessionDidActivate(HttpSessionEvent se)`**

The method provides notification that the session has just been activated.

Syntax:

```
public void sessionCreated(HttpSessionEvent se)
```

where,

- `se` is the object of `HttpSessionEvent`.

- **`sessionWillPassivate(HttpSessionEvent se)`**

The method provide notification that the session is about to be passivated.

Syntax:

```
public void sessionWillPassivate(HttpSessionEvent se)
```

where,

- `se` is the object of `HttpSessionEvent`.

Code Snippet 24 implements HttpSessionActivationListener interface and its methods.

Code Snippet 24:

```
public class SessionTracker implements HttpSessionActivationListener {
    public void sessionWillPassivate(HttpSessionEvent se) {
        System.out.println("session is about to be passivated");
    }
    public void sessionDidActivate(HttpSessionEvent se) {
        System.out.println("session has just been activated");
    }
}
```

5.6.12 HttpSessionEvent

The notifications for the changes to the sessions within a Web application are represented by HttpSessionEvent.

The method belonging to this class is as follows:

- getSession**

This method returns the session that changed within a Web application.

Syntax:

```
public HttpSession getSession()
```

Code Snippet 25 shows the getSession() method.

Code Snippet 25:

```
// returns modified session object//
public HttpSession getSession()
{
    return session;
}
```

5.6.13 Use of Listeners for Handling Events

EventListeners should be executed quickly, because all event listening and drawing methods are executed in the same thread.

While designing the program, one should implement their EventListeners in a class that is not public. For more secure implementation then the implementation can be private.

Syntax:

```
public void actionPerformed(ActionEvent e)
```

Code Snippet 26 shows the implementation of the event handling for the beep button.

Code Snippet 26:

```
public class Beeper implements ActionListener {  
    ...  
    //here initialization:  
    button.addActionListener(this);  
    ...  
    public void actionPerformed(ActionEvent e) {  
        ...//Make a beep sound...  
    }  
}
```

Check Your Progress

1. Which of the following tasks that can be performed using JDBC in Java application?

- | | |
|-----|--|
| (A) | Operate the database transactions. |
| (B) | Request the JDBC connection from the server manager. |
| (C) | Get the results from database. |
| (D) | Leave the JDBC connection open. |

- | | | | |
|-----|------|-----|------|
| (A) | A, B | (C) | B, C |
| (B) | A, C | (D) | A, D |

2. Which type of JDBC driver is the fastest?

- | | |
|-----|--------|
| (A) | Type-1 |
| (B) | Type-2 |
| (C) | Type-3 |
| (D) | Type-4 |

3. For using Listener objects, select the correct option.

- | | |
|-----|--|
| (A) | Define listener class. |
| (B) | Define listener class, and specify it in <code>web.xml</code> (deployment descriptor). |
| (C) | Only specify it in <code>web.xml</code> , no need to define it. |
| (D) | None of these. |

Check Your Progress

4. Which of the following is a correct syntax of the `getAttribute()` method?

- | | |
|-----|---|
| (A) | <code>public Object getAttribute(String name1)</code> |
| (B) | <code>public Object getAttribute(String name1, Object object1)</code> |
| (C) | <code>public Object getAttribute(Object object1)</code> |
| (D) | <code>public Name getAttribute(Object object1)</code> |

5. Match the methods to their corresponding functions.

	Function		Method
(A)	Return the name of the attribute that changed on the ServletContext	(1)	<code>contextInitialized()</code>
(B)	Notifies the beginning of initialization process of Web application	(2)	<code>getAttribute()</code>
(C)	Returns the servlet container attribute name	(3)	<code>getName()</code>
(D)	Returns the value of the attribute that has been added, removed, or replaced	(4)	<code>attributeAdded()</code>
(E)	Notifies the new attribute was added to the ServletContext	(5)	<code>getValue()</code>

(A)	A-2, B-3, C-4, D-5, E-1	(C)	A-4, B-5, C-1, D-2, E-3
(B)	A-3, B-1, C-2, D-5, E-4	(D)	A-5, B-1, C-2, D-3, E-4

Check Your Progress

6. Which method among the following options belongs to HttpSessionAttributeListener?

- | | |
|-----|---|
| (A) | public void setAttribute(java.lang.String name, java.lang.Object value) |
| (B) | public void setMaxInactiveInterval(int interval) |
| (C) | public void attributeAdded(HttpSessionBindingEvent se) |
| (D) | public void removeValue(java.lang.String name) |

7. Which method among the following options belongs to HttpSessionBindingEvent?

- | | |
|-----|---|
| (A) | public HttpSession getSession() |
| (B) | public void valueBound(HttpSessionBindingEvent event) |
| (C) | public boolean isNew() |
| (D) | public void removeValue(java.lang.String name) |

8. Which method among the following options belongs to HttpSessionActivationListener interface?

- | | |
|-----|--|
| (A) | public java.lang.Object getValue() |
| (B) | public void sessionDidActivate(HttpSessionEvent se) |
| (C) | public boolean isNew() |
| (D) | public void attributeRemoved(HttpSessionBindingEvent se) |

Answer

1.	B
2.	B
3.	B
4.	A
5.	D
6.	C
7.	B
8.	B

Summary

- The most common operation performed by the Servlet is storing and retrieving database information.
- Java provides various mechanisms using which data can be accessed from the database using JDBC or JPA API.
- The JDBC API provides classes and interfaces that allow a Web application to access and perform operations on the databases.
- JPA is an ORM technology that persist the entities in the database.
- Java annotations or XML are used to define the mapping of entity to existing database tables.
- In JPA, persistent objects are referred as Entities. Entities are plain old Java objects that are persisted to relational databases or legacy systems.
- A persistence context represents a set of managed entity instances that exist in a particular data store.
- The DAO pattern provides the easy maintenance of the applications by separating the business logics from the database access.
- The events in a servlet life cycle is monitored by defining listener objects. These objects methods get invoked when life cycle events occur.



To enhance your knowledge,

visit **REFERENCES**



www.onlinevarsity.com



Welcome to the Session, **Asynchronous Servlet Communication**.

This session explains in depth about the concept of asynchronous Servlet and asynchronous listeners. The session explains the use of server push mechanism in Servlets and also explains how to develop asynchronous JavaScript client in Web applications. Further, the session explains how to perform the non-blocking Input/Output (I/O) operations in Servlet. Finally, the session concludes with the explanation of upgrading the protocol in the request and response sent by the Servlet.

In this Session, you will learn to:

- Explain the need of Asynchronous Servlet
- Explain how to create Asynchronous Servlet and Asynchronous Listener
- Explain the concept of server push mechanism
- Explain how to create an asynchronous JavaScript client using XMLHttpRequest object
- Explain the need of non-blocking I/O support in Servlet
- Explain how to implement non-blocking I/O in asynchronous Servlet
- Explain the need for protocol upgrade
- Explain the process of protocol upgrade in a Servlet

6.1 Implicit Objects

The architecture of the Servlet is based on multithreaded model. This means the Web container creates Servlet threads to process the client requests. Each client will receive its dedicated thread which will serve the request and generate a response to be sent back to the client. The Servlet multithreaded model works fine when the clients request are less for a Servlet. The Web container normally creates a pool of threads ready to serve the client with the Servlet instances.

Consider a situation where the Web server receives multiple client requests for the Servlet. To serve multiple requests, the Web container needs large number of threads in the container pool. A thread associated with the client request may not process the request all the time and may sit idle. This can lead to thread starvation on the server.

Some of the reasons, where the thread can go idle while processing the request are as follows:

- Servlet may take a long running task like making a database connection call for query execution or invoking a remote Web service. In these situations, the thread needs to wait for the resource availability, so that it can generate the response to be sent to the client.
- Servlet may have to wait for some dependent event to proceed for the response generation. The situation can occur when waiting to receive information from another client to proceed with the response generation.

While the threads are engaged in serving clients, the maximum threshold limit to hold threads in the pool may hit. This can affect the Web server scalability, which means either the server will run out of memory or will not remain with the threads in the container pool.

Servlet 3.0 API has provided a new feature in its specification to process the request asynchronously. With Asynchronous processing, one can avoid blocking requests by allowing the thread to perform other operations, while the input is returned to the client.

6.1.1 Asynchronous Processing in Servlets

The Java Servlet API provides asynchronous processing support for servlets and filters in the Web applications. In asynchronous processing, a Servlet thread waiting for a resource is released by the container and returned to the pool. The long running task performed by the thread is carried out asynchronously in the background.

Asynchronous processing is performed in the background thread. Once the task is completed, the background thread can generate a response or a request which needs to be dispatched to a Servlet for processing.

To handle the generated result, a notification is sent to the Servlet container. It will allocate a different Servlet thread available in the pool to handle the result and interact with the client with the generated response.

Following are the steps to be followed for processing the request asynchronously in the Servlet. These are as follows:

1. The request sent to the Servlet is intercepted by the filters which perform some pre-processing tasks for the Servlet. For example, authentication of the user can be pre-processed within a Servlet.
2. The Servlet receive the request parameters to process them.
3. The Servlet requests for some external connection or data. For example, sends a requests for obtaining a connection with a database. If there are already some requests waiting for the connection, then the Servlet joins a waiting queue for a JDBC connection.
4. While waiting for the JDBC connection, the Servlet returns back to the pool without generating a response.
5. The JDBC connection operation is assigned to an asynchronous context object which is responsible for handling the request in the background thread.
6. When the JDBC connection is available, the asynchronous context object gets the connection object and notifies the Servlet container for a Servlet thread to handle the connection.

6.2 Handling Asynchronous Servlet

To support asynchronous processing, the separation of request from the generated response needs to be separated. This is achieved by using the class `javax.servlet.AsyncContext`. This class is used to process the request asynchronously within the Servlet.

Apart from creating the object of `AsyncContext`, you need to enable the `Servlet` class with asynchronous processing. This is done by enabling the attribute `asyncSupported` to true in the `@WebServlet` annotation.

6.2.1 AsyncContext Class

The `AsyncContext` class provide methods that can be used to obtain the instance of the `AsyncContext` within the `Servlet service()` method.

Table 6.1 lists the methods of the `AsyncContext` class.

Method	Description
<code>void start(Runnable run)</code>	This method gets a thread from the managed thread pool to run the specified Runnable thread.
<code>ServletRequest getRequest()</code>	This method obtains the parameters from the request for the asynchronous context.
<code>ServletResponse getResponse()</code>	This method is used inside the asynchronous context to write to the response with the results of the blocking operation.
<code>void complete()</code>	This method completes the asynchronous operation and closes the response object associated with the asynchronous object.
<code>void dispatch(String path)</code>	This method dispatches the request and response objects to the URL to be forwarded to another Servlet thread.

Table 6.1: Methods of `AsyncContext` Class

Following are the steps to be performed to handle asynchronous request in the asynchronous Servlet:

- Obtain the object of the `AsyncContext` class by invoking the `startAsync()` method on the `ServletRequest` object.
- Invoke the `asyncContext.start()` method by passing a `Runnable` `thread` reference which is responsible to execute a running task in the background.
- Call `setTimeout()` on the `asyncContext`, after the number of milliseconds the container has to wait for the specified task to complete. In case if there is no timeout setting then the container's default will be used. If the task fails to complete within the specified time, it throws an exception.
- Call `asyncContext.complete()` or `asyncContext.dispatch()` from the Runnable thread to compete the task or forward the request and response object to some other Servlet.

Code Snippet 1 shows the skeleton for obtaining the asynchronous context within the Servlet `doGet()` or `doPost()` method.

Code Snippet 1:

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet {

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

        Final AsyncContext asyncContext = request.startAsync ();
        asyncContext.setTimeout (...);

        asyncContext.start (new Runnable () {
            @Override
            public void run () { // long running task
                ...
            }
        })
    }
}
```

In the code, the Servlet class is annotated with the attribute `asyncSupported = true` which means the Servlet can handle the request and response **asynchronously**.

Further, to processing asynchronous requests, the object of the `AsyncContext` class is obtained by calling the method `startAsync()` on the request object of the Servlet. This means that the response will be generated only when the blocked operation is either completed or the request is dispatched to the other Servlet by the `asyncContext` object.

Then, the statement, `asyncContext.start (new Runnable () { ... })` obtains the new thread from the container pool. The new thread can access the asynchronous context object and process the request and response.

Code Snippet 2 demonstrates the code that processes the request asynchronously in the Servlet.

Code Snippet 2:

```
...  
@WebServlet(name = "AsyncDispatchServlet", urlPatterns = { "/  
asyncDispatch" }, asyncSupported = true)  
public class AsyncDispatchServlet extends HttpServlet {  
  
    @Override  
  
    public void doGet(final HttpServletRequest request,  
  
                      HttpServletResponse response) throws ServletException,  
                      IOException {  
  
        final AsyncContext asyncContext = request.startAsync();  
  
        request.setAttribute("mainThread", Thread.currentThread().  
getName());  
  
        asyncContext.setTimeout(5000);  
  
        asyncContext.start(new Runnable() {  
  
            @Override  
  
            public void run() {  
  
                // long-running task  
  
                try {  
  
                    Thread.sleep(3000);  
  
                } catch (InterruptedException e) {  
  
                    ...  
                }  
  
                request.setAttribute("workerThread", Thread.  
currentThread().getName());  
  
                // Dispatch the request object to a  
  
                asyncContext.dispatch("/threadNames.jsp");  
            }  
        });  
    }  
}
```

The servlet in Code Snippet 2 supports asynchronous processing and its long-running task to keep the sleep mode on for three seconds. Servlets `doGet()` method proves that the long-running task is executed in a different thread than the main thread. The `doGet()` method attaches the name of the main thread and that of the worker thread to the `ServletRequest` and dispatches to a `threadNames.jsp` page. The `threadNames.jsp` page displays `mainThread` and `workerThread` thread names in the output.

Code Snippet 3 shows the `threadNames.jsp` page.

Code Snippet 3:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Asynchronous servlet</title>
</head>
<body>
Main thread: ${mainThread}
<br/>
Worker thread: ${workerThread}
</body>
</html>
```

Remember, at the end of task, call the `dispatch()` or `complete()` method on the `asyncContext` object, so that it will not wait until it times out.

Figure 6.1 shows the name of the main thread and the name of the worker thread. As you see both names are different, which proves that the worker thread is different from the main thread.

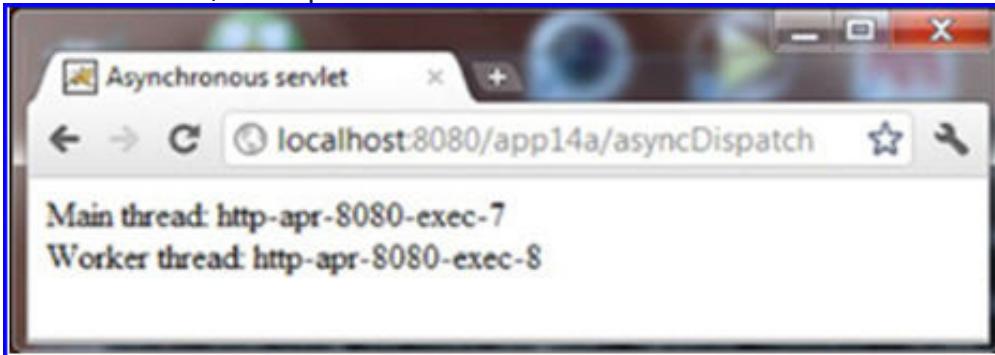


Figure 6.1: Output of AsyncDispatchServlet

Consider a scenario where a servlet sends a progress update every second so that the user can monitor the progress. It sends HTML response and a simple JavaScript code to update the HTML page.

Code Snippet 4 develops an asynchronous Servlet that sends progress updates to HTML page after every second.

Code Snippet 4:

```
package servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class AsyncCompleteServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.println("<html><head><title>" + "Async Servlet</title></head>");
        writer.println("<body><div id='progress'></div>");
        final AsyncContext asyncContext = request.startAsync();
        asyncContext.setTimeout(60000);

        asyncContext.start(new Runnable() {
            @Override
            public void run() {
```

```
System.out.println("new thread:" + Thread.currentThread());  
for (int i = 0; i < 10; i++) {  
    writer.println("<script>");  
    writer.println("document.getElementById(" +  
        "'progress').innerHTML = '" + i * 10) + "% complete'");  
    writer.println("</script>");  
    writer.flush();  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) { . . . }  
}  
writer.println("<script>");  
writer.println("document.getElementById(" +  
    "'progress').innerHTML = 'DONE'");  
writer.println("</script>");  
writer.println("</body></html>");  
asyncContext.complete();  
});  
}  
}
```

In the code, the `asyncContext` object is obtained by invoking the `startAsync()` on the request object. The object creates a thread that creates an HTML page and JavaScript code to update the progress element displayed on the with the value.

Note that the `AsyncCompleteServlet` developed in Code Snippet 4 has not been enabled with the asynchronous behavior. Thus, the Servlet can be annotated either using the `@WebServlet` annotation or configured in the deployment descriptor, `web.xml`.

Code Snippet 5 displays the deployment descriptor, web.xml for configuring the AsyncCompleteServlet.

Code Snippet 5:

```
<web-app>
    ...
    <servlet>
        <servlet-name>AsyncComplete</servlet-name>
        <servlet-class>servlet.AsyncCompleteServlet</servlet-class>
        <async-supported>true</async-supported>
    </servlet>

    <servlet-mapping>
        <servlet-name>AsyncComplete</servlet-name>
        <url-pattern>/asyncComplete</url-pattern>
    </servlet-mapping>

</web-app>
```

Figure 6.2 shows the result of HTML page that receives progress updates from the asynchronous Servlet.

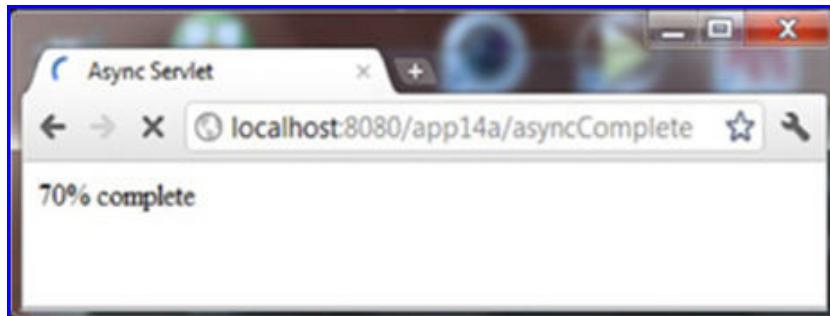


Figure 6.2: Output of AsyncCompleteServlet Servlet

6.2.2 AsyncListener Interface

When the `AsyncContext` object is processing the request and response asynchronously, it passes through a series of events that monitor the life cycle of the `AsyncContext` object.

To handle these events, the `AsyncContext` object can be registered with the `AsyncListener` interface. The `AsyncListener` interface defines the methods that are used to notify the events occurring on the `AsyncContext` object.

You register an `AsyncListener` on an `AsyncContext` object by calling the `addListener()` method on it.

Table 6.2 lists the methods of the `AsyncListener` interface.

Method	Description
<code>void onStartAsync (AsyncEvent event)</code>	This method gets called when an asynchronous operation has been initiated.
<code>void onComplete (AsyncEvent event)</code>	This method gets called when asynchronous operation has completed.
<code>void onError (AsyncEvent event)</code>	This method gets called in the event an asynchronous operation has failed.
<code>void onTimeout (AsyncEvent event)</code>	This method gets called when an asynchronous has timed out, especially, when it failed to finish within the specified timeout.

Table 6.2: Methods of AsyncListener Interface

All the four methods receive an `AsyncEvent` object that gets fired when the asynchronous operation is started, completion of asynchronous processing, error occurrence, and timeout.

Code Snippet 6 demonstrates the code that `AsyncListenerServlet` class that registers the listener to receive event notifications.

Code Snippet 6:

```
package servlet;

import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import listener.MyAsyncListener;

@WebServlet(name = "AsyncListenerServlet", urlPatterns = { "/asyncListener" }, asyncSupported = true)

public class AsyncListenerServlet extends HttpServlet {

    private static final long serialVersionUID = 62738L;

    @Override

    public void doGet(final HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
                      IOException {

        // Obtaining the Asynchronous context object

        final AsyncContext asyncContext = request.startAsync();

        // Setting the time out duration for asyncContext object

        asyncContext.setTimeout(5000);

        // Registering the events

        asyncContext.addListener(new MyAsyncListener());

        // Obtaining a new thread

        asyncContext.start(new Runnable() {

            @Override

            public void run() {

                try {

                    Thread.sleep(3000);

                } catch (InterruptedException e) { ... }

                    String greeting = "hi from listener";

                    System.out.println("wait....");

                    request.setAttribute("greeting", greeting);

                    asyncContext.dispatch("/test.jsp");

                } });

            }

        }

}
```

Code Snippet 7 shows the implementation of the `AsyncListener` interface methods.

Code Snippet 7:

```
public class MyAsyncListener implements AsyncListener {  
  
    @Override  
    public void onComplete(AsyncEvent asyncEvent)  
        throws IOException {  
        System.out.println("onComplete");  
    }  
  
    @Override  
    public void onError(AsyncEvent asyncEvent)  
        throws IOException {  
        System.out.println("onError");  
    }  
  
    @Override  
    public void onStartAsync(AsyncEvent asyncEvent)  
        throws IOException {  
        System.out.println("onStartAsync");  
    }  
  
    @Override  
    public void onTimeout(AsyncEvent asyncEvent)  
        throws IOException {  
        System.out.println("onTimeout");  
    }  
}
```

6.3 Server Push Mechanism in Servlet

The communication between the client and Web server is based on the HTTP protocol. The HTTP protocol implements a strict request-response model, which means that the client sends a HTTP request and waits until the HTTP response is received.

However, as the Web technologies are being updated with the asynchronous behavior, so the need for creating more interactive applications have increased. This has led to the development of modern Web clients that allow the page to interact with the server and update the page without loading the whole page. The mechanism of sending the data asynchronously from the Web server to the client, without redrawing the whole page is referred as server push.

There are many techniques to achieve server push mechanism in Web applications. However, one of the most popular programming approaches used in modern Web applications is Asynchronous Java and XML (AJAX).

Consider a situation where a Web page is called by a user to search for information about musical albums. The page includes a text field which allows the user to enter the name of the album. As soon as the user enters the first character of the search word, the Web application attempts to complete the name by listing all album names whose first or last name begins with the characters entered. This feature of auto-complete the search text is showing the AJAX feature implementation in the Web application.

Using AJAX, the client Web browser uses a JavaScript code that fetches a small amount of data from the Server and updates or integrates the displayed page with the received response from the server, without fetching the entire page.

Figure 6.3 shows the asynchronous communication of Web client and Web server based on AJAX mechanism in the Web application.

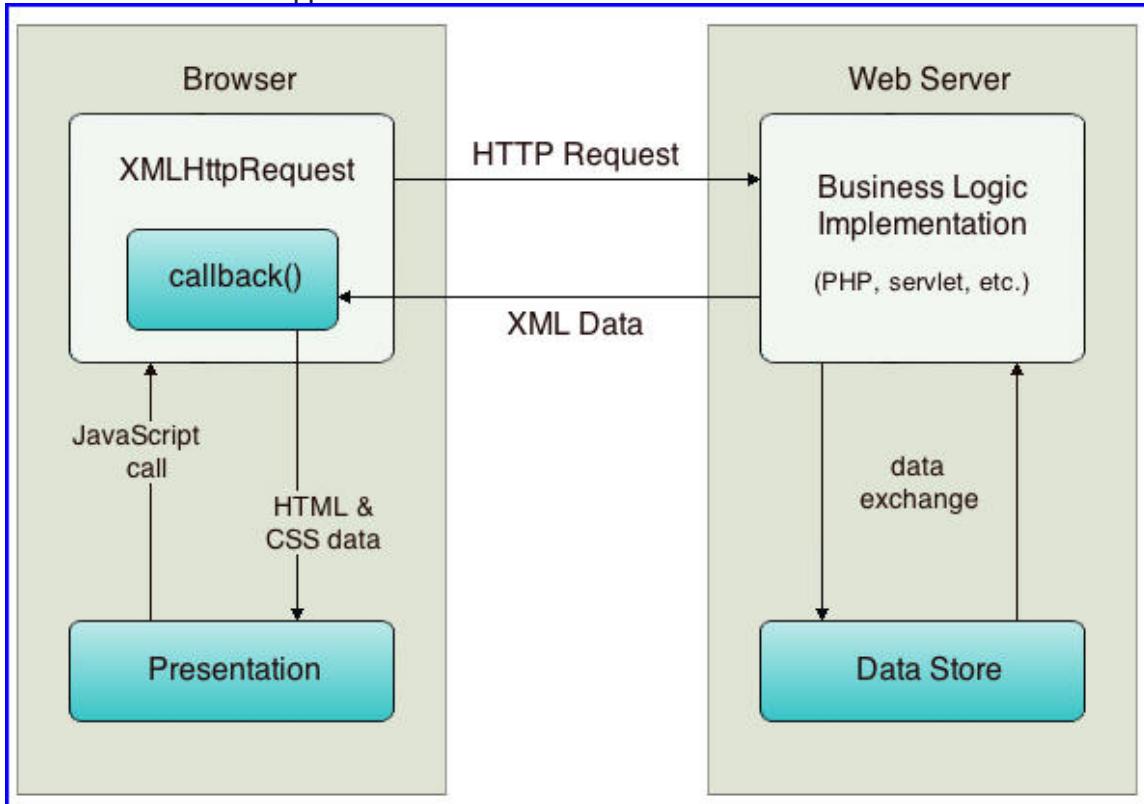


Figure 6.3: Asynchronous Communication in Web Application

As shown in the figure, the Web application uses JavaScript to modify the client's page, whereas the Web server uses the Java Servlet to process the HTTP request asynchronously.

The AJAX mechanism works with an `XMLHttpRequest` object that is used to pass the requests and responses asynchronously between the client and server. All modern Web browsers support the `XMLHttpRequest` object.

6.3.1 Developing Asynchronous Client

The `XMLHttpRequest` object is used in the JavaScript code to interact asynchronously with the Web server.

Following are the steps for creating the asynchronous JavaScript client:

1. Create an `XMLHttpRequest` object.
2. Send the request to the server using the methods of the `XMLHttpRequest` object.
3. Receive and process the response received from the server.

Code Snippet 8 demonstrates the code to check the support for XMLHttpRequest object in the HTML page and process the request and response using the XMLHttpRequest object.

Code Snippet 8:

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
function getXMLHttpRequest() {
    var xmlhttpReq=false;
    // Creating XMLHttpRequest object for non-Microsoft browsers
    if (window.XMLHttpRequest) {
        xmlhttpReq=new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        try {
            // Creating XMLHttpRequest object in later versions
            // of Internet Explorer
            xmlhttpReq=new ActiveXObject("Msxml2.XMLHTTP");
        } catch (exp1) {

            try {
                // to create XMLHttpRequest object in older versions
                // of Internet Explorer
                xmlhttpReq=new ActiveXObject("Microsoft.XMLHTTP");
            } catch (exp2) {
                xmlhttpReq=false;
            }
        }
    }
    return xmlhttpReq;
}
```

```
* AJAX call starts with this function*/  
  
function makeRequest() {  
  
    var XMLHttpRequest = getXMLHttpRequest();  
  
    XMLHttpRequest.onreadystatechange =  
        getReadyStateHandler(XMLHttpRequest);  
  
    XMLHttpRequest.open("GET", "helloWorld.do", true);  
  
    XMLHttpRequest.send();  
  
}  
  
/* Returns a function that waits for the state change in  
XMLHttpRequest*/  
  
function getReadyStateHandler(XMLHttpRequest) {  
  
    // an anonymous function returned  
    // it listens to the XMLHttpRequest instance  
    return function() {  
  
        if (XMLHttpRequest.readyState == 4) {  
  
            if (XMLHttpRequest.status == 200) {  
  
                document.getElementById("hello").innerHTML =  
XMLHttpRequest.responseText;  
  
            } else {  
  
                alert("HTTP error " + XMLHttpRequest.status + ": "  
+ XMLHttpRequest.statusText);  
  
            }  
  
        }  
  
    };  
};
```

The function `getXMLHttpRequest()` checks if the browser supports the `XMLHttpRequest` object. If the object is supported, then create an `XMLHttpRequest` object. For modern browser, the `XMLHttpRequest()` method is used, whereas for the older version of the Internet Explorer browser, the `XMLHttpRequest` object is obtained using `ActiveXObject()` method.

The next step is to send the asynchronous request to the Web server. To send the request, the two methods namely, `open()` and `send()` of `XMLHttpRequest` object are invoked. These methods are used to prepare the HTTP Request and send it to the server.

The syntax of the `open()` is as follows:

Syntax:

```
open(method, url, async)
```

where,

- **method:** Specifies the type of request method, that is, `GET` or `POST` is used to send the request.
- **url:** Specifies the location of the component handling the request on the server.
- **async:** Specifies whether the asynchronous communication is supported or not. The value can be true or false.

Here, the HTTP request `GET` method is used to send the data to the Servlet mapped with the URL, `helloWorld.do`, and the asynchronous variable is set to true. The `send()` method sends the data to the request. In case of `GET` method, the data is sent as the query string, so nothing is sent with the `send()` method. However, if the method is `POST`, then `send()` method will have the data.

Then the JavaScript code will wait for the server response. Once the server is ready with the response, the `onreadystatechange` event is fired which indicates the change in the state. In the `onreadystatechange` event, you can specify what to do when the server response is ready to be processed. When `readyState` is 4 which means server has finished processing request and ready with the response, then check the status is 200, which means response status as OK.

The callback function is invoked and the response is processed to display the result on the page.

6.4 Non-blocking Input/Output in Servlet

Consider a situation when client is submitting a large HTTP POST request over a slow network connection. Using traditional I/O, the container thread associated with this request would be sometimes sitting idle waiting for the rest of the request. In this process, a large part of CPU memory is wasted. Hence, Servlet API provides non-blocking I/O support for Servlets and filters during asynchronous processing of requests.

6.4.1 Implementation of Non-blocking I/O Support

The non-blocking I/O is supported by `javax.servlet.ServletInputStream`, `javax.servlet.ServletOutputStream`, `javax.servlet.ReadListener`, and `javax.servlet.writeListener`.

Following steps are used for non-blocking I/O to process requests and write responses inside service methods:

1. Enable the asynchronous mode for the Servlet using the `@WebServlet` annotation. For example, `@WebServlet(urlPatterns={"/asyncioservlet"}, asyncSupported=true)`
2. Obtain an input stream and/or an output stream from the request and response objects in the `service()` method.
3. Assign a read listener to the input stream and/or a write listener to the output stream.
4. Process the request and the response inside the listener's callback methods.

Code Snippet 9 demonstrates the use of non-blocking I/O in the asynchronous Servlet.

Code Snippet 9:

```
package com.basics;

import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletInputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/asyncioservlet"}, 
asyncSupported=true)

public class FirstServlet extends HttpServlet {

    @Override
    public void doPost(HttpServletRequest request,
HttpServletResponse response) throws IOException {
```

```
final AsyncContext acontextResponse = request.startAsync();
final ServletInputStream inputStream = request.
getInputStream();

inputStream.setReadListener(new ReadListener() {
    byte buffer[] = new byte[4*1024];
    StringBuilder sbuilder = new StringBuilder();
    public void onDataAvailable() {
        try {
            do {
                int length = inputStream.read(buffer);
                sbuilder.append(new String(buffer, 0, length));
            } while(inputStream.isReady());
        } catch (IOException ex) { ex.printStackTrace(); }
    }
    public void onAllDataRead() {
        try {
            acontextResponse.getResponse().getWriter()
                .write("...the response...");
        } catch (IOException ex) { ex.printStackTrace(); }
        acontextResponse.complete();
    }
});
```

Code Snippet 9 enables the servlet with asynchronous processing. The `AsyncContext` object named `acontextResponse` is created to process the request asynchronously. Then, the Servlet obtains the input stream associated with the request and assigns a read listener defined as an inner class. The listener reads parts of the request as they become available and then writes some response to the client when it finishes reading the request.

6.5 Protocol Upgrade Processing

Different client sent requests to server through different protocols. Some client change protocol information in protocol information. With Servlet 3.1 API, the HTTP protocol has been upgraded from HTTP 1.0 to HTTP 1.1. HTTP 1.0 was a stateless protocol, whereas HTTP 1.1 can persist the connection between the client and the server. The advantage of persistent connection is that the connection is alive and can be used for multiple requests with the client.

6.5.1 Upgrading to HTTP 1.1 Protocol

In HTTP/1.1, on a current connection, using the `Upgrade` header field, clients can switch to a different protocol. If the server accepts the request, of switching to the protocol as indicated by the client, an HTTP response with status 101 (switching protocols) is generated. After this, the client and the server communicate using the new protocol.

Code Snippet 10 demonstrates the use of headers to upgrade the protocol information in the response.

Code Snippet 10:

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/ABCDresource"})
public class ABCDUpgradeServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {
        if ("ABCD".equals(request.getHeader("Upgrade"))) {
            /* Accept upgrade request */
            response.setStatus(101);
            response.setHeader("Upgrade", "ABCD");
        } else { /* ... write error response ... */ }
    }
}
```

The class is used to read the request header and upgrade the protocol data.

Code Snippet 11 shows the implementation of `HttpUpgradeHandler`.

Code Snippet 11:

```
package com.authentication;

import javax.servlet.ServletInputStream;
import javax.servlet.ServletOutputStream;

public class ABCDUpgradeHandler implements HttpUpgradeHandler {

    public XYZPUpgradeHandler upgrade(XYZPUpgradeHandler protocol) {
        return protocol;
    }

    public void destroy() { }

}
```

Check Your Progress

1. Which of the following class does not supports the Servlet Non-blocking I/O?

- (A) javax.servlet.ServletInputStream
- (B) javax.servlet.ServletOutputStream
- (C) javax.servlet.ReadListener
- (D) javax.servlet.WritingListener

2. Which of the following object is used for performing asynchronous communication in JavaScript ?

- (A) XMLHttpRequest
- (B) XMLHttpRequest
- (C) XMLHttpRequest
- (D) XMLHttpRequest

3. Which of the following is not AsyncListener method?

- (A) onStartAsync
- (B) onComplete
- (C) onTimeout
- (D) onErrors

4. Which of the following attribute value is set to true for creating a asynchronous Servlet?

- (A) asyncSupported
- (B) async
- (C) syncSupported
- (D) sync

Check Your Progress

5. Which of the following method returns AsyncContext object?

- | | |
|-----|--------------|
| (A) | startAsync() |
| (B) | stopAsync() |
| (C) | Async() |
| (D) | firstAsync() |

Answer

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

Summary

- The architecture of the Servlet is based on multithreaded model. In this model, the Web container normally creates a pool of threads ready to serve the client with the Servlet instance.
- The Java Servlet API provides asynchronous processing support for servlets and filters in the Web applications.
- In asynchronous processing, a Servlet thread waiting for a resource is released by the container and returned to the pool.
- The class `javax.servlet.AsyncContext` is used to process the request asynchronously within the Servlet.
- To handle these events, the `AsyncContext` object can be registered with the `AsyncListener` interface.
- The mechanism of sending the data asynchronously from the Web server to the client, without redrawing the whole page is referred as server push.
- One of the most popular programming approach used in modern Web applications to push the server data to the client is performed using AJAX mechanism.
- The AJAX mechanism works with an `XMLHttpRequest` object that is used to pass the requests and responses asynchronously between the client and server.
- Servlet API provides non-blocking I/O support for Servlets and filters to process input and output asynchronously in the request.
- With Servlet 3.1 API, the HTTP protocol has been upgraded from HTTP 1.0 to HTTP 1.1. HTTP 1.0 was a stateless protocol, whereas HTTP 1.1 can persist the connection between the client and the server.



Welcome to the Session, **JavaServer Pages**.

This session will give an insight to the basics of JavaServer Pages (JSP) and its features. The session explains the life cycle of JSP page managed by the JSP container. Further, the session explains the various scriptlet and directive tags available in a JSP page.

In this Session, you will learn to:

- Explain the need of JSP
- List the benefits of JSP
- Identify the situations where to use JSP and servlets
- Explain JSP architecture
- Identify various phases in the life cycle of a JSP page
- Explain the various scriptlet elements in JSP
- List and explain the use of various directives in JSP

7.1 Introduction

Consider a situation where a developer has to greet the user on his/her successful registration on the page. To accomplish this task, if the Servlet technology is used, the GreetingServlet class is created. The GreetingServlet processes the request and generates a response to be sent to the client.

Figure 7.1 shows the GreetingServlet class.

```
***** GreetingServlet.java *****

import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class GreetingServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        String name = req.getParameter("name");
        String email = req.getParameter("email");
        String message = null;
        GregorianCalendar calendar = new GregorianCalendar();
        if (calendar.get(Calendar.AM_PM) == Calendar.AM)
            message = "Good Morning";
        else
            message = "Good Afternoon";
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<p>" + message + ", " + name + "</p>");
        out.println("<p> Thanks for registering your email(" + email
                  + ") with us.</p>");
        out.println("<p>- The Pro Java Team.</p>");
        out.println("</html>");
        out.println("</body>");
        out.close();
    }
}
```

HTML tags clubbed with Java code

Figure 7.1: Servlet Code

As seen in figure 7.1, to generate the output, the GreetingServlet code uses Java code that will display the string value in the output. Also, the code contains HTML elements embedded in the same Servlet that take care of the presentation of the string on the Web page.

The HTML elements represent the template data displayed on the Web page. However, embedding HTML elements along with the Java code result in the development of rigid applications, which have a tight coupling between presentation and application layer. Also, any modifications in the Servlet results in page re-compilation, which means embedded HTML elements are also re-compiled.

Thus, to segregate the presentation text from the application, Sun laid a new specification referred to as Java Server Pages (JSP). The JSP specification is an extension of the existing Servlet API and leverages all the features of Servlets.

7.1.1 JSP Page

A JSP page is a text document containing static content, JSP tags, and Java code which simplifies the generation of dynamic contents. The static contents are expressed as HTML, XML, or XHTML markup tags and are used to generate the user interfaces on the page. The Java code and JSP elements are used to generate dynamic contents on the Web page.

Thus, the segregation of HTML tags and JSP tags separates the presentation code from the application code.

7.1.2 Application Layers

Figure 7.2 shows the application layers in the Web application.

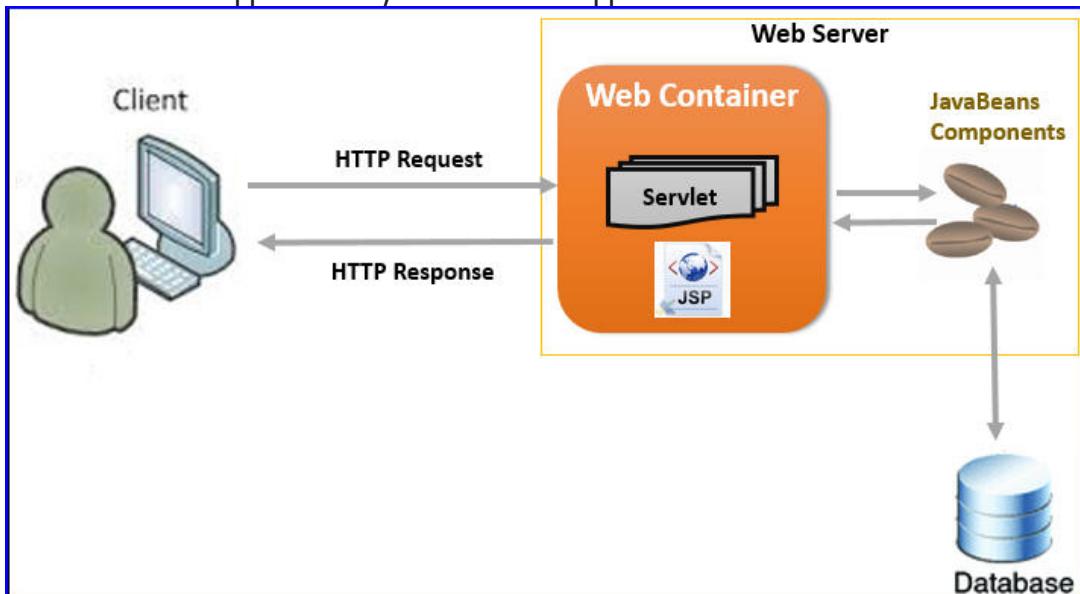


Figure 7.2: Web Application Layers

Most software applications consist of following three layers:

- Presentation layer contains user interface and code to build the interface.
- Application or Business layer contains code to validate data accepted in presentation logic and other task specific code.
- Data access layer contains code to operate on the databases.

In a three-tier architecture, code related to each layer is separated from each other. JSP is used in presentation layer for presenting dynamic content on the Web page.

7.1.3 Benefits of JSP

JSP helps to access server data efficiently and with greater speed. Some of the advantages of using JSP are:

- **Separation of content generation from presentation**

JSP separates content generation from presentation by implementing presentation logic in a JSP page and content logic on server in a JavaBean component.

- **Emphasizing reusable components**

JSP pages use reusable components, such as JavaBeans which can be used by multiple programs. JavaBeans enables to perform complex functions in a JSP page. These beans can be shared and exchanged among the tiers of the Web application.

- **Simplified page development**

JSP allows a Web developer with limited knowledge in scripting language to design a page. Programmers can also generate dynamic content using standard tags provided by JSP.

- **Access and instantiate JavaBean components**

JSP supports the use of JavaBean components with JSP language elements. You can easily create and initialize beans and get and set their values.

7.1.4 Use of Servlet and JSP

JSP technology simplifies the process of creating pages by separating the Web presentation from Web content. In most of the applications, the response sent to the client is a combination of template data and dynamically generated data. In this situation, it is much easier to work with JSP pages than to do everything with servlets.

Servlets are well suited for handling binary data dynamically, for example, for uploading files or for creating dynamic images, since they need not contain any display logic.

7.2 JSP Life Cycle

Every JSP page is translated into a Servlet on a Web server, before execution. The translated Servlet is responsible for handling the request and generating dynamic responses.

7.2.1 Architecture of JSP

Figure 7.3 depicts JSP architecture.

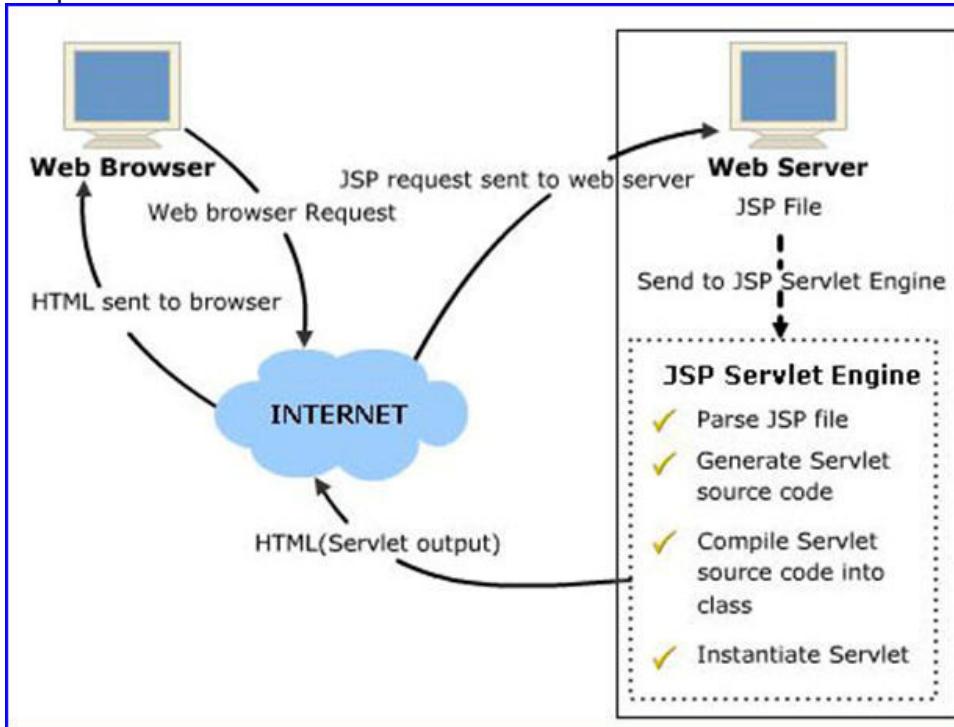


Figure 7.3: JSP Architecture

Whenever the Web browser sends a URL to the Web server requesting a JSP page, the Web engine passes the requested JSP file to the JSP container which parses the JSP file to generate a Servlet source code. This servlet source code is compiled into a .class file and is instantiated by the life cycle methods. The HTML output from the compiled Servlet is sent through the Internet and results are displayed on the user's Web browser.

7.2.2 Life Cycle Phases of JSP

During the process of parsing and compiling a JSP page into the Servlet, the JSP page is passed through two phases which are described as follows:

Translation Phase

Translation is the first phase of JSP life cycle. In this phase, a servlet code to implement JSP tags is automatically generated, compiled, and loaded into the Servlet container. During translation, the JSP engine writes the entire static contents such as markup tags to the response stream in the Servlet. Similarly, the JSP elements are converted into the equivalent Java code in the generated Servlet code.

Figure 7.4 depicts the translation of JSP page into the Servlet class.

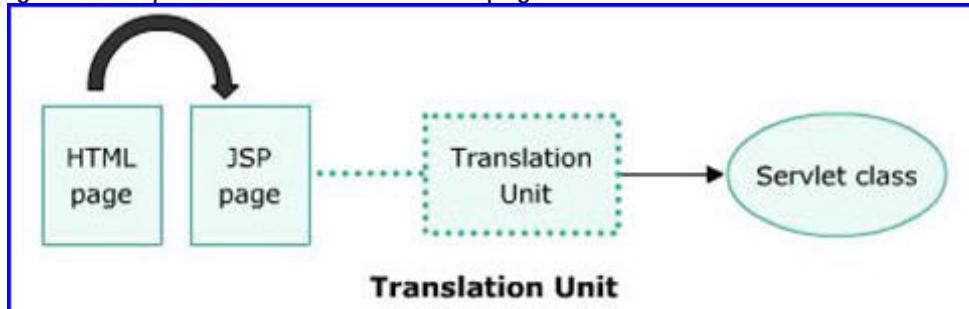


Figure 7.4: Translation Phase

After translation, the Servlet source code is compiled to `.class` file which is a compiled Servlet class. This generated `.class` file is handed to the Servlet container for managing the Servlet life cycle methods.

The process of translation is performed only for the first time invocation of the JSP page. For all subsequent requests, the compiled Servlet is loaded into the Servlet container for execution.

□ Execution Phase

The execution of JSP page is similar to the Servlet's execution. The generated Servlet class must implement the `javax.servlet.jsp.HttpJspPage` interface defined in the JSP API. The `HttpJspPage` interface defines the life cycle methods.

The life cycle methods are executed by the Servlet container on the loaded instance of the converted Servlet. These methods are as follows:

- `jspInit()` – Is similar to `init()` method of Servlet and is invoked on the instantiated Servlet to initialize it with the parameters. A JSP page can override this method by including a definition for it in a declaration element.
- `_jspService()` – Is similar to `service()` method of Servlet and is invoked by the Servlet container at each request. This method is defined automatically by the JSP container and corresponds to the body of the JSP page.
- `jspDestroy()` – Is executed by the Servlet container before destroying the JSP page from the memory.

7.3 Elements of a JSP Page

A JSP page contains HTML tags, which display static content on a Web page. It also contains JSP tags which are used to generate dynamic content. These tags are categorized into standard tags and custom tags.

The standard tags are used to process request, include Java objects and directives, and perform actions on the JSP page. These tags include scripting tags, directive tags, and action tags. All the tags are written within angular brackets `<>`.

Custom tags perform operations, such as processing forms, accessing databases, and other enterprise services, such as e-mail and directories.

7.3.1 Scripting Tags

The JSP tags are used to embed Java code snippets in the JSP page for generating dynamic contents on the Web page. These tags are as follows:

Expressions

JSP expressions can be used to display individual variables or the result of some calculation on a Web page.

A JSP expression contains a Java statement whose value will be evaluated and inserted into the generated Web page. At run time, JSP expression is evaluated and the result is converted into a string. This evaluated string will be displayed on the Web page.

Syntax:

```
<%= Java Expression %>
```

where,

- `<%=` is the opening delimiter for the scriptlet.
- `Java Expression` indicates a valid Java statement.
- `%>` is the closing delimiter.

For example, the expression, `<%= Math.PI%>` looks similar to a markup is a Java expression that will insert the value of PI into the output.

Scriptlets

A JSP scriptlet is used to embed Java code within an HTML code. The Java code is inserted into the `_jspService()` method of the servlet. The scriptlets are executed when the request of the client is being processed. The scriptlets are used to add complex data to an HTML form.

Syntax:

```
<%= Java code fragment %>
```

Figure 7.5 depicts the use of scriptlet in a JSP page.

The screenshot shows a JSP code editor with a blue border. Inside, the JSP code is displayed with syntax highlighting. A red box highlights a section of Java code within a scriptlet tag. The code defines variables 'a' and 'b', calculates their sum, and prints the result. The rest of the JSP structure, including the HTML tags and the main body content, is shown in a light gray background.

```
<HTML>
  <HEAD>
    <TITLE>An Automatic Conversion</TITLE>
  </HEAD>

  <BODY>
    <H1>An Automatic Conversion</H1>
    <%>
      int a;
      int b = 9000;
      a = (b*16)+40000;
      out.println("Value = " + a);
    <%>
  </BODY>
</HTML>
```

Figure 7.5: Using Scriptlet in a JSP Page

□ Declarations

JSP declarations allow the user to define variables and methods that are inserted into the Servlet, outside the `service()` method.

You can declare variables within the scriptlet tag also, but that variable will be local to that scriptlet. However, variable declared using the declaration tag will be visible in the whole JSP page. Also, methods cannot be declared within the scriptlet tag.

A single declaration tag can be used to define multiple variables of same data type.

Syntax:

```
<%! Java declaration code %>
```

Code Snippet 1 demonstrates the use of declaration tag in the JSP page.

Code Snippet 1:

```
<html>
...
<body>
    <h3>Area of a Rectangle</h3>
    <%! int length=20, breadth=30, area=0; %>
    <%! private String units = "cm"; %>

    <% area = length * breadth; %>
    <p> The area of the Rectangle is:
        <%= area %><%= units %>
    </p>
</body>
</html>
```

Figure 7.6 shows the output of Code Snippet 1.



Figure 7.6: Output – Declaration Tag

Comments

JSP comments are used for documenting JSP code which should not be visible to the client when viewing the source of the Web page. These comments are called server-side comments and are encoded within `<%--` and `--%>` symbol.

Syntax:

```
<%-- a JSP comment --%>
```

Example: `<%-- This is the JSP comment in a file --%>`

7.4 Directive Tags

JSP directives control the processing of the entire page. These directives identify the packages to be imported and the interfaces to be implemented. However, these directives do not produce any output. They inform the JSP engine about the actions to be performed on the JSP page.

The overall structure of the Servlet class is affected by JSP directives.

Syntax:

```
<%@ directiveName attribute = "value"%>
```

where,

- directiveName is the name of the directive.
- attribute is attribute of directive.
- value is the value of attribute specified.

You can declare these directives anywhere on the page. However, mostly they are declared at the top of the JSP page.

In JSP, there are three directives:

- page** - The page directive provides instructions that control the structure of the page.
- include** – The include directive includes static contents on the current JSP page.
- taglib** – The taglib directive includes the custom library tags on the JSP page.

7.4.1 page Directive

The **page** directive provides instructions that control the structure of the page. It can be used to import classes from the packages, set the content type for the page, enable or disable page as error page, set the language on the page, and so on.

The page directive defines and manipulates a number of important attributes that affect the entire JSP page. A page directive is written at the beginning of a JSP page.

A JSP page can contain any number of page directives. All directives in the page are processed together during translation and result is applied together to the JSP page.

For example, `<%@ page import="java.util.* , java.io.*" %>` is used to import more than one package in the JSP page. If more than one package is imported, then separate the package names by commas.

Some of the attributes that can be defined for a page are:

- language**

The **language** attribute of the page directive is used to specify the language of the script. The default value of language in a JSP page is Java.

errorPage

A path name to this JSP file sends exceptions. If the pathname begins with a '/', the path is relative to the root directory and is resolved by the Web server. If not, the path name is relative to the current JSP file.

 autoFlush

The buffered output is flushed automatically when the buffer is full. If the autoFlush is set to true, which is default value, the buffer will be flushed. Otherwise, an exception will be thrown when the buffer overflows.

 session

The client must join an HTTP session in order to use the JSP page. If the value is true, the session object refers to the current or new session. The default value for session is true.

Syntax:

```
<%@ page attribute %>
```

where,

- page indicates page directive.
- attribute specifies attribute-value pair of page directive.

The valid parameters are:

```
<%@ page language="java"  
    extends="className"  
    import="className{, +}"  
    session="true|false"  
    buffer="none|sizeInKB"  
    autoFlush="true|false"  
    isThreadSafe="true|false"  
    info="text"  
    errorPage="jspUrl"  
    isErrorPage="true|false"  
    contentType="mimeType{; charset=chars  
et}"%>
```

Code Snippet 2 demonstrates how to set the language on the JSP page by using the `page` directive.

Code Snippet 2:

```
<html>
<%@ page language="Java" %>
<head>
    <title>Testing Page Directive</title>
</head>
<body>
    <h1>Testing Page Directive</h1>
    This page is testing Page Directive.
</body>
</html>
```

7.4.2 include Directive

The `include` directive is used to insert content of another resource in a JSP page at run time. The contents are included physically during page translation. The resource or content can be a file in a text-based format, such as text, HTML, or JSP.

The remote resource should be given with the proper relative file path in the `include` directive.

Syntax:

```
<%@ include file = "Filename" %>
```

where,

- `include` indicates the directive.
- `Filename` specifies the name of a file to include along with relative path.

Figure 7.7 depicts the include directive.

```
<html>
  <head><title>Test for include directive </title></head>
  <body bgcolor="white">
    <font color="red">
      Today's date is :
      <%@ include file="date.jsp" %>
    </font>
  </body>
</html>
```

Figure 7.7: include Directive

7.4.3 taglib Directive

The `taglib` directive allows the JSP page to create custom tags, which are defined by the user. Custom tags have access to all the objects that are available for a JSP page.

A tag library is a group of custom tags that extend the functionality of a JSP page one after the other. The `taglib` directive specifies name of the tag library, which contains compiled Java code for a tag to be used. Using this tag library, the JSP engine determines what action is to be taken when a particular tag appears in a JSP page.

Syntax:

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

where,

- `tagLibraryURI` specifies Uniform Resource Identifier (URI) that identifies the tag library descriptor.
- `prefix` specifies tag name that is used to define a custom tag.
- `tagPrefix` defines the prefix string in `prefix:tagname`.

Figure 7.8 depicts the taglib directive.



The screenshot shows a code editor interface for a JSP page. A red box highlights the first line of the code, which is the taglib directive:

```
<%@ taglib uri="tags" prefix="mt" %>
```

The rest of the page's content is displayed below, enclosed in a blue box:

```
<HTML>
  <HEAD>
    <TITLE>Hello World</TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
    <HR>
    <mt:helloWorld/>
    <HR>
  </BODY>
</HTML>
```

Figure 7.8: taglib Directive

Check Your Progress

1. Identify which of the following statements related to JSP pages are true.

(A)	JSP implements presentation logic in a JSP page and content logic on server in a JavaBean.
(B)	JavaBeans cannot be shared and exchanged among Web programmers.
(C)	By using standard tags, even programmers with limited knowledge in scripting language, can design a JSP page.
(D)	Java Server Pages is used in Application layer.

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

2. Which one is the correct option from the following statements for translation unit?

(A)	A JSP page only includes the contents of servlet pages.
(B)	The translation unit processes the JSP page and converts it to HTML page.
(C)	The translation unit consists of the JSP source file as well as all of its static include files.
(D)	Dynamic include files form a part of the translation unit.

3. Which are the correct options from the following statements for creating JSP expressions?

(A)	JSP expressions are used to display individual variables, or the result of some calculation in a Web page.
(B)	A JSP expression contains a JSP statement whose value will be inserted into the generated Web page.
(C)	At run time, JSP expression is evaluated and the result is converted into a string.
(D)	The expression is used to access the servlet elements in JSP.
(E)	The evaluated string in a JSP expression is directed to the JSP page.

Check Your Progress

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

4. Which one is the correct option from the following statements for creating JSP scriptlets?

(A)	A JSP scriptlet is used to embed Java code within a JSP code.
(B)	The Java code is inserted into the <code>jspService()</code> method of the servlet.
(C)	The scriptlets are executed when the request of the server is being processed.
(D)	The scriptlets are used to add data to a JSP form.

5. Which is the correct syntax of directive from the following options?

(A)	<code><%@ directiveName attribute="value" %></code>
(B)	<code><!@ directiveName attribute="value" !></code>
(C)	<code><%@ directiveName attribute="value" !></code>
(D)	<code><!@ directiveName attribute="value" %></code>

6. Which is the correct option from the following statements for `include` directive?

(A)	The <code>include</code> directive is used to insert content of another resource in a JSP page at run time.
(B)	The remote resource should be given with the file path in the <code>include</code> directive.
(C)	The remote resource is a servlet file in a text-based format.
(D)	The <code>include</code> directive is used for including the page description in a Servlet.

Answer

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

Summary

- ❑ JSP technology is used for developing dynamic Web sites and provides server-side scripting support for creating Web applications.
- ❑ A JSP page is a mixture of standard HTML tags, Web page content, and dynamic content that is specified using JSP elements.
- ❑ The JSP elements are expressions, scriptlets, comments, declarations, directives, and actions.
- ❑ Directives are used to specify the structure of the resulting servlet and actions are JSP tags that transfer control to other server objects or perform operations on other objects.
- ❑ The page directive is used to set the page attributes, such as the language to be used and encoding format.
- ❑ The include directive is used to insert a file referenced in the directive into the JSP page.
- ❑ The taglib directive links the JSP page to an XML document that describes a set of custom JSP tags and determines which tag handler class can implement the action of each tag.



Welcome to the Session, **JSP Implicit Objects**.

This session explains the use of implicit objects in a JSP page. The session provides the explanation on the different types of implicit objects available in JSP.

In this Session, you will learn to:

- Explain the concept of implicit objects in JSP
- List various types of implicit objects in JSP
- Explain how to use the request object
- Explain how to use the response object
- Explain how to use the out object
- Explain how to use the session object
- Explain how to use the application object
- Explain how to use the pageContext object
- Explain how to use the page object
- Explain how to use the config object
- Explain how to use the exception object

8.1 Implicit Objects

Implicit objects are basically a set of Java objects that are available in every JSP page. These objects are created and loaded by the Web container. This means a Web programmer does not need to create implicit objects in a JSP page, as they are automatically available on the page.

The implicit objects are also referred to as pre-defined variables and are accessible within the scripting elements in the JSP page.

8.1.1 Types of Implicit Objects

When the JSP page is translated into a Servlet class, all the implicit objects declarations are taken within the `_jspService()` method. Some of the implicit objects provided by JSP are classified into categories that are as follows:

Input and output objects

The objects control page input and output. These include objects such as `request`, `response`, and `out`. The `request` object controls the data coming into the page. The `response` object controls the information generated as a result of `request` object. The `out` object controls the output data generated as a response to the request.

Scope communication and control objects

The scope communication objects provide access to all the objects available in the given scope. Objects in a JSP application are accessible according to the specified scope. The scope of an object is the section where that object is accessible.

Servlet objects

These objects provide information about the page context. It processes `request` object from a client and sends the `response` object back to the client.

Error object

The object handles error in a JSP page using an implicit object known as `Exception`. You can access this object by declaring your JSP page as an error page. To do so, use the `isErrorPage` attribute of the `page` directive. For example, `<%@page isErrorPage="true"%>`.

8.2 Input and Output Objects

The `input` object represents the input data passed through an HTTP request to the JSP page. Similarly, the `output` object handles the content to be sent to the client in response.

8.2.1 Request Object

The `request` object contains the information sent from the client browser through HTTP protocol.

The information stored in the request object includes the source of the request, the requested URL headers, cookies, and parameters associated with the request.

When the JSP page is translated into the Servlet class, the request object is passed as a reference of the `javax.servlet.http.HttpServletRequest` interface by the container. The scope of the `request` object is page-level which means that the object will be accessible only in the current page.

Table 8.1 lists some of the methods of the request object available on the JSP page for the developers.

Method	Description
<code>public java.lang.String getParameter(java.lang.String name)</code>	This method returns the value of a request parameter as a string, or <code>null</code> if the parameter does not exist. For example, <code>value = request.getParameter("paramName");</code>
<code>public java.lang.String[] getParameterValues(java.lang.String name)</code>	This method returns an array of string objects which contains all the values the given request parameter has or <code>null</code> if the parameter does not exist. For example, <code>String values[] = request.getParameterValues("paramName");</code>
<code>public java.util.Enumeration getParameterNames()</code>	This method returns an enumeration of string objects, which contains the names of the parameters. If the request has no parameters, this method returns an empty enumeration. For example, <code>enumeration names = request.getParameterNames();</code>
<code>public void setAttribute(java.lang.String name, java.lang.Object o)</code>	This method stores an attribute in the request. Attributes are reset between requests. For example, <code>String personName = new String("Grace"); request.setAttribute("name", personName);</code>
<code>public java.lang.Object getAttribute(java.lang.String name)</code>	This method returns the name of the string from the named attribute. For example, <code>String personName = (String) request.getAttribute("name");</code>

Method	Description
public java.lang.String getRemoteHost()	This method returns the name of the client or the last proxy that sends the request. If the hostname is not resolved this method returns the dotted-string form of the IP address. For example, host: <%=request.getRemoteHost()%>
public java.lang.String getRemoteAddr()	This method returns the Internet protocol address of the client or last proxy that sent the request. For example, addr: <%=request.getRemoteAddr()%>

Table 8.1: Methods of request Object

Code Snippet 1 demonstrates the methods of `request` object to receive data from a registration page.

Code Snippet 1:

```
<!-- index.jsp -->
...
<form action="register.jsp">
    First Name:
    <input type="text" name="firstname">
    <br/><br/>
    Favorite Game:
    <input type="checkbox" name="game" value="Cricket">Cricket<BR>
    <input type="checkbox" name="game" value="Hockey">Hockey<BR>
    <input type="checkbox" name="game" value="Baseball">
                                Baseball<BR>
    <input type="checkbox" name="game" value="Badminton">Badminton<BR>
    <br/><br/>
    <input type="submit" value="Submit" name="Submit">
</form>
...
```

```
/* register.jsp */
. . .
<%
    // Retrieves data from text field
    String firstName = request.getParameter("firstname");
    // Retrieves data from check boxes and list box
    String favGame[] = request.getParameterValues("game");
    // Displays the form data
    out.println("Name is: " + "<font color='blue'>" + firstName + "</font>");
    out.println("<br/><br/>Favorite Game: " + "\n");
    for (int i = 0; i < favGame.length; i++) {
        out.println("\t<font color='blue'>" + favGame[i] + "</font>");
    }
%>
<br/><br/>The parameters sent to the Servlet are:<br/>
<font color="blue">
<%
    Enumeration enumParam = request.getParameterNames();
    while (enumParam.hasMoreElements()) {
        String str = (String) enumParam.nextElement();
        out.println(str);
    }
%>
</font>
. . .
```

In the code, the `getParameter()` method is used to retrieve a value from the text field which is a string value. To retrieve multiple selected values from the check box, `getParameterValues()` method is used. This method returns multiple value in an array.

The `getParameterNames()` method is used to retrieve the names of the parameters sent in the request string.

8.2.2 response Object

The `response` implicit object manages the response generated by JSP and uses HTTP protocol to send the response to client. The `response` object is a reference of the `javax.servlet.http.HttpServletResponse` interface.

Table 8.2 lists some of the methods of `response` object to be used in the JSP page.

Method	Description	Example
<code>public void addCookie(Cookie cookie)</code>	The method adds the specified cookie to the response. This method can be called more than once to set more than one cookies.	<code>Cookie MyCookie = new Cookie("RollNumber", "156"); MyCookie.tMaxAge (60*60*24*7*26); response.addCookie (MyCookie);</code>
<code>public void sendRedirect(java.lang.String location) throws java.io.IOException</code>	The method sends a temporary redirect response to the client using the specified redirect location URL. This method can also accept relative URLs.	<code>response.sendRedirect ("myserver.com/thePage.htm?ID=737");</code>
<code>public java.lang.String encodeURL(java.lang.String url)</code>	The method is used to encode the specified URL by including session id. This form of the url can be used in html tags that use a url, such as <code><a href.....></code> . This enables the server to keep track of the session.	<code>response.encodeURL ("buyerPage.jsp");</code>
<code>public java.lang.String encodeRedirectUrl(java.lang.String url)</code>	The method returns a rewritten url that can be used with the <code>sendRedirect</code> method of <code>response</code> object.	<code>if (name == null){ response.sendRedirect(response.encodeURL("homePage.jsp")) }</code>

Table 8.2: Methods of `response` Object

8.2.3 **out Object**

The out object represents the output stream. This output stream will be sent to the client as a response for the request. The out object is an instance of the javax.servlet.jsp.JspWriter class. It uses all standard write(), print(), and println() methods defined in javax.servlet.jsp.JspWriter class to display the output. It has page scope.

Some of the methods of out object are as follows:

- **public abstract void flush()throws java.io.IOException**

This method flushes the buffer. Only the invocation of flush() will flush all the buffers in a chain of writers and output streams. The method may be invoked indirectly if the buffer size is more.

Code Snippet 2 shows how to flush the page content.

Code Snippet 2:

```
out.flush();  
<jsp:forwardpage="login.jsp"/>
```

- **public abstract void clear()throws java.io.IOException**

This method clears the contents of the buffer. If the buffer has been already been flushed then the clear operation will throw an IOException.

Code Snippet 3 shows how to clear the buffer.

Code Snippet 3:

```
// If buffer is not empty, buffer is cleared  
if (out.getBufferSize() != 0)  
    out.clear();
```

- **public abstract void close()throws java.io.IOException**

For the JspWriter, this method need not be invoked explicitly as the code generated by the JSP container will automatically include close().

Code Snippet 4 shows the use of close() method.

Code Snippet 4:

```
out.print("Welcome!!!!");  
out.close();
```

- **public abstract void println(java.lang.String x) throws java.io.IOException**

This method prints a string and then terminates the line. It behaves as the combination of print(String) and then println().

For example, `out.println("The output is:" + ex);`

- **public abstract void print(java.lang.String s) throws java.io.IOException**

This method prints a string. If the argument is null, then the string 'null' is printed.

Code Snippet 5 demonstrates the use of `print()` method.

Code Snippet 5:

```
out.print("Welcome!!!");  
out.print("To Aptech Training");
```

8.3

Scope Communication Objects

Apart from the implicit objects, a JSP page can also access Java objects that are created explicitly. These objects are either created within the scriptlets or through actions. Example of an action can be creation of a JavaBean object on the Web page.

Each created object is associated with a scope attribute which defines where the object is accessible and its lifetime.

8.3.1

session Object

The session object provides all the objects available in the JSP pages within the session. A session expires when the user does not send any request for a long time or when the user closes the Web browser. This object is an instance of `javax.servlet.http.HttpSession` interface. It has session scope.

Some of the methods are as follows:

- **public boolean isNew()**

In Web applications such as shopping cart, when a visitor opens a page, the server needs to check whether a session already exists or not. The `isNew()` method is used for checking whether the session is newly created in this page or not. This will return false if a session already exists and true otherwise.

- **public void setAttribute(java.lang.String name,java.lang.Object value)**

In shopping cart, when a visitor opens a page with the username, the `setAttribute()` method holds the objects in the existing session.

Code Snippet 6 demonstrates how to associate a new session object with the request.

Code Snippet 6:

```
...
<%
if (session.isNew())
{
    UserLogin userLoginObj = new UserLogin(name, password);
    session.setAttribute("loginuser", userLoginObj);
}
%>
...
```

In the code, the `userLoginObj` is an object of the `UserLogin` class which stores session data, such as username and password. The `setAttribute()` method maps the `userLoginObj` to the `loginuser` which acts as a name to access the object.

- **`public java.lang.Object getAttribute(java.lang.String name)`**

The method returns the object bound with the specified name in the current session, otherwise returns null.

Code Snippet 7 shows how to access the named `HttpSession` object.

Code Snippet 7:

```
...
<%
String name = (String) session.getAttribute("loginuser");
%>


|            |                        |
|------------|------------------------|
| User name: | <% out.print(name); %> |
|------------|------------------------|


...
...
```

- **public java.util.Enumeration getAttributeNames()**
The method returns an enumeration of string objects. The string contains the names of all the objects bound to this session.
- **public void invalidate()**
The method invalidates a session and then release the objects bound to it.
Code Snippet 8 shows how to invalidate a session.

Code Snippet 8:

```
<%
// Retrieve the user's session
HttpSession session = request.getSession();
// Invalidate the session
session.invalidate();
%>
```

- **public void removeAttribute(java.lang.String name)**
The method removes the object bounded with the specified name from this session. For example, `session.removeAttribute("loginuser")` ;

8.3.2 application Object

The application object is used to share the data between all application pages. So all users can share the information of a given application using the application object. The application object can be accessed by any JSP present in the application. The application object is an instance of the `javax.servlet.ServletContext` interface. It has application scope.

Some of the methods of application object are:

- **public void setAttribute(java.lang.String name, java.lang.Object object)**

The method binds an object to a given attribute name in this servlet context.

Code Snippet 9 shows how to set an attribute to be accessed in any JSP page in the application.

Code Snippet 9:

```
 . . .
<%
    String Initialized="yes";
    application.setAttribute("init", Initialized);
%>
```

- **public java.lang.Object getAttribute(java.lang.String name)**

The method returns the servlet container attribute with the given name or null if there is no attribute.

- **public java.util.Enumeration getAttributeNames()**

The method returns an enumeration containing the attribute names available within this servlet context.

Code Snippet 10 demonstrates how to retrieve all application objects set in the application.

Code Snippet 10:

```
<%
    String username="john";
    String password="apt001";
    application.setAttribute("username", username);
    application.setAttribute("password", password);
    // Retrieves all attributes
    Enumeration enum=application.getAttributeNames();
    // Print the attributes
    while(enum.hasMoreElements()) {
        String attrName=(String)enum.nextElement();
        out.println(attrName+"<BR>");
    }
%>
```

- **public void removeAttribute(java.lang.String name)**
The method removes the attribute with the given name from the servlet context. For example, `application.removeAttribute("password");`
- **public java.lang.String getServerInfo()**
The method returns the name and version of the servlet container on which the servlet is running. For example, `out.println("Server Information:" + application.getServerInfo());`
- **public void log(java.lang.String msg)**
The method writes the specified message to a servlet log file, usually an event log. For example, `application.log (Hello log Message);`

8.3.3 pageContext Object

The `pageContext` object allows user to access all the implicit objects defined in the `page` scope. The `pageContext` object is an instance of the `javax.servlet.jsp.PageContext` class. It uses methods defined in the `PageContext` class to access implicit objects in a Web page.

The `javax.servlet.jsp.PageContext` class provides following fields that are used by `pageContext` object to find the scope or specify the scope of the objects.

- **PAGE _ SCOPE**
This specifies the scope for attributes stored in the `pageContext` object. This is the default.
- **REQUEST _ SCOPE**
This specifies the scope for attributes stored in the request object. It remains available until current request is complete.
- **SESSION _ SCOPE**
This specifies the scope for attributes stored in the session object. It remains available until `HttpSession` object is invalidated.
- **APPLICATION _ SCOPE**
This specifies the scope for attributes stored in the `application` object. It remains available until it is reclaimed.

The `pageContext` object provides methods to access all attributes defined by implicit object in a page. The `pageContext` object also provide methods to transfer the control from one Web page to another.

The methods are:

- **public abstract void setAttribute(java.lang.String name, java.lang.Object value, int scope)**
The method registers the name and value specified with appropriate scope. If the value passed in is null, this method functions in the same way as `removeAttribute(name, scope)`.

getAttribute()

This method returns the object associated with the name in the page scope, otherwise it returns null. It throws `java.lang.NullPointerException`, if the name is null.

Syntax:

```
public abstract java.lang.Object getAttribute(java.lang.String  
name)
```

where,

- `name` is attribute name to be retrieved.

Code Snippet 11 shows the `pageContext` object used to set and retrieve the object.

Code Snippet 11:

```
<%  
  
String personName = "Cleveland";  
  
pageContext.setAttribute("name", personName, APPLICATION_  
SCOPE);  
  
%>  
  
...  
  
<%  
  
String personName = (String) pageContext.getAttribute("name",  
APPLICATION_SCOPE);  
  
out.print("Person name:" + personName);  
  
%>
```

 **public abstract java.util.Enumeration
getAttributeNamesInScope(int scope)**

This method enumerates all the attributes in a given scope. It throws `java.lang.IllegalArgumentException` if the scope is invalid.

Code Snippet 12 demonstrates how to retrieve all the objects with request scope.

Code Snippet 12:

```
// Prints all attribute names with request scope
<%
    for (java.util.Enumeration e = pageContext.
getAttributeNamesInScope (
    javax.servlet.jsp.PageContext.REQUEST_SCOPE) ;
    e.hasMoreElements () ; ) { %>
    <%= e.nextElement () %><br>
<% } %>
```

- **public abstract void removeAttribute(java.lang.String name)**

This method removes the object reference associated with the given name from all the scopes. It throws `java.lang.NullPointerException`, if the name is null.

Code Snippet 13 demonstrates how to remove the objects from the JSP page.

Code Snippet 13:

```
<%
    out.println("User Name: " + request.getAttribute("Name") +
    "<br/>");

    out.println("Password: " + request.getAttribute("Password") +
    "<br/>");

    pageContext.removeAttribute("Password");
%>
```

8.4

Servlet Objects

Servlet object is a representation of the JSP page.

8.4.1

page Object

The page object is an instance of the `java.lang.Object`. The object is not frequently used by JSP programmers. A page object is an instance of the servlet processing the current request in a JSP page. It has page scope.

Code Snippet 14 demonstrates how to access the methods of the Servlet through page object.

Code Snippet 14:

```
<%  
    out.println(this.getServletInfo());  
    out.println(((Servlet)page).getServletInfo());  
%>
```

As the variable page is of the type Object, to access the Servlet methods, you have to type cast it with the Servlet.

8.4.2 config Object

The config object stores the information of the servlet, which is created during the compilation of a JSP page. The config object is an instance of the javax.servlet.ServletConfig interface.

The interface provides methods to retrieve servlet initialization parameters. The config object represents the configuration of the servlet data where a JSP page is complied. The configuration data is stored in the form of initialization parameters. It has page scope.

Some of the methods are as follows:

- **public java.lang.String getInitParameter(java.lang.String name)**

This method returns a string which contains the value of the named initialization parameter, otherwise it returns null.

- **public java.util.Enumeration getInitParameterNames()**

This method returns the names of the servlet's initialization parameters as an enumeration of string objects. Otherwise it returns an empty enumeration object, if the servlet has no initialization parameters.

Code Snippet 15 demonstrates the method that retrieves all the init parameters using `getInitParameterNames()` and then prints each parameter name using `getInitParameter()`.

Code Snippet 15:

```
public void getParameters(ServletConfig config) throws
ServletException {
    // Retrieves all init parameters
    Enumeration params = config.getInitParameterNames();
    // Prints the init parameter names and values
    while (params.hasMoreElements()) {
        String name = (String) params.nextElement();
        System.out.println("Parameter Name: " + name + " Value: " +
        config.getInitParameter(name));
    }
}
```

□ **public ServletContext getServletContext()**

This method returns a reference to the servlet context.

Code Snippet 16 shows how to set a JavaBean object in the `ServletContext`.

Code Snippet 16:

```
ServletContext context = config.getServletContext();
context.setAttribute("dbBean", dbBean);
```

□ **public java.lang.String getServletName()**

This method returns the name of this servlet instance. It can get the name that may be provided through the server administration or assigned in the Web application deployment descriptor. For example, `String servletName = config.getServletName();`

8.5 error Object

While executing and processing client requests, runtime errors can occur either inside or outside the JSP page. These errors can be processed using the `exception` object present in the JSP page.

When an error occurs, the execution of a JSP page is terminated. The `exception` object is used to handle errors in a JSP page. The `exception` object is used to trace the exception thrown during the execution.

The `exception` object is available to the JSP page that is assigned as an error page. This object is the instance of `java.lang.Throwable` class. It has page scope.

Some of the methods are:

□ **public String getMessage()**

This method returns the descriptive error message associated with the exception when it was thrown.

□ **public void printStackTrace(PrintWriters)**

This method prints the execution stack in effect when the exception was thrown to the designated output stream.

□ **public String toString()**

This method returns a string combining the class name of the exception with its error message.

The `isErrorPage` attribute is set to true, which indicates that the page is an error page. The JSP expression (`<%= %>`) is used to create an instance of the `exception` object.

The `out` object is passed as an argument to the `printStackTrace()` method which will display the stack trace information.

Code Snippet 17 demonstrates how to handle exceptions in the JSP.

Code Snippet 17:

```
<%@page isErrorPage="true" %>

<html>
<head>
    <title>Implicit Object</title>
</head>
<body>
    <h1>Implicit Object: Exception</h1>
    The following error has been detected :<br>
    <b><%=exception%></b><br>
    <%exception.printStackTrace(out);%>
</body>
</html>
```

Check Your Progress

1. Which of the following statements for request object are true?

(A)	The request object uses HTTP protocol to process the request from server.	(C)	The getParameter() method returns the value of a request parameter as a string.
(B)	HTTP protocol implements a subclass of the javax.jsp.HttpServletWriter interface.	(D)	The getParameterValues() method returns a string which contains all of the methods given in request parameter.

2. Which are the correct options for response object from the following statements?

(A)	The response implicit object manages the response generated by JSP and sends response to the client.
(B)	The sendRedirect(url) method sends a response to the server using the specified redirect location URL.
(C)	The addCookie(cookie) method adds the specified cookie to the response.
(D)	The addCookie(cookie) method is unable to set more than one cookies.
(E)	The encodeURL() method is used to encode the specified URL.

3. Which one is the correct option for out object from the following statements?

(A)	The out object is an instance of the javax.JspWriter class.	(C)	The clear() method clear the contents of the buffer and print the string in the same line.
(B)	The flush() method flushes all the buffers in a chain of Writers and OutputStreams.	(D)	The print(text) method behaves as the combination of print(String) and println().

4. Which one from the following options is correct about session object?

(A)	The session object provides all the objects available in the servlet pages.	(C)	The session continues after the user stops sending requests to the Web server.
(B)	A session expires when the user closes the Web browser.	(D)	The session object is an instance of javax.servlet.http.HttpObject interface.

Check Your Progress

5. Which one from the following options is correct about application object?

(A)	The application object is used to share the data only between specified pages.	(C)	The setAttribute() method binds an object to a given attribute name in this servlet context.
(B)	The getAttribute() method returns the servlet attribute and null if there is more than one attribute.	(D)	The getServerInfo() method returns the name of the servlet attribute on which the servlet is running.

6. Which are the correct options for pageContext object from the following statements?

(A)	The pageContext object allows user to access all the implicit objects defined in the page scope.
(B)	The setAttribute() method registers the name and value outside the scope.
(C)	The removeAttribute() method removes the object reference associated with the given name from all scopes.
(D)	The APPLICATION_SCOPE field specifies the scope for attributes stored in the pageContext object.
(E)	The PAGE_SCOPE field specifies the scope for attributes stored in the pageContext object.

7. Which are the correct options about page object from the following statements?

(A)	The page object is an instance of the java.lang.Object.
(B)	The page object is an instance of the javax.servlet.jsp class.
(C)	A page object is an instance of the servlet processing the current request in a JSP page.
(D)	The page objects use methods defined in this class to access implicit objects in a Web page.
(E)	The page object has page scope.

Answer

1.	C
2.	A
3.	A
4.	B
5.	C
6.	A
7.	A

Summary

- JSP implicit objects are a set of Java objects that are created and maintained automatically by the Web container.
- The request object represents the request from the client for a Web page. The response implicit object handles the response generated by JSP and sends response to the client.
- The out object represents the output stream.
- The Web server creates a session object for multiple requests sent by a single user. The application object represents the application to which the JSP page belongs.
- The pageContext object allows user to access all the implicit objects defined in the page scope.
- The page object provides access to all the objects, which are defined in a Web page. The config object stores the information of the servlet.
- When an error occurs the execution of a JSP page is terminated. The exception implicit object is used to trace exceptions that occur in a JSP page.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION





Welcome to the Session, **Standard Actions and JavaBeans**.

This session explains the standard actions used to include dynamic content on the JSP page. The session also explains the use of JavaBean components in JSP. It explains the different actions such as `jsp:useBean`, `jsp:setProperty`, and `jsp:getProperty` to instantiate and access properties from the JavaBean component.

In this Session, you will learn to:

- Explain the concept of standard actions in JSP
- Explain how to use the `<jsp:include>` element
- Explain how to use the `<jsp:forward>` element
- Explain how to use the `<jsp:param>` element
- Explain how to use the `<jsp:plugin>` element
- Explain how to use the `<jsp:fallback>` element
- Explain how to use the `<jsp:text>` element
- Explain the concept of JavaBeans
- Explain how to declare and access JavaBeans components in JSP
- Explain accessing JavaBean properties from scripting elements
- Explain how to access non-string data type properties from scripting elements
- Explain how to access indexed properties from scripting elements

9.1 Introduction

Standard actions are XML like tags. They take the form of an XML tag with a name prefixed `jsp`.

The standard actions are used for:

- Forwarding requests and performing includes in pages.
- Embedding the appropriate HTML on pages.
- Interacting between pages and JavaBeans.
- Providing additional functionality to tag libraries.

9.1.1 Use of Standard Actions

JSP standard actions are performed when a browser requests for a JSP page.

The properties of standard actions in JSP are:

- It uses `<jsp>` prefix.
- The attributes are case sensitive.
- Values in the attributes must be enclosed in double quotes.
- Standard actions can be either an empty or a container tag.

Syntax:

```
<jsp:action_name attribute="value" attribute="value"/>
```

or

```
<jsp:action_name attribute="value" attribute="value">  
    ...  
</jsp:action_name>
```

where,

- `action_name` is action to be used.
- `attribute` is replaced by the attributes of the specified action.

Various standard actions that are available in JSP, namely, `<jsp:include>`, `<jsp:forward>`, `<jsp:param>`, `<jsp:params>`, `<jsp:plugin>`, `<jsp:fallback>`, and `<jsp:text>`.

9.1.2 <jsp:include>

The `<jsp:include>` element gives you choice to include either a static or dynamic file in the current requested JSP page is executed. The results are different for each type of inclusion.

If the file is static, then the content is included in the calling JSP file. In case of dynamic, it acts on a request and sends back a result that is included in the JSP page.

You cannot determine whether a file is static or dynamic from a pathname. When you have no idea whether the file is static or dynamic, it is better to use `<jsp:include>` element, because it handles both types of files.

Syntax:

```
<jsp:include page="weburl" flush="true" />
```

where,

- `page` specifies the relative Web address of the page to be included in the current page.
- `flush` attribute is used to flush out the data stored in the buffer. The value of `flush` attribute is `false` by default.

Code Snippet 1 demonstrates the use of `<jsp:include>` action using `index.jsp` and `printdate.jsp` files to display current date and time on the JSP page.

Code Snippet 1:

```
<!-- index.jsp -->
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Dynamic Content Inclusion</title>
  </head>
  <body>
    <h4><font color="BLUE">Displaying Current Date and Time </font></h4>
    <b>Today is:</b><jsp:include page="printdate.jsp"/>
    <p><i>The Date and Time are displayed as a result of evaluation of
another JSP page.</i></p>
  </body>
</html>
```

```
<!-- printdate.jsp-->
<%@page contentType="text/html" pageEncoding="UTF-8"
import="java.util.*"%>
<html>
    ...
<body>
    <%
        Date today = new Date();
        out.print(today.toString());
    %>
</body>
</html>
```

Figure 9.1 shows the output of Code Snippet 1.



Figure 9.1: Output – jsp:include Action

9.1.3 `<jsp:forward>`

The `<jsp:forward>` element is used to redirect the request object containing the client request from one JSP to another target page. The target page can be an HTML file, another JSP file, or a Servlet.

In the source JSP file, the code after the `<jsp:forward>` element is not processed. To pass parameter names and values to the target file, you can use a `<jsp:param>` clause with `<jsp:forward>` element.

Syntax:

```
<jsp:forward page="url"/>
```

where,

- `page` specifies the relative Web address of the target page.

Code Snippet 2 demonstrates the use of `<jsp:forward>` action using `index.jsp` and `printdate.jsp` files to display current date and time on the next JSP page.

Code Snippet 2:

```
<!-- index.jsp-->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  ...
<body>
  <h4><font color="BLUE">Displaying Current Date and Time </font></h4>
  <!-- Forwards request to the other Web resource -->
  <jsp:forward page="printdate.jsp"/>
  <p><i>The request is forwarded to the next page to display Date and Time.</i></p>
</body>
</html>
```

```
<!-- printdate.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"
import="java.util.*"%>
<html>
  ...
<body>
  <%! Date today = new Date();%>
  <b>Today is:</b><% out.print(today.toString());%>
</body>
</html>
```

Figure 9.2 shows the output of Code Snippet 2.

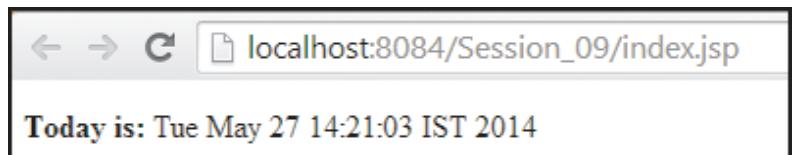


Figure 9.2: Output – `jsp:forward` Action

9.1.4 <jsp:param>

The `<jsp:param>` element allows you to pass one or more name and value pairs as parameters to an included or forwarded resource such as a JSP page, Servlet, or other resource that can process the parameter.

If you want to send more than one parameters to the included or forwarded resource, you can use more than one `<jsp:param>` clause.

Syntax:

```
<jsp:param name="parameterName" value="{parameterValue | <%
expression %>} />
```

where,

- `name` attribute specifies the parameter name and that takes a case-sensitive literal string.
- `value` attribute specifies the parameter value and takes either a case-sensitive literal string or an expression that is evaluated at request time.

Code Snippet 3 demonstrates forwarding request to the page along with the parameter values.

Code Snippet 3:

```
<!-- Product.jsp -->
<html>
  ...
<body>
  <jsp:include page="order.jsp" flush="true">
    <jsp:param name="currency_type" value="Dollar"/>
    <jsp:param name="amount" value="$110"/>
  </jsp:include>
</body>
</html>
```

```
<!-- Order.jsp -->
<body>

<% String currency = request.getParameter("currency_type");
   String amount = request.getParameter("amount");
%>

<b><u>Param Values</u></b> <br/><br/>
<% out.println("Currency: " + currency); %> <br/><br/>
<% out.println("Amount: " + amount); %>

</body>
```

Figure 9.3 shows the output of Code Snippet 3.

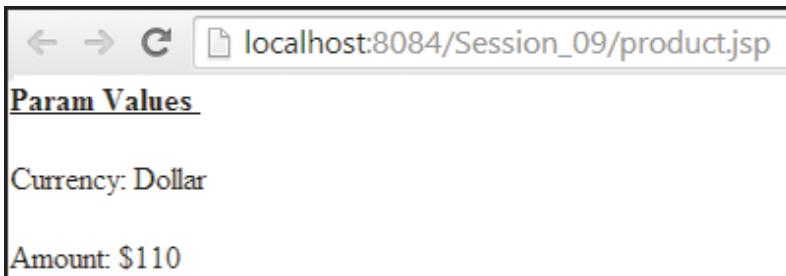


Figure 9.3: Output – `jsp:param` Action

9.1.5 `<jsp:plugin>`

In the client Web browser, the `<jsp:plugin>` element plays or displays an object, using a Java plugin that is available in the browser or downloaded from a specified URL. The plug-in object can be an applet or a JavaBean component that can be displayed on a JSP page.

The `<jsp:plugin>` element is replaced by either an `<object>` or `<embed>` element, according to the browser version when the JSP file is translated and compiled in Java.

The `<jsp:plugin>` element provides attributes that are used for formatting Java object. These attributes are similar to HTML tag attributes. To pass values to the Java objects, the `<jsp:param>` or `<jsp:params>` actions can be used.

Syntax:

```
<jsp:plugin type="bean|applet" code="className" codebase="cl  
assFileDirectoryName" { align="alignment" | archive="archiveList"  
| height="height" | hspace="hspace" | jreversion="jreversion"  
| name="componentName" | vspace="vspace" | width="width" |  
nspluginurl="url" | iepluginurl="url" |  
  
<jsp:params>  
  
{ <jsp:param name="paramName" value="paramValue" /> } +  
  
</jsp:params> }  
  
{ <jsp:fallback>arbitrary-text</jsp:fallback> } >  
  
</jsp:plugin>
```

where,

- Attributes specified between {} are providing configuration data for presenting the component on the Web page.
- **<jsp:params>** element provides parameters to the Java object.
- **<jsp:fallback>** element specifies the content to be used if the plug-in is not existing.

Code Snippet 4 demonstrates the **<jsp:plugin>** element to display an applet on the JSP page.

Code Snippet 4:

```
<jsp:plugin type=applet code="game.class" "codebase="/html">  
  
<jsp:params>  
  
  <jsp:param name="image" value="image/shape.mol" />  
  
</jsp:params>  
  
<jsp:fallback>  
  
  <p>Unable to start the plugin. </p>  
  
</jsp:fallback>  
  
</jsp:plugin>
```

9.1.6 `<jsp:fallback>`

A text message that conveys the user that a plug-in could not start is known as fallback. If the plug-in starts, but the applet or bean does not, the plug-in usually displays a popup window explaining the error to the user.

The `<jsp:fallback>` action specifies an alternative message to the browser, if the browser fails to start the plug-in.

Syntax:

```
<jsp:fallback> text message </jsp:fallback>
```

9.1.7 `<jsp:text>`

A `<jsp:text>` element encloses contents that are displayed within the body of a JSP page or a JSP document. It does not have any sub-element and can appear anywhere in the JSP page where content has to be displayed in the output.

The `<jsp:text>` element can contain expressions which are evaluated and their result is displayed in the output.

Code Snippet 5 demonstrates the use of expression language with `<jsp:text>` element.

Code Snippet 5:

```
<html>
  <body>
    <jsp:text>
      This is the product page.
    </jsp:text>
    . .
    <jsp:text>
      Rectangle Perimeter is: ${2* 10 + 2* 20}
    </jsp:text>
  </body>
</html>
```

9.2

JavaBeans Actions

JavaBeans are reusable components that can be developed in Java. These components define the interactivity of Java objects. JavaBeans allow creation of Java object that can be embedded in multiple applications, servlets, and JSP.

JavaBeans have certain requirements, as they:

- Must be a public class.
- Must have a public constructor with no arguments.
- Must have no public instance variables.
- Must have getter and setter methods to read and write the bean properties.
- Must implement `java.io.Serializable` interface.

9.2.1 Components of JavaBeans

JavaBeans are Java classes that include:

- Properties**

Properties represent bean attributes that can be used to modify the appearance or behavior of the JavaBean.

- Methods**

Methods allow implementing a bean and can be called from other components or from a scripting environment, such as JSP.

- Events**

It allows communication between JavaBeans.

Code Snippet 6 demonstrates the JavaBean component.

Code Snippet 6:

```
import java.io.Serializable;
public class Person implements Serializable
{
    Private String firstName;
    // Constructor
    public Person() { }
    // Sets name
    public void setFirstName(String name)
    {
        firstName=name;  }
    // Retrieves name
    public string getFirstName()
    {
        return firstName;  }
} // End of Person class
```

As seen in the code, the getter and setter methods are public methods defined in a JavaBean class, Person. These methods are also referred as accessor methods and allow retrieving and setting variable values of JavaBean properties.

Get method

This method allows retrieving a variable value.

Syntax:

```
public returnType getPropertyname()
```

where,

- returnType is the data type returned by the get() method.

Set method

This method allows setting or changing the value of a property.

Syntax:

```
public void setPropertyname (datatype parameter)
```

where,

- parameter is the property that will be set.

9.2.2 JavaBeans Components in JSP

JavaBeans components are Java classes which can be reused and composed together as applications. JSP technology directly supports using JavaBeans components with JSP language elements. You can create and initialize JavaBeans and get and set the values for the properties.

JavaBeans component controls access to the private properties of a class by providing public methods.

9.3

Declaring JavaBeans in JSP

The <jsp:useBean> element is used to locate or instantiate a JavaBean component. The <jsp:useBean> first tries to locate an instance of the bean, otherwise it instantiates the bean from a class.

To locate or instantiate the bean, the <jsp:useBean> follows the steps, they are as follows:

1. Attempts to locate a bean within the scope.
2. Defines an object reference variable with the name.

3. Stores a reference to it in the variable, if it retrieves the bean.
4. Instantiates it from the specified class, if it cannot retrieve the bean.

Syntax:

```
<jsp:useBean id="BeanName" class="BeanClass" scope="page|session|application|request"/>
```

where,

- **id** uniquely creates a name for referring the bean.
- **class** specifies the fully qualified classname that defines bean.
- **scope** specifies the scope within which the bean is available. The default scope is **page** if not specified.

Consider that the **Person** JavaBean class is created in the Web application under `com.bean` package. Now, to access the properties of the **Person** class within the JSP page, you need to instantiate it. The `jsp:useBean` action is used to include the bean in the current JSP page.

Code Snippet 7 demonstrates the instantiation of **Person** JavaBean in the JSP page.

Code Snippet 7:

```
<html>
  ...
<body>
  <jsp:useBean id="personID" class="com.bean.Person"
    scope="request"/>
</body>
</html>
```

The code checks if any bean class instance with reference `personID` exists in the request scope, if true then accesses the bean reference. Otherwise, creates a new instance and sets it with request scope.

The java equivalent code for Code Snippet 7 is as follows:

```
Person personID = (Person) request.getAttribute("personID");
if (personID == null)
{
    personID = new Person();
    request.setAttribute("personID", personID);
}
```

9.3.1 Setting Value in JavaBeans

The `<jsp:setProperty>` element sets the value of the properties in a bean using the bean's setter methods. You must declare the JavaBean with `<jsp:useBean>` before you set a property value with `<jsp:setProperty>`.

You can use `<jsp:setProperty>` to set property values in several ways:

- By passing all the values accepted from the user.
- By passing a specific value to set a specific property of the JavaBean.

Syntax:

```
<jsp:setProperty name="BeanAlias" property="PropertyName" value=
"Value" param="Parameter" />
```

where,

- **name** specifies the id of the bean used in `jsp:useBean` action.
- **property** specifies the property name of the bean.
- **value** specifies the explicit value to set for the property.
- **param** specifies the value sent in the request parameter to be assigned to a property.

Code Snippet 8 demonstrates how to set the value for the `firstName` property of `Person` class in JSP.

Code Snippet 8:

```
<html>
  ...
<body>
  <jsp:useBean id="personID" class="com.bean.Person"
    scope="request"/>
  
  <jsp:setProperty name="personID" property="firstName"
    value="John"/>
</body>
</html>
```

In the code, the `jsp:useBean` action is used to include the `Person` class in the current JSP page. Then, the `jsp:setProperty` action is used to set the value, John to the `firstName` property for the bean instance, `personID`.

You can also set the value for the `firstName` by retrieving it from the request parameter sent through HTML form field. Thus, the action tag will be, `<jsp:setProperty name="personID" property="firstName"/>`.

Here, the value of the request parameter `firstName` is retrieved and set in the bean property `firstName`.

However, if the supplied request parameter name is not same as bean property name, then you can use the `param` attribute. For example, if the HTML form field name to accept users' first name is set as `name`, then `jsp:setProperty` action can be written as,

```
<jsp:setProperty name="personID" property="firstName"
param="name"/>.
```

9.3.2 Retrieving Value from JavaBeans

The `<jsp:getProperty>` element retrieves a bean property value using the getter methods and displays the output in a JSP page. Before using `<jsp:getProperty>` element, you must create or locate a bean with `<jsp:useBean>`.

Syntax:

```
<jsp:getProperty name="BeanAlias" property="PropertyName" />
```

where,

- `name` specifies the id of the bean specified in the `jsp:useBean` action.
- `property` specifies the property name from which the value is to be retrieved.

Code Snippet 9 demonstrates how to retrieve value of the `firstName` property of `Person` class in JSP.

Code Snippet 9:

```
<html>
...
<body>
  <jsp:useBean id="personID" class="com.bean.Person"
  scope="request"/>
  Name is: <jsp:getProperty name="personID" property="firstName"/>
</body>
</html>
```

In the code, the `useBean` is used to access the `Person` class instance in the current JSP page. Then, to retrieve the value of the name property from the JavaBean instance, the `jsp:getProperty` action is used.

The equivalent Java code for the `jsp:getProperty` is, `out.print(personID.getFirstName());`

9.4

Accessing JavaBeans within Scriptlets

You can access JavaBeans from scripting element in different ways. The `jsp:getProperty` and expression convert the value into a string and insert it into an implicit `out` object.

If you want to retrieve the value of a property without converting it and insert it into the `out` object, you must have to use a scriptlet. Although, scriptlets are very useful for dynamic processing. However, using custom tags to access object properties and perform flow control is considered to be a better approach.

Code Snippet 10 demonstrates how to create the JavaBean instance and access its properties in JSP.

Code Snippet 10:

```
<html>
  ...
<body>
  // Instantiating Book bean of package pkg using scriptlet
  <% pkg.Book book1 = new pkg.Book(); %>
  // Retrieves the title property using bean getter method
  <%=book1.getTitle()%>
  // Sets title property to a new value
  <% book1.setTitle("Guide to Servlets and JSP"); %>
  ...
</body>
</html>
```

9.4.1 Accessing Non-string Data Type Properties

Bean properties are represented by setter and getter methods. The `jsp:setProperty` action is used to set the properties of a bean by the values of the request parameter. It is inconvenient if the bean's properties are other than string types, because the request parameters are sent as string only.

The JSP container converts the string values into non-string values by the attribute values that evaluate the correct data type to set the property value.

Some of the conversions from string to appropriate data type is done by using Java wrapper classes. For example,

```
String ageStr = request.getParameter("age");  
int ageInt = Integer.valueOf(ageStr);  
  
or  
  
String amtStr = request.getParameter("Amount");  
double amtDouble = Double.valueOf(amtStr);
```

9.4.2 Accessing Indexed Properties

An indexed property is an array of properties or objects that supports a range of values. This property enables the accessor to specify an element of a property to read or write.

If there is an indexed property 'Tomy' of type string, it may be possible from a scripting environment to access an individual indexed value using the index. For example, `b.Tomy[3]` and also to access the same property as an array using `b.Tomy`.

The indexed getter and setter methods throws a runtime exception `java.lang.ArrayIndexOutOfBoundsException` if an index is used that is outside the current array bounds. The value assigned to an indexed property must be an array.

Code Snippet 11 shows how to declare the getter and setter method for the indexed property `studentRegister`.

Code Snippet 11:

```
// Retrieves indexed properties  
public studentRegister[] getStudentRegister()  
  
// Sets indexed properties  
public int setStudentRegister(StudentRegister int [])
```

Check Your Progress

1. Which of the following statements are true regarding JSP standard actions?

(A)	Standard actions take the form of an XML tag with a name suffixed <code>.jsp</code> .
(B)	JSP uses standard actions for calling the JavaBean methods.
(C)	The attributes in standard action cannot distinguish between uppercase and lowercase.
(D)	JSP standard actions are performed when a browser requests for a JSP page.
(E)	In standard action, the values in the attributes must be enclosed in double quotes.

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

2. Which of the following statement is correct for `<jsp:include>` element?

(A)	The <code><jsp:include></code> element is designed to include only a static file in a JSP file.	(C)	In case of dynamic file, it acts on a request and sends back a result that is included in the JSP page.
(B)	In case of static file, the content is excluded from the calling JSP file.	(D)	It is easy to determine whether a file is static or dynamic from a pathname.

3. Match the following JSP actions with their corresponding descriptions.

Action		Description	
(A)	<code><jsp:plugin></code>	(1)	Specifies an alternative message to the browser if the browser fails to start the plug-in.
(B)	<code><jsp:fallback></code>	(2)	Encloses content that has to be displayed in the output.

Check Your Progress

(C)	<jsp:text>	(3)	Allows you to pass one or more name and value pairs as parameters.
(D)	<jsp:param>	(4)	Replaced by either an <object> or <embed> element.

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

4. Which of the following statements are correct for `jsp:useBean` action?

(A)	The <code><jsp:useBean></code> element is used to locate or instantiate a JavaBeans component.
(B)	To locate the bean, it defines an object variable.
(C)	The <code><jsp:useBean></code> first tries to locate an instance of the bean, otherwise it instantiates the bean from a class.
(D)	To locate or instantiate the bean, it search arbitrarily throughout the class.
(E)	To locate or instantiate the bean, the <code><jsp:useBean></code> attempts to locate a bean within the scope.

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

5. Which of the following options is the correct declaration for the `jsp:getProperty` action?

(A)	<code><jsp:getProperty id =“BeanAlias” property=“PropertyName” /></code>
(B)	<code><jsp:getProperty name=“BeanAlias” property=“PropertyName” param=“FormPropertyName”/></code>
(C)	<code><jsp:getProperty name=“BeanAlias” property=“PropertyName” /></code>
(D)	<code><jsp:getProperty id=“BeanAlias” name=“PropertyName” /></code>

Answer

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

Summary

- JSP standard actions are XML like tags that are processed when a browser requests for a JSP page.
- Standard actions in the JSP standard library use the <jsp> prefix. Various standard actions are available that include, <jsp:include>, <jsp:forward>, <jsp:param>, <jsp:params>, <jsp:plugin>, <jsp:fallback>, and <jsp:text>.
- JavaBeans are reusable components that can be developed in Java. These are platform independent and define the interactivity of Java objects in the applications.
- JavaBeans component controls access to the private properties of a class by providing public methods.
- The jsp:useBean action is used to create a reference and include an existing JavaBean component in JSP.
- The jsp:getProperty and jsp:setProperty actions act as getter and setter methods to access and retrieve values from the JavaBean components.
- An indexed property is an array of properties or objects that supports a range of values. This property enables the accessor to specify an element of a property to read or write.



Welcome to the Session, **Model-View-Controller Architecture**.

This session briefly explains the use of JSP models followed in Web application development. It explains the Model 1 architecture with its advantages and disadvantages. Further, the session explains the JSP Model 2 architecture, which is also referred to as Model-View-Controller (MVC) followed by modern Web applications. It explains the components of MVC involved in the Web application development. Finally, the session concludes with the development of Web application based on MVC architecture.

In this Session, you will learn to:

- Describe the use of JSP models in Web applications
- Explain JSP Model 1
- Explain JSP Model 2
- Explain the Model-View-Controller architecture
- Explain the relationship between the components of MVC
- Explain Controller and its purpose in MVC
- Explain View and its purpose in MVC
- Explain Model and its purpose in MVC
- Develop a Web application based on MVC architecture

10.1 Introduction

In most of the Web applications, the user interaction is through forms which are used as an interface for storing and retrieving data from a data store. With time, the approaches to develop such Web applications have changed. For example, in order to improve the application performance and to reduce the Line of Code (LOC), the programmers tend to integrate code for designing and data access in the Web application. This helps to achieve simplicity and flexibility. However, there are significant problems in implementing it. One such problem might be the more frequent change in user interface as compared to the persistence logics leads to the recompilation of whole application.

As a solution, Sun Microsystems provided the JSP specification which addresses the problem of tightly-coupled presentation and business logic in Servlets. Based on the popularity and benefits of JSP in Web development, the commonly used approaches using JSP were identified. These approaches are also referred to as JSP models.

10.2 JSP Models

The JSP specification presents two approaches for developing Web applications using JSP pages. There are two types of programming models for developing Web application:

JSP Model 1

The Model 1 architecture is very simple, the HTML or JSP page sends request along with the data to Web container. The Web container invokes the mapped Servlet which handles all responsibilities for the request. The responsibilities include processing the request, validating data, handling the business logic, and generating a response back to browser.

JSP Model 2

Model 2 or commonly known as Model-View-Controller (MVC), solves many problem of Model 1, by providing a clear separation of application responsibilities. In the MVC architecture, a central servlet, known as the Controller, receives all requests for the application from JSP. The Controller then processes the request, works with the Model to prepare any data needed by the View and forwards the data back to the JSP. In this architecture, the business and presentation logic are separated from each other, which help to reuse the logic.

10.2.1 JSP Model 1 Architecture

The purpose of Model 1 architecture is to enable Web designers to develop Web applications such that it separates business logic from presentation logic.

- Business logic deals with the method of modeling real world business objects such as accounts, loans, travel, and such others in the application. It also deals with the storage mechanism for these objects, object interactions, access, and update rights for them.
- Presentation logic deals with methods of displaying these objects. For example, decisions related to displaying user accounts in a list form.

Figure 10.1 depicts the Model 1 architecture.

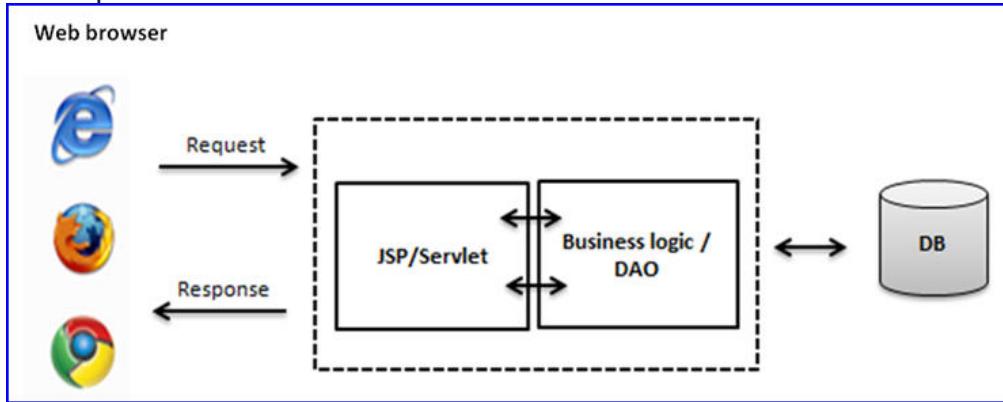


Figure 10.1: Model 1 Architecture

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.

In page-centric architecture, the business process logic and control decisions are hard-coded inside JSP pages in the form of JavaBeans, scriptlets, and expressions.

Figure 10.2 shows an example of JSP Model 1 architecture for an online shopping Web application with all JSP pages.

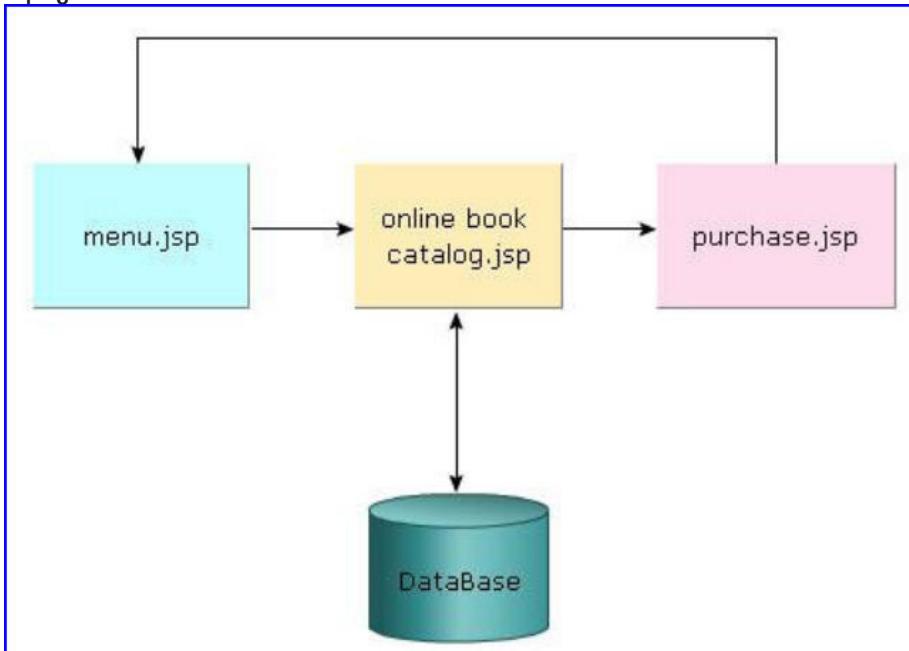


Figure 10.2: JSP Module 1 – Online Shopping Web Application

Advantages of JSP Model 1

The advantage of JSP Model 1 is that it makes development easier as there are no Servlets involved in the application.

Disadvantages of JSP Model 1

- As business logic and presentation logic are tied together, it is difficult for building and maintaining a complex enterprise application.
- Each of the JSP pages is individually responsible for control logic, application logic, and also to present results to the user. This makes the JSP Model 1 more dependent and less extensible.

10.2.2 JSP Model 2 Architecture

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.

Figure 10.3 shows the Model 2 architecture.

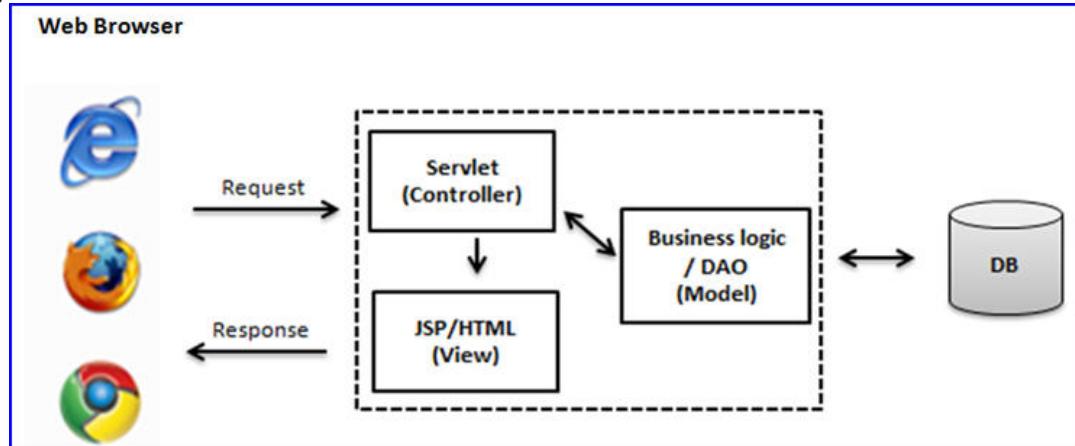


Figure 10.3: Model 2 Architecture

Here, a Servlet acts as a Controller. Thus, Model 2 has a Servlet-centric architecture. 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 handling the View component, it retrieves the objects created by the Servlet, and extracts dynamic content for insertion within a template for display.

Advantages of Model 2

- Web applications based on this model are easier to maintain and extendable, as business and presentation logic are separated from each other.
- Testing is easy in model 2.

10.3 Model-View-Controller (MVC)

MVC is a software architectural pattern. This pattern divides the application logic from User Interface. The division permits independent development, testing, and maintenance of each component.

Figure 10.4 shows a brief overview of the components under the MVC architecture.

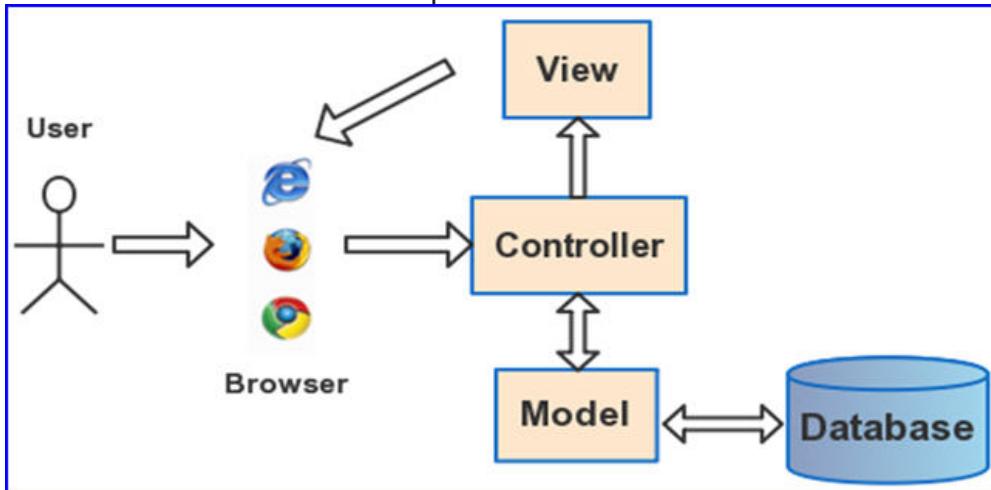


Figure 10.4: Components in MVC Architecture

This model divides the Web based application into three layers:

- **Controller:** At the core of the MVC architecture are the Controller components. The Controller is typically a servlet that receives requests from the browser (or any other source) and manages the flow of data between the Model layer and the View layer. The Controller processes the user requests. It processes the user request and based on the action, the Controller calls methods in the Model to fulfil the requested action and after the action has been taken on the data in model, The Controller is responsible for redirecting the appropriate view to the user.
- **View:** The View component is used to generate the response to the browser, what the user sees. Often the view components are simple JSPs or HTML pages.
- **Model:** The Model component contains the business logics and functions that manipulate the business data. It is a layer between Controller and the database. The Controller can access the functionalities encapsulated in the Model.



The MVC pattern has been around since 1979. Trygve Reenskaug, an engineer who was working on a language called Smalltalk at XEROX, first described it. He named this concept as 'Thing Model View Editor'. The MVC model can be found in UI toolkits such as Nokia's Qt, Apple's Cocoa, Java Swing, and MFC library.

10.3.1 Relationships between Components

The different relationships between MVC components are as follows:

View and Controller Relationship

In this relationship, the Controller is responsible for creating and selecting views.

Model and View Relationship

In this relationship, the view depends on the model. If a change is made to the model then there might be a requirement to make parallel changes in the view.

Model and Controller Relationship

In this relationship, the Controller is dependent on the model. If a change is made to the model interface then there might be a requirement to make parallel changes to the Controller.

10.3.2 MVC in Web Applications

While developing a Web application using MVC architecture the different components and their roles are as follows:

- Model encapsulates data and business logic using JavaBean components or Plain Old Java Object (POJO), a database API, or an XML file.
- View shows the current state of the Model using an HTML or a JSP page.
- Controller updates the state of the Model and generates one or more views using Servlet.

10.3.3 Implementation of MVC Pattern

Consider a scenario where in a Web application you have to develop login service which will validate login details and accordingly display the appropriate JSP page. To design such service, we will use the MVC pattern.

Code Snippet 1 develops the JSP pages required to handle the presentation and result pages.

Code Snippet 1:

```
<!-- login.jsp-->
<html>
    <head>
        <title>MVC Example</title>
    </head>
    <body>
        <form action="LoginController" method="post">
            Enter username : <input type="text" name="username"><br>
            Enter password : <input type="password" name="password">
        <br>
            <input type="submit" />
        </form>
    </body>
</html>
```

```
<!-- success.jsp -->
success.jsp
<html>
    <head>
        <title>Success</title>
    </head>
    <body>
        Welcome, you have successfully login.
    </body>
</html>
```

```
<!-- error.jsp -->
<html>
    <head>
        <title>Error</title>
    </head>
    <body>
        Login failed, please try again.
    </body>
</html>
```

The code shows the designing of three JSP pages namely, login.jsp, success.jsp, and error.jsp. The login page displays the form to the user for filling the user name and password.

The success page will be displayed when the validation of the details is successful.

Otherwise, the error page is displayed to the user.

Next, we will design a Controller that takes request from the user, this is the Servlet class. The servlet class calls the Model which is JavaBean.

Code Snippet 2 demonstrates the design of the Controller named LoginController.java.

Code Snippet 2:

```
package com.mvc.Controller;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.mvc.model.LoginModel;

public class LoginController extends HttpServlet {
    RequestDispatcher rd=null
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    // Receive the values from the request parameters
    String username=request.getParameter("username");
    String password=request.getParameter("password");

    // Instantiate the LoginModel JavaBean
    LoginModel login=new LoginModel();

    // Verify the login credentials from model
    String result=login.authenticate(username, password);

    // Dispatch the control based on the value of the result variable
    if (result.equals("success")) {
        rd=request.getRequestDispatcher("/success.jsp");
    } else {
        rd=request.getRequestDispatcher("/error.jsp");
    }
    // Forward the response to appropriate JSP
    rd.forward(request, response);
}
}
```

In the code, the user details sent from the login.jsp page are received through `request.getParameter()` method. Then the values are sent to the JavaBean named LoginModel which will validate the values and return the appropriate string. Finally, based on the obtained string, the request to appropriate JSP page is sent.

Code Snippet 3 demonstrates the design of LoginModel class.

Code Snippet 3:

```
package com.mvc.model;
public class LoginModel {
    public String authenticate(String username, String password) {

        // Validate the login credentials with the values
        if (("username".equalsIgnoreCase(username))
            && ("password".equals(password))) {
            return "success";
        } else {
            return "failure";
        }
    }
}
```

Code Snippet 4 shows the deployment descriptor, web.xml used to configure the servlet in the Login application.

Code Snippet 4:

```
<!-- web.xml -->
<?xml version="1.0" encoding="UTF-8"?>
...
<servlet>
    <servlet-name>LoginController</servlet-name>
    <servlet-class>com.mvc.Controller.LoginController</servlet-
class>
</servlet>

<servlet-m apping>
    <servlet-name>LoginController</servlet-name>
    <url-pattern>/LoginController</url-pattern>
</servlet-m apping>
```

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

The web.xml is placed in WEB-INF folder and describes the deployment of an application to run servlet and JSP pages. The login.jsp page is the default page to open on sending request to the Web application.

Figure 10.5 shows the request to the URL localhost:8080/TestServlet. This displays the login.jsp with the username and password fields.

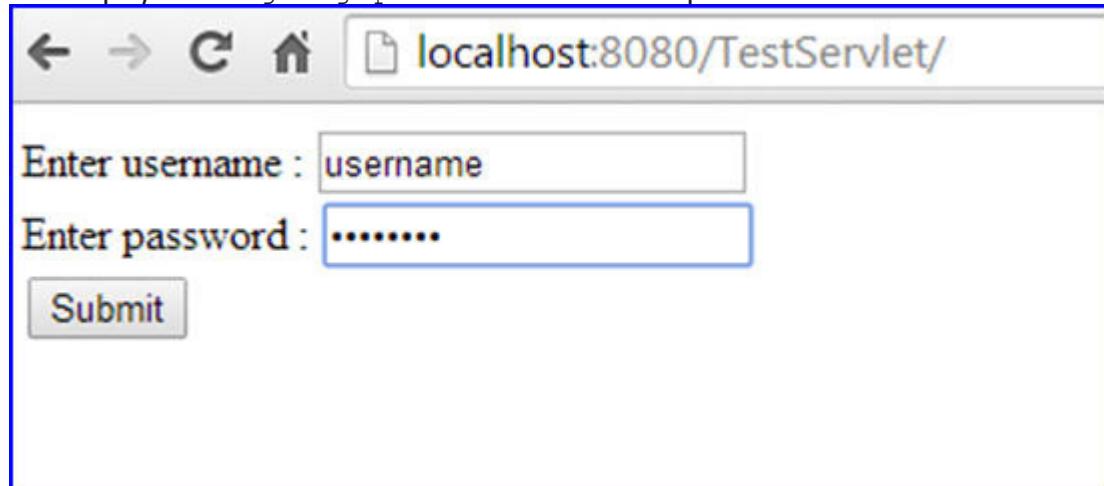


Figure 10.5: login.jsp

As shown in figure 10.5, the **username** and **password** values are sent in request to the Controller, that is, LoginController servlet which invokes the JavaBean method for validating user entered details. As the entered details are correct, the Controller dispatches the request to **success.jsp** page to display the response in the browser.

Figure 10.6 shows the success.jsp.

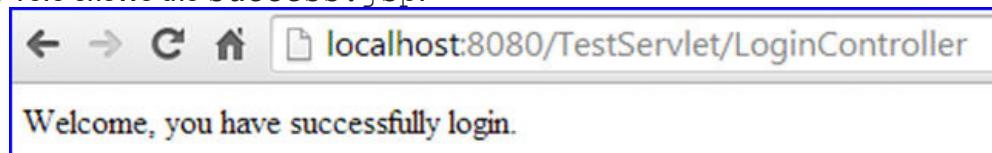


Figure 10.6: success.jsp

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 are correct and true for Controller component?

(A)	The Controller is typically a JSP that receives requests from the browser
(B)	There can be many number of controllers in MVC architecture
(C)	There is only one controller in MVC architecture based on JSP Model 2
(D)	Controller class access the database through View component

3. Match the columns.

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

(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 MVC component displays the data in browser pages?

(A)	Controller	(C)	Model
(B)	View	(D)	Configuration file

Check Your Progress

5. Which of the following MVC component handle the request and responses?

(A)	Controller	(C)	Model
(B)	View	(D)	Configuration file

6. Which of the following MVC component handles the business logic?

(A)	Controller	(C)	Model
(B)	View	(D)	Configuration file

Answer

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

Summary

- The JSP specification presents two approaches for developing Web applications namely, JSP Model I and JSP Model II.
- JSP Model II is also known as MVC.
- MVC is a software design pattern, which can be used to design medium and large sized applications.
- MVC has three components as follows:
 - Model
 - View
 - Controller
- In MVC Web application, Servlet acts as controller, which receives the request from client.
- The view handles presentation of the content on the Web page and could be an HTML file or a JSP file.
- The Model component contains the business logics and functions that manipulate the business data.



Welcome to the Session, **JSP Expression Language**.

The session explains the Expression Language (EL) supported in Java. It explains different implicit objects that can be expressed in EL language. The session also explains how to work with the functions, operators, and tag libraries using EL language in the JSP page.

In this Session, you will learn to:

- Explain how to use script expressions in JSP
- Describe the implicit objects used in EL
- Describe the various operators used in EL
- Explain how to create static method and tag library descriptor using EL
- Explain how to modify deployment descriptor using EL
- Explain how to access EL functions within JSP
- Explain the concept of boxing and unboxing
- Explain how to coerce a value to string or number type

11.1 Expression Language

Expression Language (EL) is a primary feature of the JSP technology. The JSP Standard Tag Library (JSTL) expert group and JSP 2.0 expert group at the Java Community Process developed the JSP expression language.

EL is simple and robust. It can handle both expressions and literals, which are constants and are assigned some memory location. The advantage of EL is that it provides cleaner syntax and is specifically designed for JSP. EL makes it possible to easily access application data stored in JavaBeans components.

EL is a great help to the page authors in accessing and manipulating the application data without mastering the complexities of the programming language, such as Java and JavaScript. JSP expression language can be used to display the generated dynamic content in a table on a Web page. In addition, EL can also be used in HTML tags.

Syntax:

```
$ {EL expression}
```

where,

- \$ indicates the beginning of an expression in EL.
- { is the opening delimiter.
- EL expression specifies the expression.
- } is the closing delimiter.

Code Snippet 1 demonstrates the use of the expression language on JSP.

Code Snippet 1:

```
<!-- result.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Result</title>
  </head>
  <body>
    <h1>Qualifying Exam Criteria:</h1>
```

```
<br />
<br />
<p>Student should atleast score:</p>
<pre>
    Subject Marks
    Maths  ${40+30}
    Java   ${40+35}
    C++    ${40+35}
    Database ${40+35}
</pre>
<hr />
<p>You're accessing the Website on: ${header['user-agent']} p>
</body>
</html>
```

Figure 11.1 shows output where EL expression prints qualifying marks and user agent.

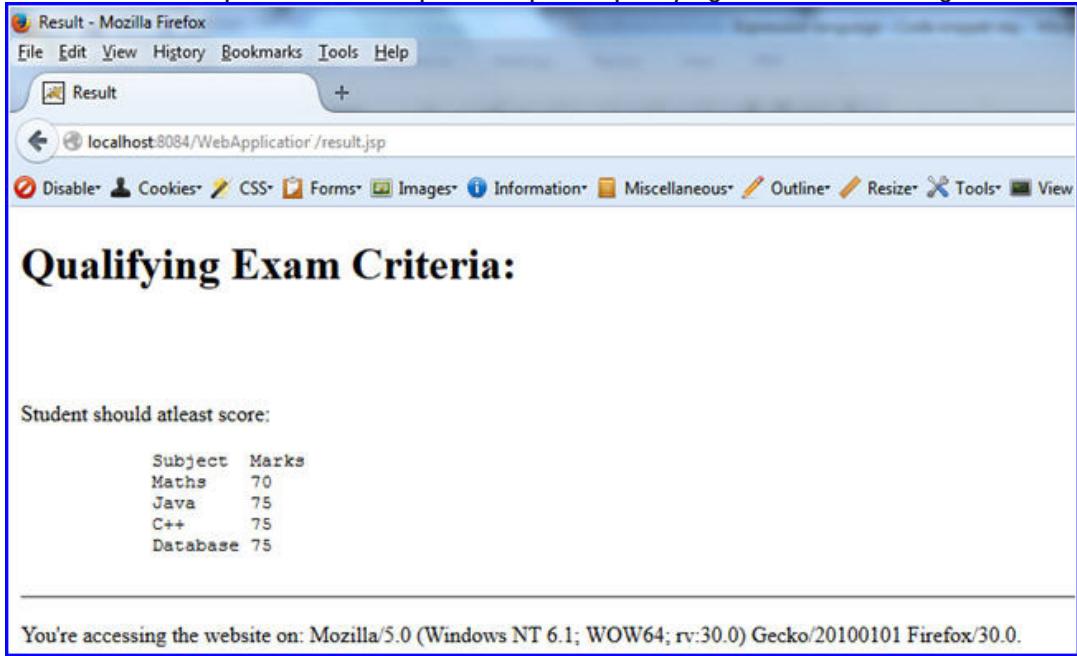


Figure 11.1: Expression Language

11.2 EL Implicit Objects

JSP implicit objects are a standard set of classes. JSP container provides the methods and variables in the classes to the user. EL expressions access implicit objects by name. The user creates an instance of an implicit object to use available methods and variables.

Implicit objects defined in JSP EL are as follows:

- **pageContext**

The `pageContext` object can be used without creating an instance of the object. The `pageContext` object provides access to page attributes. For example, `${pageContext.request}` will return the request object of `pageContext`. This request object can be used to access different page attributes.

- **servletContext**

The `servletContext` object specifies the JSP page, servlet, and Web components contained in the same application.

- **session**

The `session` object represents the session created for the client sending a request.

- **request**

The `request` object represents the request accepted by the JSP page from client. Information of the request is accessed using `request` object.

11.2.1 Request Headers and Parameters

Several implicit objects are available for the easy access to the following objects:

- **param**

The `param` returns a value that maps a request parameter name to a single string value.

Code Snippet 2 demonstrates the use of EL to read parameters.

Code Snippet 2:

```
<!-- If the request parameter name is null or an empty string, this  
snippet returns true. -->  
${empty param.Name}
```

- **paramvalues**

The `paramValues` returns an array of values, which is mapped to the request parameters from client.

Code Snippet 3 demonstrates the use of EL to read parameter values from an array.

Code Snippet 3:

```
<!-- Takes the address line as the input -->
<br>Address Line 1: ${paramValues.address[0]}
<br>Address Line 2: ${paramValues.address[1]}
```

header

The `header` returns a request header name and maps the value to single string value.

Code Snippet 4 demonstrates the use of `header`.

Code Snippet 4:

```
<!--returns the host as a header name -->
${header["host"]}
```

headerValues

The `headerValues` returns an array of values that is mapped to the request header.

Example: `${headerValues.name}`

cookie

The `cookie` returns the cookie name mapped to a single cookie object.

Example: `${cookie.name.value}`

initParam

The `initParam` returns a context initialization parameter name, which is mapped to a single value

11.2.2 Scoped Variables

The JSP API provides the facility of storing data and retrieving them from four different scopes within the JSP container. The EL enhances this facility by further supporting the retrieval of the stored objects as scoped variables. The term **scoped variable** means that the variable is confined to the mentioned context only.

For example, for a specific request, the object will be retrieved only during the processing of the page in question. If the object is stored as session scope, then it will be retrieved by any page during the single interactive session with the Web application.

The four scopes for implicit objects in JSP are as follows:

□ **pageScope**

The `pageScope` returns page-scoped variable names, which are mapped to their values. The `pageScope` is accessible from the JSP page that creates the object.

Example: To access the page-scoped attribute, `${pageScope.book}`

□ **requestScope**

The `requestScope` provides access to the attributes of request object. The `requestScope` object returns requestscoped variable names, which are mapped to their values. The `requestScope` is accessible from Web components handling a request that belongs to the session.

Example: To retrieve the value of the request attribute, `${requestScope.student.name}`

□ **sessionScope**

The `sessionScope` returns session-scoped variable names, which are mapped to their values. The `sessionScope` is accessible from Web components handling a request that belongs to the session.

Example: To retrieve the value of the book object stored in the session, `${sessionScope.book.numberOfPages}`

□ **applicationScope**

The `applicationScope` returns application-scoped variable and maps the variable name to their values.

Example: To check the value of application object, `${applicationScope.booklist == null}`

11.2.3 Page Context

The `pageContext` implicit object defines the context for the JSP page. It provides access to page attributes.

The `pageContext` object provides Web page information using the following objects:

□ **servletContext**

The `servletContext` object specifies the servlet of the JSP page and the Web components contained in the same application. It provides methods that a servlet uses to communicate with its servlet container.

□ **session**

The `session` object represents the session created for the client to send requests. It is helpful in maintaining the client state where series of requests are inter-related.

request

The `request` object represents the request accepted by the JSP page from a client and information of the request.

response

The `response` object represents the response sent to the client by a JSP page. The response contains the data passed between a client and servlet.

11.3 EL Operators

EL helps in easy access of application data stored in JavaBeans components.

Table 11.1 shows all operators used by the JSP EL.

Category	Operators
Variable	<code>.</code> and <code>[]</code>
Arithmetic	<code>+, - (binary)</code> , <code>*</code> , <code>/</code> and <code>div</code> , <code>%</code> and <code>mod</code> , <code>- (unary)</code>
Conditional	<code>A ? B : C</code>
Relational	<code>==</code> , <code>eq</code> , <code>!=</code> , <code>ne</code> , <code><</code> , <code>lt</code> , <code>></code> , <code>gt</code> , <code><=</code> , <code>le</code> , <code>>=</code> , <code>ge</code>
Logical	<code>and</code> , <code>&&</code> , <code>or</code> , <code> </code> , <code>not</code> , <code>!</code>
Empty/Null checking	<code>empty</code>

Table 11.1: Operators Used by the JSP EL

11.3.1 EL Arithmetic Operators

Operators are used to perform different arithmetic, relational, and logical operations. Dot operator `(.)` or `[]` is used to access value of a variable.

An arithmetic statement written in JSP EL may contain more than one operator. The EL supports the following arithmetic operators:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` and `div` (Division)
- `%` and `mod` (Modulo division)

The EL arithmetic operators accept strings that can be converted into numbers as parameters. Thus, the expression `${"2"+"2"}` will evaluate to output 4.

Code Snippet 5 demonstrates the use of EL with arithmetic operators.

Code Snippet 5:

```
<%-- The following code illustrates how you can use the math operators  
of the EL --%>  
  
<html>  
  
    ${2+2}  
  
    ${3-1}  
  
    ${2 * 2}  
  
    ${10/2}  
  
    ${10 div 2}  
  
    ${10 % 9}  
  
    ${10 mod 9}  
  
</html>
```

The output of this code is as follows:

4 2 4 5.0 5.0 1 1

11.3.2 EL Relational Operators

By using various relational operators, comparisons can be made against other values, such as boolean, string, integer, or floating point literals. Various relational operators used in Expression Language are as follows:

- == Equal to
- != Inequality
- < Lesser than
- > Greater than
- <= Lesser than or equal to
- >= Greater than or equal to

Code Snippet 6 demonstrates the EL with relational operators.

Code Snippet 6:

```
<!– comparing numbers -->  
  
4 > '3' ${4 > '3'} <br/>  
  
'4' > 3 ${'4' > 3} <br/>
```

```
'4' > '3' ${ '4' > '3' } <br/>
4 >= 3 ${ 4 >= 3 }<br/>
4 <= 3 ${ 4 < 3 }<br/>
4 == '4' ${ 4 == 4 }<br/>
```

11.3.3 EL Logical Operators

The logical operators supported by Expression language are as follows:

- and, && (Logical AND)
- or, || (Logical OR)
- ! or not (Boolean complement)

Code Snippet 7 demonstrates the use of EL with logical operators.

Code Snippet 7:

```
<!--Test the condition-->
<c:if test="${ (guess >= 10) && (guess <= 20) }">
    <b>You're in the range!</b><br/>
</c:if>
<c:if test="${ (guess < 10) || (guess > 20) }">
    <b>Try again!</b><br/>
</c:if>
```

11.3.4 EL Empty Operators

Whether the value is null or empty can be determined using the empty operator, which is a prefix operation.

The empty operator returns true if the string is empty. The string is said to be empty if it contains no character. If the string is not empty, then empty operator returns false.

Code Snippet 8 demonstrates the use of empty operators.

Code Snippet 8:

```
<% This code returns true if the string is empty %>
<b>
empty "" ${empty ""}<br/>
empty "sometext" ${empty "sometext"}<br/>
</b>
```

11.3.5 EL Dot Operators

The Dot (.) operator is used to access attribute values of JavaBean and map values within the EL. When you use the dot operator, the code to the left of the operator must specify a JavaBean or a map, and the code to the right of the operator must specify a JavaBean or a map key and the property name follows the conventions of Java identifiers.

Syntax:

```
${mapObject.keyName}
```

The EL allows to replace dot notation with array notation (square brackets).

Example: \${param.header} can be replaced with \${param["header"]}

The array notation operator works similar to the dot operator. However, it has its own advantages as follows:

- Array notation operator can be used to access elements of an array or a list by specifying an index. That is the value inside the brackets can itself be a variable in array notation while with dot notation the property name must be a literal value.

- The array notation lets you use values as property names.

Code Snippet 9 demonstrates the use of dot operator.

Code Snippet 9:

```
<!-- home.jsp -->
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Home</title>
  </head>
  <body>
    <p>Enter Username and Submit:</p><br /><br />
    <form action="welcomeuser.jsp" method="post">
      <input name="txtval" type="text" size="20" maxlength="60"
placeholder="Enter Username..." required/>
      <input name="sbtName" type="submit" value="submit" />
    </form>
    <%
      /* setAttribute(name,object) binds an object to a given attribute
name */
      application.setAttribute("PHP", "Contact Form in PHP");
      application.setAttribute("HTML5", "HTML5 new tags");
      application.setAttribute("Review", "Complete Review of all the
documents");
    %>
  </body>
</html>
```

```
<!-- welcomeuser.jsp-->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Employee Work Details</title>
  </head>
  <body>
    <h1>Work Details!</h1><br /><br />
    <b>Employee: ${param.txtval}</b><br />

    <p>Please refer to the given topics and complete it till Monday:</p>
    <ul>
      <li>${applicationScope.PHP}</li><br />
      <li>${applicationScope.HTML5}</li><br />
      <li>${applicationScope.Review}</li><br />
    </ul>
  </body>
</html>
```

Figure 11.2 shows the username entered as RockSmith during submitting the form.

The screenshot shows a Mozilla Firefox browser window. The title bar says "Home - Mozilla Firefox". The menu bar includes "File", "Edit", "View", "History", "Bookmarks", "Tools", and "Help". Below the menu is a toolbar with icons for Home, Stop, Back, Forward, and Stop. The address bar shows "localhost:8084/WebApplication /home.jsp". The main content area displays a form with the text "Enter Username and Submit:". Below the text is a text input field containing "RockSmith" and a submit button labeled "submit".

Figure 11.2: Username Details

Figure 11.3 shows the output after submitting the form successfully.

The screenshot shows a Mozilla Firefox browser window titled "Employee Work Details - Mozilla Firefox". The menu bar and address bar are similar to Figure 11.2. The main content area displays a large heading "Work Details!". Below it, the text "Employee: RockSmith" is shown. Further down, the text "Please refer the below given topics and complete it till Monday:" is followed by a bulleted list:

- Contact Form in PHP
- HTML5 new tags
- Complete Review of all the documents

Figure 11.3: Output of Code Snippet 9

11.4 Functions Using EL

JSP allows the users to define a function that can be invoked in an expression. Functions are defined using the same technique as custom tags.

11.4.1 Creating static Method

The **static** Java methods can be called within the EL expression. To access the function using EL, the function must be implemented as a **static** function in a Java class. You can define many functions in a single class. After defining the functions, you need to map the function name with EL using a Tag Library Descriptor (TLD) file.

Syntax:

```
ns:funcName(arg1, arg2, ...)
```

where,

- ns refers to name space and function name. Name space is generally a class or a tag library or a function name to access a static method in some class.

Figure 11.4 demonstrates the static function.

```
package mypackage;
public class MyFunctions {

    public static double
        average(double [] values){
            double dblSum=0.0;
            int iCount = values.length();
            for (int i=0;i<iCount;i++)
                dblSum+=values[i];
            return dblSum/iCount;
        }
}
```

Figure 11.4: Static Function

11.4.2 Creating Tag Library Descriptor

A TLD file uses XML syntax to map the name of functions defined in a class with EL. In the `<function>` tag in a tag library descriptor file, you need to mention the name of the function using element `<name>`. Also, you need to mention the class in which the function is defined using `<function-class>` element and the signature of the function in the TLD file using `<function-signature>` element. You then save this TLD file in the `/WEB-INF/tlds` folder, where `tlds` is a user-created folder.

Figure 11.5 depicts EL function configuration in the TLD.

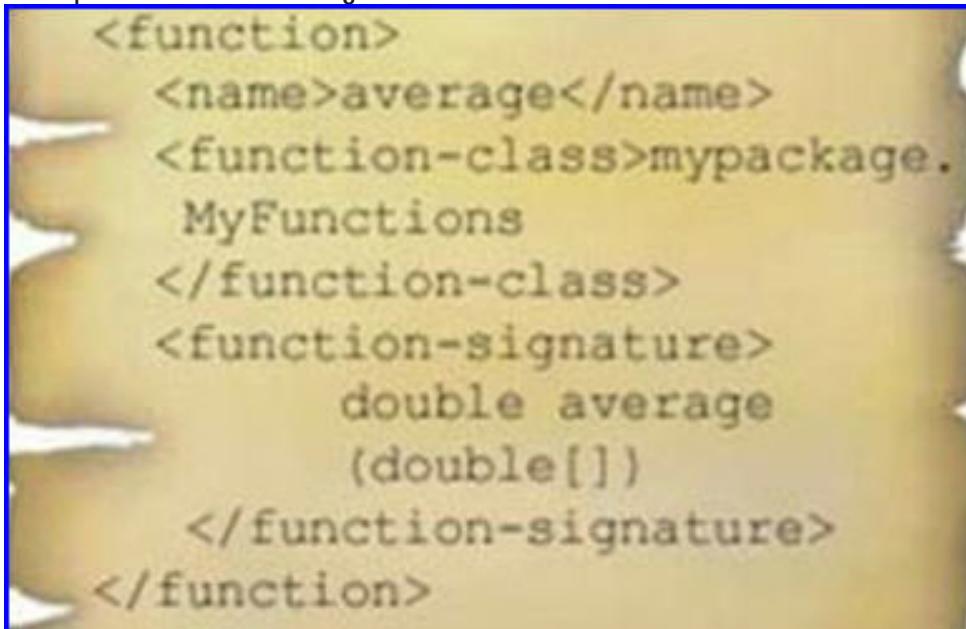


Figure 11.5: EL Function Configuration in TLD

11.4.3 Modifying Deployment Descriptor

The default mode for JSP version 1.2 technology or before is to ignore EL expressions. The default mode for JSP pages delivered with JSP version 2.0 technology is to evaluate EL expressions.

Setting the value of the `<el-ignored>` element in the deployment descriptor can explicitly change the default mode. The `<el-ignored>` element is a sub element of `<jsp-property-group>`. It has no sub elements. Its valid values are `true` and `false`.

In deployment descriptor, `web.xml` file, declare as follows:

Syntax:

```
<el-ignored>false</el-ignored>
```

where,

- `true` indicates that EL expressions will be ignored.

- `false` indicates that EL expressions will be enabled for interpretation by servlet container.

Code Snippet 10 shows the deployment descriptor, `web.xml` which has to be modified for setting the support for the EL in JSP.

Code Snippet 10:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         ...
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>false</el-ignored>
    </jsp-property-group>
</jsp-config>
</web-app>
```

In the code,

- `<jsp-config>`: Includes JSP configuration, such as interpretation of tag library and property information.
- `<jsp-property-group>`: Defines a set of properties that applies to a set of files representing the JSP pages.
- `<url-pattern>`: Specifies that JSP properties defined in `<jsp-property-group>` to specific JSP pages, `*.jsp` indicates that these apply to all JSP pages.
- `<el-ignored>`: Enables interpretation of JSP EL in JSP pages.
- `<scripting-enabled>`: Allows JSP scripting.

11.4.4 Accessing EL Functions within JSP

To access the function created in a TLD file using a JSP file, you need to import the TLD file using the `taglib` directive. In the directive statement, you need to mention a prefix for the tags and the location of the TLD file.

After importing the TLD file, you can access the function using an EL expression.

For the `taglib` directive, syntax declares as follows:

Syntax:

```
<%@ taglib prefix = "prefix" uri = "path" %>
```

where,

- `prefix` is the prefix to be used for the tags defined in the TLD file.

- path is the location of the TLD file.

For accessing the function:

Syntax:

```
<%=${prefix:functionName(arguments)}%>
```

Code Snippet 11 shows how to access EL function in JSP.

Code Snippet 11:

```
<%@taglib prefix="fn" uri="/WEB-INF/tlds/functions"%>
...
Average of the values is : <%=${fn:average(values)}%>
...
```

11.5 Coercion

Automatic conversion of a data from one data type to another data type within an expression is called Coercion. Coercion occurs when the datum is stored as one data type but its context requires a different data type.

11.5.1 Coercion Concept in EL

Coercion means that the parameters are converted to the appropriate objects or primitives automatically. The JSTL defines appropriate conversions and default values. For example, a string parameter from a request will be coerced to the appropriate object or primitive.

If a parameter that represents the month is passed in the request as a string, the value of the month variable will be correct because the string will be coerced to the correct type when used.

Figure 11.6 depicts types of coercion.

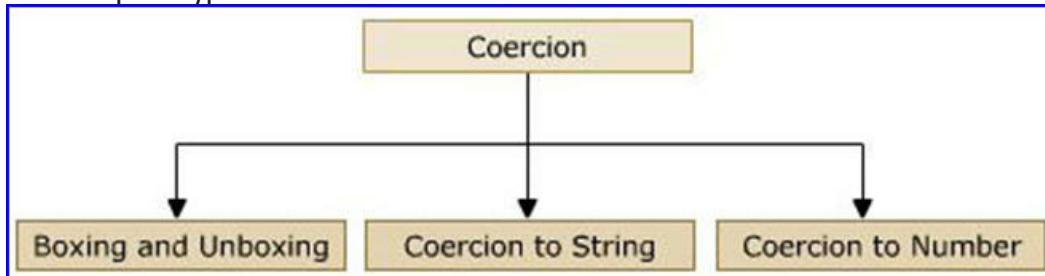


Figure 11.6: Types of Coercion

11.5.2 Boxing and Unboxing

Boxing converts values of primitive type to corresponding values of reference type.

Unboxing converts values of reference type to corresponding values of primitive type.

There are precise rules for boxing and unboxing. They are as follows:

- The precise rules for boxing
 - a. If `i` is a `boolean` value, then boxing conversion converts `i` into a reference `r` of class and type `Boolean`, such that `r.value() == i`.
 - b. If `i` is a `byte` value, then boxing conversion converts `i` into a reference `r` of class and type `Byte`, such that `r.value() == i`.
- The precise rules for unboxing
 - a. If `r` is a `Boolean` reference, then unboxing conversion converts `r` into a value `v` of type `boolean`, such that `r.value() == v`.
 - b. If `r` is a `Byte` reference, then unboxing conversion converts `r` into a value `v` of type `byte`, such that `r.value() == v`.
 - c. If `r` is a `Character` reference, then unboxing conversion converts `r` into a value `v` of type `char`, such that `r.value() == v`.

11.5.3 Coercion to String

The rule to coerce a value to `String` type is as follows:

- `A` is `String`, return `A`.
- `A` is `null`, return `""`.
- `A.toString()` throws exception, return `error`. Otherwise return `A.toString()`.

11.5.4 Coercion to Number

The rule to coerce a value to number type is as follows:

- If `A` is `null` or `""`, return `0`

`A` is character and is converted to short, you apply the following rules:

- If `A` is `Boolean`, return `error`
- If `A` is number type, return `A`

`A` is number, coerce occurs quietly to type `N` using the following algorithms:

- If `N` is `BigInteger`
- If `A` is `BigDecimal`, return `A.toBigInteger()`
- Otherwise, return `BigInteger.valueOf(A.longValue())`
- if `N` is `BigDecimal`

- If A is a BigInteger, return new BigDecimal(A)
- Otherwise, return new BigDecimal(A.doubleValue())
- If N is Byte, return new Byte(A.byteValue())
- If N is Short, return new Short(A.shortValue())
- If N is Integer, return new Integer(A.intValue())
- If N is Long, return new Long(A.longValue())
- If N is Float, return new Float(A.floatValue())
- If N is Double, return new Double(A.doubleValue())
- Otherwise return error

If a variable myInteger is to be declared and its value in an expression is used as a number, a variable is declared with the value using <c:set> as shown in Code Snippet 12.

Code Snippet 12:

```
<c:set var="myInteger" value="${param.month}" />
<p>
The value of myInteger is: <c:out value="${myInteger}" />.
</p>
```

The given statement can be demonstrated by performing a multiplication operation to show that the type is correct: <c:out value="\${myInteger * 2}" />

Check Your Progress

1. Which of the following statements relating to JSP script expression is incorrect?

(A)	JSP is used to display the dynamic content on a Web page.	(C)	The JSP expression is inserted into the page after being evaluated and converted to string.
(B)	The dynamic content generated using JSP language can be formatted.	(D)	Semicolon is not used at the end of the JSP script expression.

2. Which option is the correct one for <jsp:include> from the following statements?

(A)	EL is simple and robust.
(B)	EL can handle both expressions and literals.
(C)	The application data stored can be accessed but cannot be manipulated by EL.
(D)	The advantage of EL is that it provides cleaner syntax and is specifically designed for JSP.
(E)	JSP expression language cannot be used to display the generated dynamic content in a table on a Web page.
(F)	EL makes it possible to easily access application data stored in JavaBeans components.
(G)	JSP expression language can be used to display the generated dynamic content in a table on a Web page.
(H)	The page authors should be master in programming language, such as Java and JavaScript to utilize the benefits of EL.

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

3. Which implicit object defined in JSP EL can be used without creating an instance of the object?

(A)	servletContext	(C)	session
(B)	pageContext	(D)	request

Check Your Progress

4. Which scope is accessible from the JSP page that creates the object?

(A) requestScope	(C) sessionScope
(B) pageScope	(D) applicationScope

5. Which among the following statements related to `pageContext` is incorrect?

(A)	The <code>servletContext</code> object specifies the servlet of the JSP page and Web components contained in the same application.
(B)	The <code>session</code> object represents the session created for the client sending a request.
(C)	The <code>request</code> object represents the request accepted by the JSP page from a client and information of the request.
(D)	The <code>response</code> object represents the response sent by the client using a JSP page.

6. What will be the output of the code snippet `${"2"+"2"}` ?

(A) <code>{"2"+"2"}</code>	(C) <code>2+2</code>
(B) <code>22</code>	(D) <code>4</code>

7. Which among the following operators is not a relational operator?

(A) Logical and (<code>&&</code>)	(C) Inequality (<code>!=</code>)
(B) Logical or (<code> </code>)	(D) Boolean complement (<code>!</code> or <code>not</code>)

Answer

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

Summary

- EL is simple and robust. It can handle both expressions and literals, which are constants and are assigned some memory location.
- EL is a great help to the page authors in accessing and manipulating the application data without mastering the complexities of the programming language such as Java and JavaScript.
- JSP implicit objects are a standard set of classes. The user creates an instance of an implicit object to use available methods and variables.
- Operators are used to perform different arithmetic, relational, and logical operations. Dot operator (.) or [] is used to access value of a variable. Various operators used in Expression Language are arithmetic operators, relational operators, logical operators, empty operators, and dot operators.
- In Expression language the static java methods can be called within the EL expression. To access the function using EL, the function must be implemented as a static function in a java class.
- The TLD file uses XML syntax to map the name of functions defined in a class with EL. Setting the value of the <el-ignored> element in the deployment descriptor can explicitly change the default mode.
- The accessing of the function created in a TLD file using a JSP file is possible by importing the TLD file using the taglib directive.
- Coercion means that the parameters are converted to the appropriate objects or primitives automatically. Coercion is an implicit type conversion.

Technowise



Are you a
TECHNO GEEK
looking for updates?

Login to

www.onlinevarsity.com



Welcome to the Session, **JavaServer Pages Standard Tag Library**.

This session introduces you to JavaServer Pages Standard Tag Library (JSTL) that provides a script-free environment. The session explain the core and SQL library tags available in JSTL and their usage in the JSP page.

In this Session, you will learn to:

- Explain the concept and need for JSTL
- List the advantages of using JSTL
- Describe the different tag libraries available in JSTL
- Explain how to configure JSTL library in NetBeans
- Explain general purpose tags
- Explain decision-making in the tags
- Explain iteration tags in the core tag library
- Explain the different tags available in the SQL tag library

12.1 Introduction

Consider a situation in a shopping cart Web application, where a JSP page generates a dynamic list displaying user-selected products on the Web page. To display such a dynamic list in a cart with selected or removed products, only HTML tags will not be useful. In such a case, the Web designer has to involve himself in coding the scripts, such as iterating through the list elements or displaying data based on a condition. This may not be feasible in all the situations.

To help Web designers, Sun Microsystems developed a pre-defined tag library which contains tags related to the core common functionalities performed in Java applications. The pre-defined tag library is known as JavaServer Pages Standard Tag Library (JSTL) that helps you to reuse standard tags that work in the similar manner in every Java Web application. JSTL allows programming using tags rather than scriptlet code.

12.1.1 Designing JSP Pages with JSTL

Organizations handling large Web applications need separate job roles to handle different parts of the development. Some of the roles found in Web applications are described as follows:

- **Web Designers** – They are involved in creating a view part of the application. The views are basically HTML pages.
- **Web Component Developers** – They are responsible for developing the controller of the application. The controller is basically a Servlet written in Java language.
- **Business Component Developers** – They are responsible for creating the model for the application. The model is basically a component, such as a Java class or an Enterprise JavaBean (EJB) that are basically used to process the business logic of an application.

The use of tag libraries, such as JSTL helps people involved in the application development to focus on their area of expertise, such as Web designers, who are involved in designing Web pages. The JSTL tag library help Web designers to integrate the Java technology code into JSP, without the need to write complex scriptlet code.

12.1.2 Advantages of JSTL

JSTL is a very beneficial technology for the development of JSP applications. Some of the benefits of using JSTL are as follows:

- JSTL provides most of the functionality necessary for the development of JSP application, so you don't have to develop your own tag.
- JSTL has a number of tags for common tasks, such as interacting with databases, iterating through lists, formatting data, handling XML data, and more. You can use these tags and create a program with business logic that saves lots of development time that you could waste in the developing the common tags.

- JSTL is based on XML, which is very similar to HTML. Hence, HTML programmers easily start programming using JSTL.
- Since JSTL is always expressed in XML-compliant tags, it will be easier for HTML generation tools to parse the JSTL code that is contained within the document.
- JSTL includes tags that support both formatting and multilingual support, which provides a consistent approach to formatting of numbers and strings, and internationalization (I18N) support features in JSP scriptlet code.
- JSTL provides mechanism that enables the programmer to develop own custom tags as well.

12.1.3 JSTL Tag Libraries

JSTL provide various tag libraries that can be used for many functionalities.

Figure 12.1 shows the JSTL tag libraries.

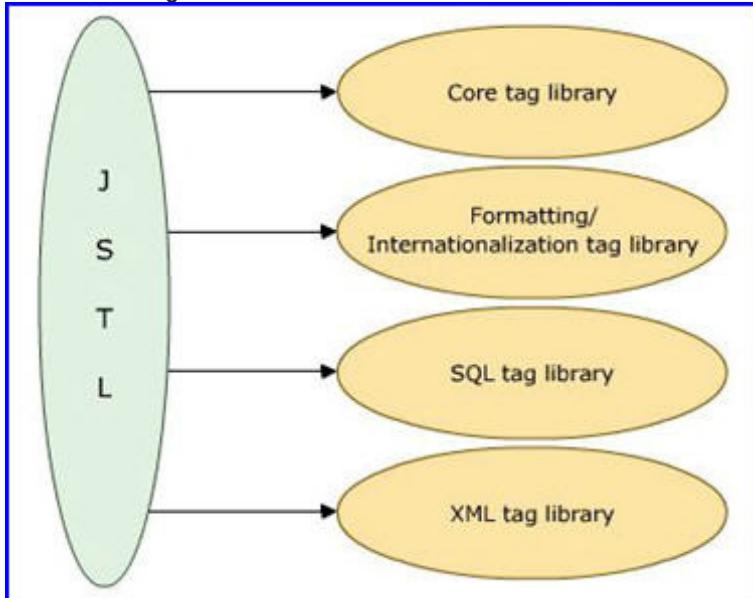


Figure 12.1: JSTL Tag Libraries

□ Core Tag Library

The library contains the tags for looping, expression evaluation, and basic input/output. It can be declared by `<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>`.

□ Formatting/Internationalization (I18N) Tag Library

The library contains the tags which are used to parse the data such as dates and time, based on the current location. It can be declared by `<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>`.

□ SQLTag Library

The library contains tags which are used to access SQL databases. It provides an interface for executing SQL queries on database through JSP. It can be declared by `<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>`.

□ XML Tag Library

The library contains tags for accessing XML elements. In JSTL, XML processing is an important feature, as it is used in many Web applications for managing data. It can be declared by `<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>`.

12.1.4 JSTL Library in NetBeans

To use the JSTL tags on the JSP page, you need to include the JSTL library in the Web application. Perform the following steps:

1. Right-click **Libraries** and select **Add Library**.
2. Select **JSTL 1.1** under **Available Libraries** and click **Add Library**. The **JSTL 1.1 - standard.jar** and **JSTL 1.1 - jstl.jar** are added in the project.

Figure 12.2 shows the JSTL library added in **MyWebApp** application project created in NetBeans IDE.

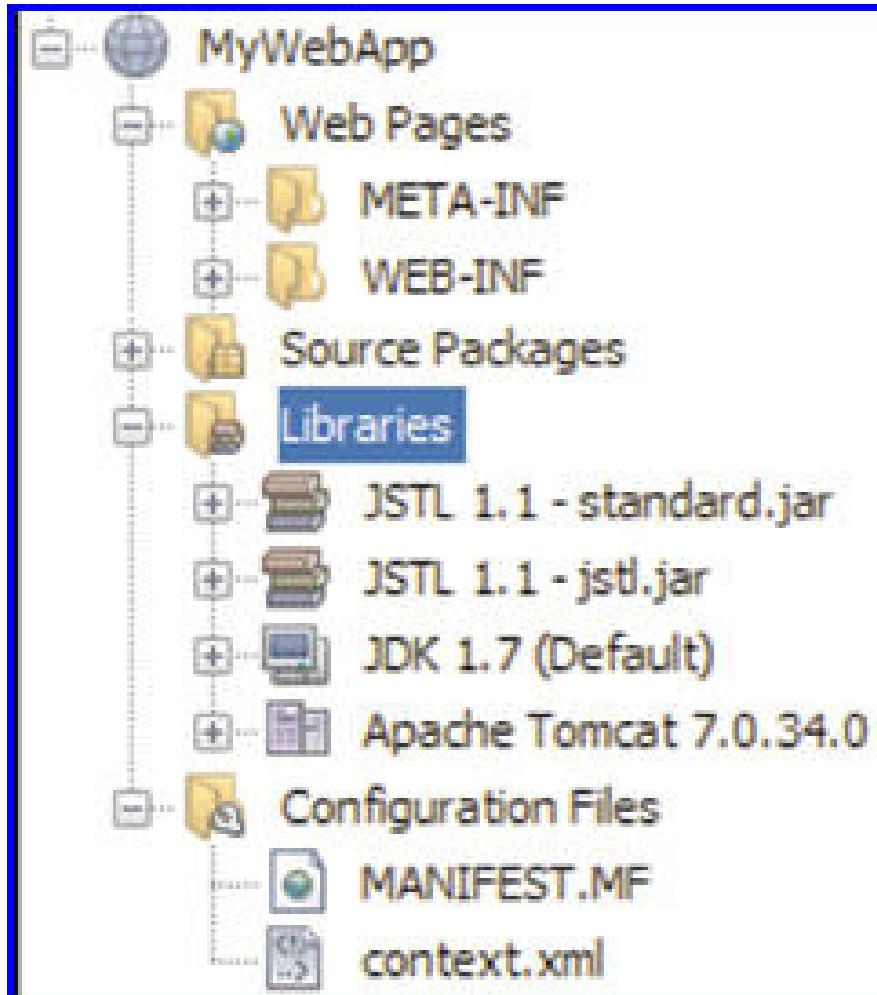


Figure 12.2: JSTL Library

12.2 Core Tag Library

Core tag library provide support for conditional logic, iteration, forward or redirect response, URL, catch exception, and so on. It is the most frequently used JSTL tag library containing core group of tags. The core tag library tags are prefixed with `c`.

12.2.1 General Purpose Tags

General purpose tags are used to set, remove, and display variable values that are created within a JSP page. The core tag library contains tags for getting, setting and displaying attribute values.

The general-purpose tags are:

<c:set>

This tag assigns a value to a variable in scope.

Syntax:

```
<c:set var="varName" value="expression" scope="page/request/
session/application"/>
```

where,

- `value` specifies the expression.
- `var` specifies the name of the exported scope to hold the value specified in the tag.
- `scope` specifies the scope of variable such as page, request, session, and application. The default scope is page.

Example: `<c:set var = "sessionvariable" value = "${80+8}" scope = "session" />`

<c:remove>

This tag is used to remove a scoped variable. This is an empty tag.

Syntax:

```
<c:remove var="varName" scope="page/request/session/
application"/>
```

where,

- `var` specifies the name of the variable to be removed.
- `scope` specifies the scope of the variable.

Example: `<c:remove var="simple" scope="page" />`

<c:out>

This tag is used to evaluate an expression and store the result in the current `JspWriter` object.

Syntax:

```
<c:out value="value" escapeXml="boolean" default="defaultValue"/>
```

where,

- **value** specifies the expression.
- **default** specifies the default value if the value of the result is null.
- **escapeXml** determines that special characters in the result, such as <, >, &, ', and '' should be converted to their character entity codes. The default value of **escapeXml** is true.

Example: <c:out value = "\${sessionvariable}"></c:out>

□ <c:catch>

This tag provides an exception handling functionality such as try-catch, inside JSP pages without using scriptlets.

Syntax:

```
<c:catch [var="varName"]>
  nestedactions
  ...
</c:catch>
```

where,

- **var** specifies the name of the variable to be caught.

Code Snippet 1 demonstrates how to catch the exception using **<c:catch> tag**.

Code Snippet 1:

```
<body>
<c:catch var="e">
  100 divided by 0 is <c:out value="${100/0}" />
<br />
</c:catch>
```

The **catch** tag provides an error handling mechanism for the division operation. The exception raised by dividing the number from 0 is stored in the variable **var**.

Code Snippet 2 demonstrates the core tags used in login application.

Code Snippet 2:

```
<!-- welcome.html -->
<body>
    <form action="login.jsp">
        User Name:
        <input type="text" name="username"/>
        <br/><br/>
        Password:
        <input type="password" name="password"/>
        <br/><br/>
        <input type="submit" value="Submit" name="Submit">
    </form>
</body>
```

```
<!-- login.jsp -->
...
<body>
    <c:set var="name" value="${param.username}"/>
    <c:set var="password" value="${param.password}"/>
    Entered Name: <c:out value="${name}" /><br/><br/>
    Entered Password: <c:out value="${password}" />
</body>
```

The `login.jsp` page assigns the variables `name` and `password` with the value from the request parameters `username` and `password` respectively. The `out` tag outputs the value of `name` and `password` respectively.

12.2.2 Decision-making Tags

JSTL provides decision-making tags to support conditions in a JSP page. Decision-making tags are necessary as the contents or the output of the JSP page is often conditional based on the value of the dynamic application data.

The two types of decision-making tags are:

- `<c:if>`

The tag is used for conditional execution of the code. This tag is a container tag that allows the execution of the body if the test attribute evaluates to true.

Syntax:

```
<c:if test="testCondition" var="varName" scope="page/
request/session/application">

    Body Content

</c:if>
```

where,

- **test** specifies the test condition.
- **var** specifies the name of the variable of the test condition.
- **scope** specifies the scope of the variable, **var**.

In **<c:if>** tag, attribute **var** and **scope** are optional.

Code Snippet 3 demonstrates the use of **if** tag to evaluate.

Code Snippet 3:

```
<!-- Test the username -->
<c:if test="${name=='admin'}">
    <h4>Valid User </h4>
</c:if>
```

The code uses **if** tag to check the value of the **name** variable. If the condition evaluates to true, that is, the value assigned to the **name** variable is **admin**, then the body gets executed.

□ <c:choose>

The tag is similar to the **switch** statement in Java. The **<c:choose>** tag performs conditional block execution.

The **<c:choose>** tag processes body of the **<c:when>** tag. Multiple **<c:when>** tags can be embedded in a **<c:choose>** tag. If none of the conditions evaluates to true, then the body of **<c:otherwise>** tag is processed.

Syntax:

```
<c:choose>
    <c:when test="testcondition">
        Body Content
    </c:when>
    ...
    <c:otherwise>
        Body Content
    </c:otherwise>
</c:choose>
```

where,

- `<c:when>` is the body of `<c:choose>`. It is a container tag and has test condition. It will execute the body content if the test condition evaluates to true.
- `<c:otherwise>` is executed when none of the test conditions of `<c:when>` evaluates to true.

Code Snippet 4 demonstrates the `choose` tag.

Code Snippet 4:

```
// Performs the checking of the condition
<c:choose>
    <c:when test="${name == 'admin'}">
        <h4>Valid User </h4>
    </c:when>
    <c:otherwise>
        <h4> Invalid User </h4>
    </c:otherwise>
</c:choose>
```

The `choose` tag performs evaluation of the condition to test if name entered is '`admin`'. If the condition evaluates to true, then the nested `when` tag gets executed, else the nested `otherwise` tag gets executed.

12.2.3 Iteration Tags

The iteration tag is required for performing looping function. The object can be retrieved from a collection in the JavaBeans components and assigned to a scripting variable by using iteration tags.

The two iteration tags are:

- <c:forEach>**

This tag is used to repeat the body content over a collection of objects. The iteration will continue for a number of times specified by the user in the code.

Syntax:

```
<c:forEach var="varName" item="collection" varStatus=
"varStatusName" begin="begin" end="end" step="step">
    Body Content
</c:forEach>
```

where,

- **var** specifies the name of the exported scoped variable.
- **item** specifies the collection of items to iterate over.
- **varStatus** specifies the name of the variable for the status of iteration.
- **begin** specifies the index from which the iteration is to begin.
- **end** specifies the index at which the iteration is to end.
- **step** specifies that iteration will process every item of the collection.

Code Snippet 5 demonstrates the `forEach` tag.

Code Snippet 5:

```
// Displays objects from collection "companies"
<c:forEach var="company" items="${companies}">
    ${company} <br>
</c:forEach>
// Displays values from 10 to 100
<c:forEach var="i" begin="10" end="100">
    ${i}
</c:forEach>
```

<c:forTokens>

This tag is used to iterate over a collection of tokens separated by user-specified delimiters. It is a container tag.

Syntax:

```
<c:forTokens items="stringofToken" delims="delimiters" var=
"varName" varStatus="varStatusName" begin="begin" end="end"
step="step">
    Body Content
</c:forTokens>
```

where,

- `items` specifies the string of value to iterate.
- `delims` specifies the character that separates the tokens in the string.
- `var` specifies the name of the scope variable for the item of iteration.
- `varStatus` specifies the name of the scope variable for the status of iteration.

Code Snippet 6 demonstrates the use of `<c:forTokens>` tag.

Code Snippet 6:

```
// Displays each token at a time separated by ',' delimiter
<c:forTokens var="token" items="Tom,Dick,Harry" delims=",">
    <c:out value="${token}" />
</c:forTokens>
```

12.3 SQL Tag Library

To interact with other databases such as Oracle, MySQL, or Microsoft SQL Server, JSTL SQL tag library is used. These tags are prefixed by `sql`.

12.3.1 `<sql:setDataSource>`

JSTL SQL tags are used to access databases and are designed for low-volume Web-based applications. SQL tag library provides tags that allow direct database access within a JSP page.

The JSTL SQL tag provides the following functionalities:

Passing database queries

The functionality provides users with the SQL tag library database. This database allows executing queries, such as `SELECT` statement.

This can be executed using the `<sql:query>` tag.

Accessing query results

The functionality allows the users to access results for queries.

Database modifications

The functionality helps in modifying database using the `<sql:update>` tag.

Syntax:

```
<sql:setDataSource dataSource="datasource" | url="jdbcurl"
driver =
"driverclassname" user="username" password="userpwd" var=
"varname" scope="page/request/session/application"/>
```

where,

- `dataSource` can either be the path to Java Naming and Directory Interface (JNDI) resource or a JDBC parameter string.
- `url` is the URL associated with the database.
- `driver` is a JDBC parameter and takes the driver class name.
- `user` takes the user of the database.
- `password` takes the user password.
- `var` is the name of exported scoped variable for the data source specified.
- `scope` specifies the scope of the variable.

Code Snippet 7 demonstrates the use of `<sql:setDataSource>` tag to make the connection to SQL Server database.

Code Snippet 7:

```
// Sets the data source for the database
<sql:setDataSource
    driver="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    url="jdbc:microsoft:sqlserver://10.1.3.27:1433;
    DataBaseName=pubs;" user="sa" password="playware" var="conn"/>
```

In the code, `<sql:setDataSource>` is used to set a data source for the database. This is an empty tag and allows the user to set data source information for the database. In `<sql:setDataSource>` tag, then `url` attribute provides the JDBC url string for the SQL Server database.

12.3.2 <sql:query>

The `<sql:query>` tag searches the database and returns a result set containing rows of data. The tag can either be an empty tag or a container tag. The `SELECT` statement is used to select data from the table.

Syntax:

```
<sql:query sql="sqlQuery" var="varName" scope="{page|request|session|application}" dataSource="dataSource" maxRows="maxRows" startRow="startRow"/>
```

where,

- `sql` specifies the SQL query statement.
- `var` specifies the name of the exported scope variable for the query result.
- `scope` specifies the scope of the variable.
- `dataSource` specifies the data source associated with the database to query.
- `maxRows` specifies the maximum number of rows to be included in the result.
- `startRow` specifies the row starting at the specified index.

Syntax:

```
<sql:query var="varName" dataSource="dataSource" scope="{page|request|session|application}" maxRows="maxRows" startRow="startRow">
    SQL Statement
    <sql:param/>
</sql:query>
```

where,

- `param` takes the parameter for the query.

Code Snippet 8 demonstrates the use of `<sql:query>` tag.

Code Snippet 8:

```
// Gets the records about the 'product' from data source 'conn'
<sql:query var="products" dataSource="${conn}">
    select * from Products
</sql:query>
```

12.3.3 <sql:update>

The <sql:update> tag executes the INSERT, UPDATE, and DELETE statements. It returns 0, if no rows are affected by the DML statements.

Syntax:

```
<sql:update sql="sqlUpdate" dataSource="dataSource" var="varName" scope="{page|request|session|application}"/>
```

where,

- **sql** specifies the update, insert, or delete statement.
- **dataSource** specifies the data source associated with the database to update.
- **var** specifies the name of the exported scope variable for the result of the database update.
- **scope** specifies the scope of variable, such as page, request, session, or application.

Syntax:

```
<sql:update dataSource="dataSource" var="varName" scope="{page|request|session|application}">
    SQL Statement
    <sql:param value="value"/>
</sql:update>
```

where,

- **update** is the UPDATE statement in SQL.
- **param** takes the parameter for the query.

Code Snippet 9 demonstrates the use of <sql:update> tag.

Code Snippet 9:

```
// Adds the product details in the Product table
<sql:update var="newrow" dataSource="${conn}"
    INSERT INTO Products (ProductName, ProductType, Price, Brand,
    Description)
    VALUES ('Jeans', 'Clothes', '1000', 'Lee', 'Good Quality Jeans')
</sql:update>
```

Figure 12.3 shows the output of update.

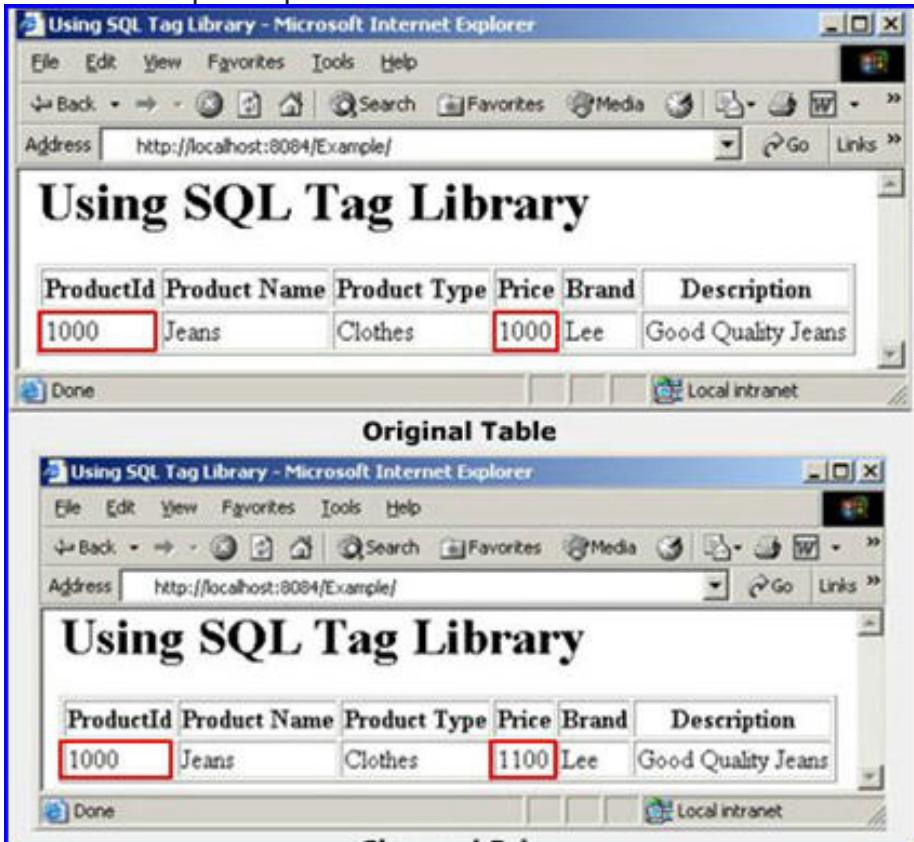


Figure 12.3: Output of Update

12.3.4 <sql:transaction>

The `<sql:transaction>` is used to establish a transaction context for `<sql:query>` and `<sql:update>` tags. The connection object is obtained from `<sql:transaction>` as this tag is responsible for managing access to the database.

Syntax:

```
<sql:transaction dataSource="dataSource" isolation=
isolationLevel>

    <sql:update> or <sql:query> statements

</sql:transaction>
```

where,

- **dataSource** sets the SQL data source which can be a string or a data source object.
- **isolation** sets the transaction isolation level. Isolation level can be `read_committed`, `read_uncommitted`, `repeatable_read`, or `Serializable`.

Code Snippet 10 demonstrates the use of `<sql:transaction>` tag.

Code Snippet 10:

```
/** The code snippet performs transaction by first accessing a data
source to create a table and then inserting a row. It then performs a SQL
queries on the table

*/
<sql:transaction dataSource="${mydatasource}">
    <sql:update var="newTable">
        create table emp (
            id int primary key,
            name varchar(80)
        )
    </sql:update>
    <sql:update var="updateCount">
        INSERT INTO emp VALUES (1, 'Jenny')
    </sql:update>
    <sql:update var="updateCount">
        INSERT INTO emp VALUES (2, 'Christina')
    </sql:update>
    ...
    <sql:query var="empQuery">
        SELECT * FROM emp
    </sql:query>
</sql:transaction>
```

12.3.5 <sql:param>

<sql:param> is used to set values for parameters markers ('?') in SQL statements. It acts as a sub tag for <sql:query> and <sql:update>.

Syntax:

```
<sql:param value="value"/>
```

where,

- value sets the value for the parameter.

Figure 12.4 depicts the use of <sql:param> tag.

```
<sql:query var="product"
    sql="select * from PUBLIC.product where id = ?" >
    <sql:param value="#{productId}" />
</sql:query>

<c:forEach var="productRow" begin="0"
    items="#{product.rows}">
    <sql:update var="product" sql="update PUBLIC. product set
        inventory = inventory - ? where id = ?" >
        <sql:param value="#{item.quantity}" />
        <sql:param value="#{productId}" />
    </sql:update>
</c:forEach>
```

Figure 12.4: <sql:param> Tag

Check Your Progress

1. Match the different tag library against their corresponding description.

Description		Library	
(A)	It contains the tags for looping, expression evaluation and basic input and output.	(1)	Formatting/Internationalization (I18N) Tag Library
(B)	It contains the tags, which are used to parse the data, such as dates and time, based on current location.	(2)	XML Tag Library
(C)	It contains tags, which are used to access databases.	(3)	Core Tag Library
(D)	It contains tags, which are used to access XML elements.	(4)	SQL Tag Library

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

2. Match the tags against the corresponding syntax.

Syntax		Tag	
(A)	<x var = "varName" value = "expression" scope = "page/ request/session/ Application"/>	(1)	<c:remove>
(B)	<x var = "varName" scope = "page/ request/session/ application"/>	(2)	<c:catch>

Check Your Progress

(C)	<x value = "value" escapeXml = "boolean" default = "defaultValue"/>	(3)	<c:set>
(D)	<x [var="varName"]> nested actions	(4)	<c:out>

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

3. Identify the incorrect statement relating to `<sql:setDataSource>` tag.

(A)	<code><sql:setDataSource></code> is an empty tag and allows the user to set data source information for the database.	(C)	In <code><sql:setDataSource></code> tag, if the dataSource attribute is used, then url attribute cannot be used.
(B)	dataSource tag is a path to JNDI resource not a JDBC parameter string.	(D)	In <code><sql:setDataSource></code> tag, scope specifies the scope of the variable.

4. Identify the incorrect statement relating to query tag.

(A)	The <code><sql:query></code> tag searches the database and returns the result set containing columns of data.	(C)	The <code><sql:query></code> tag can be a container tag.
(B)	The <code><sql:query></code> tag can be an empty tag.	(D)	The scope attribute specifies the scope of a variable.

5. Identify the incorrect statement relating to update tag.

(A)	<code><sql:update></code> executes the INSERT, UPDATE, and DELETE statements.	(C)	In <code><sql:update></code> syntax, the var attribute specifies the name of the exported scope variable for the result of the database update.
(B)	Zero is returned if no rows are affected by INSERT, UPDATE, or DELETE.	(D)	In <code><sql:update></code> syntax, scope specifies the scope of the page only.

Check Your Progress

6. Identify the incorrect statement relating to transaction tag.

(A)	<sql:transaction> tag is used to establish a transaction context for <sql:query> and <sql:update> tags.	(C)	In <sql:transaction> syntax, dataSource sets the SQL dataSource, which can only be a dataSource object.
(B)	The connection object is obtained from <sql:transaction>, as this tag is responsible for managing access to the database.	(D)	In <sql:transaction> tag, isolation level can be read_committed, read_uncommitted, repeatable_read, or serializable.

Answer

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

Summary

- JSTL provides a set of reusable standard tags.
- The standard tag library defined by the JSTL works in a similar manner everywhere and this makes the iteration over the collection using scriptlets unnecessary.
- JSTL allows programming using tags rather than scriptlet code.
- The core tag library has general-purpose tags that are used to manipulate scoped variables created within a JSP page.
- Decision-making tags are used to do conditional processing of code in a JSP page.
- Iteration tags are used to iterate over a collection of objects multiple times.
- SQL Tag Library is useful in performing database queries. It allows easy access to query results.
- The database statements, such as insert, update, and delete can be performed by SQL tags.



Welcome to the Session, **JSP Custom Tags**.

This session explains the concept of JSP custom tags. The session explains the different type of custom tags namely, classic custom tags and simple custom tag libraries.

In this Session, you will learn to:

- Explain JSP custom tags in JSP
- Describe the common terminology used in JSP custom tags
- Explain the working of custom tag libraries
- Explain the different types of custom tags available in JSP
- Explain how to create classic custom tags
- Explain use of Tag Extension API
- Explain Simple Tags API

13.1 JSP Custom Tags

Custom tag allows Java developers to embed Java code in JSP pages. By using the custom tag, the developer reduces the overhead of writing the same business logic again for a particular action to be repeated in programs.

Using custom tags in a JSP page involves three steps as follows:

- **Creating a tag library descriptor**

A Tag Library Descriptor (TLD) file contains the information on each tag available in the library. It is an XML document. TLDs are used to validate the tags.

- **Creating a tag handler class**

A tag handler is a Java class that defines a tag described in the TLD file. What a tag will implement depends on what methods can be called and what is required to be implemented.

- **Creating a JSP page that will access the custom tags**

A JSP page will use the tags defined in a tag library by using a taglib directive in the page before any custom tag is used.

13.1.1 Custom Tags Terminology

Some of the terms associated with custom tags are as follows:

- **Tag library descriptor**

This is an Extensible Markup Language (XML) file, which describes custom tags in a JSP program. This contains the definitions of a custom tag and is saved with an extension .tld. This is imported to the JSP program by using the <%@taglib> directive. The JSP container creates an instance of the imported library descriptor file and traces the handler class of the custom tag.

Code Snippet 1 shows a sample tag library descriptor that describes a tag called Name.

Code Snippet 1:

```
<taglib      version="2.0"          xmlns="http://java.sun.com/xml/
j2ee"      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
jsptaglibrary_2_0.xsd">
<!-- The tag library and the tag is described inside this tag-->
<taglib>
<!-- The version of the tag library -->
<tlib-version>2.0</tlib-version>
<!-- The JSP specification version for the tag library -->
<jsp-version>2.0</jsp-version>
<!-- This is the name assigned to the tag library -->
<short-name>tags</short-name>
<!-- This specifies the path of the TLD file-->
<uri>tag lib version id</uri>
<!-- Provides a short description of the tag library-->
<description>The tag library </description>
<!-- Provides a detailed description of the tag-->
<tag>
<!-- This is the name of the tag used in the JSP page-->
<name>name</name>
<!-- This is the name of the tag handler class for the concerned
tag-->
<tag-class>tags.NameTag</tag-class>
<!-- This specifies the type of body content. It can be empty, JSP
or tag-dependent -->
<body-content>empty</body-content>
<!-- This describes the attribute passed with the tag-->
<attribute>
<!-- This specifies the name of the attribute passed with the tag-
-->
<name>name</name>
</attribute>
</tag>
</taglib>
```

□ Tag handler

The tag handler is a simple Java class file that contains the code for the functionality of the custom tag in the JSP program. These are of two types, classic and simple.

Figure 13.1 depicts the custom tags terminology.

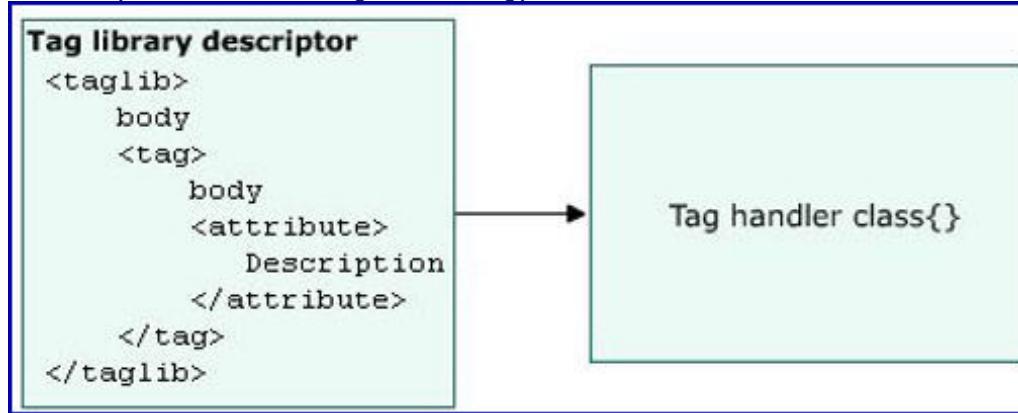


Figure 13.1: Custom Tags Terminology

13.1.2 Tag Libraries

Tag library is a collection of custom tags. This library allows the developer to write reusable code fragments, which assign functionality to tags. The reference to each tag is stored in the tag library. When a tag is used in a JSP page, the library containing the tag has to be imported into the JSP page using the `<%@taglib prefix="u" uri="/WEB-INF/tlds/sampleLib.tld" %>`

where,

- `uri` is the path that uniquely identifies the TLD file of the tag.
- `prefix` is the prefix attribute that distinguishes various tags in the JSP page.

Figure 13.2 depicts the working of custom tag libraries.

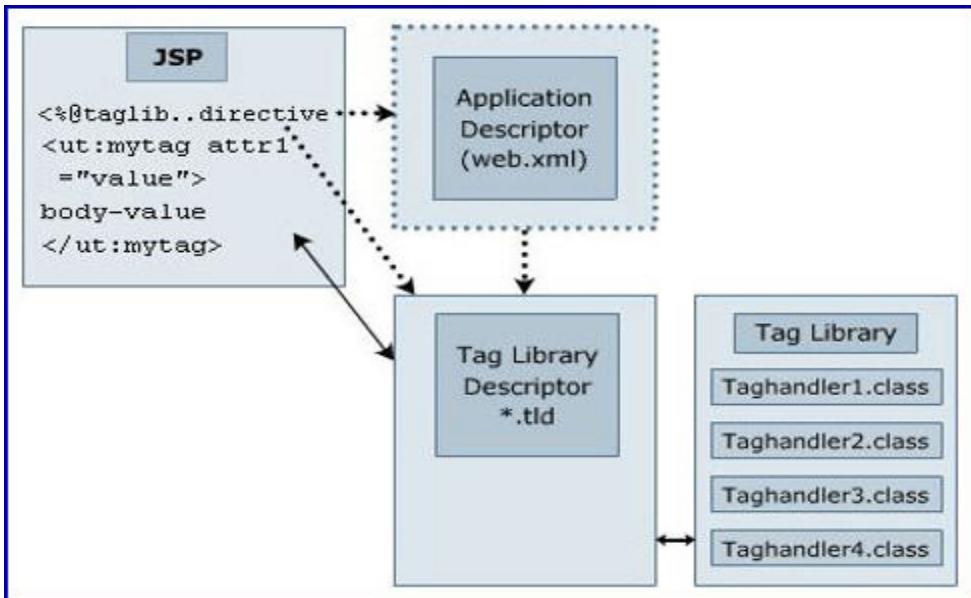


Figure 13.2: Working of Custom Tag Libraries

13.2 Custom Tag Library

The JSP engine is a component of Web container, which converts JSP codes to servlet codes. Jasper2 engine is the name of the JSP engine of Apache since Tomcat version 5.0.

The functions of Jasper2 are as follows:

- Analyze the JSP file and compile them to servlet code.
- Allow a Java object instantiated for the JSP tag to be pooled and reused any time from memory.

13.2.1 Locating TLD File

TLD file can reside in either any directory of a Web application such as:

.../WEB-INF/sampleLib.tld
.../WEB-INF/tld/sampleLib.tld

Alternatively, they can be package in a jar file and place that jar in .../WEB-INF/lib/ directory along with related jars and any related resources.

13.2.2 Associating URIs with TLD File Locations

There is always a specific URI for each .tld file, which refers to its tag in a JSP page. This is basically done by assigning path to the .tld file to `uri` attribute. The `uri` attribute is mapped to the .tld file in two ways namely, implicit and explicit mapping.

In implicit mapping the container reads all the .tld files present in the jar, then the JSP container automatically creates a mapping between the URI specifying the JAR location and the TLD file present inside the JAR file.

Syntax:

```
<%@ taglib prefix= "u" uri="/WEB-INF/TagJARfile.jar" %>
```

13.2.3 Explicit Mapping

The explicit mapping is another method to map the URI to a TLD file. This is done by defining reference to the tag library descriptor by the `<taglib>` element in the deployment descriptor, `web.xml`, of the Web application.

Each `<taglib>` element contains two sub-elements as follows:

- `<taglib-uri>` This is the URI identifying a Tag Library.
- `<taglib-location>` Describes the path to the tag library descriptor file for the custom tag.

Code Snippet 2 uses the tag `<taglib-location>` element to indicate relative path of `AttributeTag.tld`.

Code Snippet 2:

```
<taglib>
    <taglib-uri>AttributeTag</taglib-uri>
    <taglib-location>/WEB-INF/tlds/AttributeTag.tld</taglib-
location>
</taglib>
```

13.2.4 Resolving URIs to TLD File Locations

In explicit mapping, the association of the URIs to the TLD files is called the resolution of URIs. This association is necessary to locate the tag library descriptor file, which defines the tag. If the value of the `uri` attribute matches any of the `<taglib-uri>` entries, the engine uses the value of the corresponding `<taglib-location>` to locate the actual TLD file.

13.2.5 Tag Library Prefix

In a JSP page, several custom tags with same name can be used, which are defined in different tag library descriptors. To distinguish these tags from one another, they are named uniquely by using `prefix` attributes in the `<%@taglib>` directive.

Also, the prefix of a tag is used as a reference by the container to invoke its respective TLD file and tag handler class.

Syntax:

```
<%@taglib prefix="ui" uri="/WEB-INF/tlds/sampleLib_ui.tld" %>
<%@taglib prefix="logic"      uri="/WEB-INF/tlds/sampleLib_logic.
tld" %>
<%@taglib prefix="data"      uri="/WEB-INF/tlds/sampleLib_data.tld"
%>
```

13.3 Using Custom Tags in JSP Pages

Following are the types of custom tags:

- Empty tag
- Tag with attribute
- Custom tags with JSP code
- Nested custom tags

13.3.1 Tags with Attributes

The custom tags with the attributes are called as parameterized tags. Attributes are passed to the tags as arguments are passed to a method. This is done to customize the behavior of a custom tag as shown in Code Snippet 3.

Code Snippet 3:

```
<html><body>
<%@ taglib prefix="tagPrefix" uri="sampleLib.tld" %>
<h1>< tagPrefix:TagsWithAttributesuserName="userName1" /></h1>
</body></html>
```

13.3.2 Custom Tags with JSP Code

Custom tags can contain JSP code as content inside them. The JSP code is simply evaluated or manipulated before being evaluated. The manipulation can be of converting the body content to a string and then formatting it as per requirements as shown in Code Snippet 4.

Code Snippet 4:

```
<html><body>
<%@ taglib prefix="tagPrefix" uri="sampleLib.tld" %>

<tagPrefix:if condition="true">
    UserName is: <%= request.getParameter("userName") %>
</tagPrefix:if>
</body></html>
```

13.3.3 Nested Custom Tags

Nested custom tags are tags declared inside the tags. This is analogous to the `<table>` tag in an html page, which has `<tr>` and `<td>` tags inside it. One custom tag can made a container for another custom tag. The container is called the parent of the tags inside it.

Code Snippet 5 shows the nested custom tags.

Code Snippet 5:

```
<html><body>

<%@ taglib prefix="tagPrefix" uri="sampleLib.tld" %>
<tagPrefix:switch conditionValue='<%= request.getParameter("userName") %>'>

<tagPrefix:case caseValue=" userName1">
    First User
</tagPrefix:case>
<tagPrefix:case caseValue=" userName2" >
    Second User
</tagPrefix:case>
</tagPrefix:switch>

</body></html>
```

13.4 Classic Custom Tags

The classic custom tags were introduced in JSP 1.1 as an extension to the existing standard tags. These tags help in creating customized tags for JSP pages. Now, we will learn how to develop custom libraries according to the 'Classic' model of tag library development.

A classic custom tag uses a tag handler class, which implements `Tag`, `IterationTag` and `BodyTag` interfaces or extends the classes `TagSupport` or `BodyTagSupport`. The classic custom tag is then described in the Tag Library Descriptor (TLD) file.

13.4.1 <taglib> Element

The `<taglib>` element is the top-level or root element of the tag library descriptor. This tag contains some sub tags inside it. These are as follows:

- `<tlib-version>`
This element describes the version of the tag library implementation.
- `<jsp-version>`
This element defines the JSP version.
- `<short-name>`
This element uniquely identifies the tag library from the JSP page.
- `<uri>`
The element is used as a unique resource identifier for the location of a tag library.
- `<tag>`
This element provides all the information required about a particular classic custom tag used in a JSP page.

Code Snippet 6 illustrates the use of the `<taglib>` elements along with its sub elements.

Code Snippet 6:

```
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee           web-
jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>attributetag</short-name>
    <uri>/WEB-INF/tlds/AttributeTag</uri>
</taglib>
```

13.4.2 <tag> Element

The `<tag>` element contains the following subelements to describe the tag characteristics:

- `<name>`
This element assigns a unique name to the classic custom tag inside a JSP page.
- `<tag-class>`
This tag contains the name of the tag handler class that describes the functionality of a particular classic custom tag.
- `<body-content>`
This tag contains the content type for the body of the tag. This tag can be passed values such as empty, JSP, and Body-Content.
- `<attribute>`
This describes the attribute passed in the tag handler class to the JSP page containing the

classic custom tag.

Code Snippet 7 illustrates the use of the `<tag>` elements along with its sub elements.

Code Snippet 7:

```
<tag>
<name>AttributeTag</name>
<tag-class>AttributeTag</tag-class>
<body-content>empty</body-content>
</tag>
```

13.4.3 <attribute> Element

The characteristics of the attribute of a custom tag with attribute need to be described inside the TLD file. The `<attribute>` element inside the `<tag>` element describes the attribute passed in the tag handler class to the JSP page through the classic custom tag. It contains five sub tags as follows:

- name**
The name of the attribute.
- type**
The data type of the attribute.
- required**
This element specifies whether the attribute is required or not. If it is required then 'true' is passed and if not required then 'false' is passed.
- rtextrvalue**
This element specifies whether the attribute is evaluated dynamically or not. If it is evaluated at the time of request or during run time or dynamically then 'true' is passed. If static value is evaluated, then 'false' is passed.
- description**
A brief description of the attribute.

Code Snippet 8 demonstrates the **attribute** element.

Code Snippet 8:

```
<tag>
<name>MyTag</name>
<tag-class>pkg.MyTag</tag-class>
<body-content>JSP</body-content>
<attribute>
    <name>attr1</name>
    <required>true</required>
</attribute>
<attribute>
    <name>attr2</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

13.4.4 <body-content> Element

The **<body-content>** element is defined within the **<tag>** element. It indicates whether the classic custom tag contains any content inside it or not. The body content can have one of three values as follows:

- empty**

Code Snippet 9 shows the tag that does not contain any body inside it to process, then it is called an empty tag.

Code Snippet 9:

```
<tag>
<name>AttributeTag</name>
<tag-class>AttributeTag</tag-class>
<body-content>empty</body-content>
<attribute>
</attribute>
</tag>
```

□ JSP

Code Snippet 10 shows the JSP body content includes any other custom tag, core tag, scripting elements, or html text inside it.

Code Snippet 10:

```
<tag>
<name>UpperCase</name>
<tag-class>ucase.UpperCase</tag-class>
<body-content>JSP</body-content>
</tag>
```

□ tagdependent

This specifies that the tag has a body, but its content is not to be interpreted by the JSP engine, for example, we can pass SQL query as shown in Code Snippet 11.

Code Snippet 11:

```
<tag>
<name>Query</name>
<tag-class>table.Query</tag-class>
<body-content>tagdependent</body-content>
</tag>
```

13.5 Tag Extension API

The tag extension API in the JSP adds the tag functionalities to the language. It is part of the javax.servlet.jsp.tagext package. This contains a number of interfaces and classes, which have their exclusive methods.

13.5.1 Interfaces

There are basically three important interfaces defined in the tag extension API. These are as follows:

□ Tag

This interface allows communication between a tag handler and the servlet of a JSP page. This interface is particularly useful when a classic custom tag is without any body or even if it has body then it is not for manipulation.

□ IterationTag

This interface is used by tag handlers that require executing the tag, without manipulating its content. This interface extends to the Tag interface.

BodyTag

BodyTag extends IterationTag and adds two methods for supporting the buffering of body contents: doInitBody() and setBodyContent().

13.5.2 Classes

The tagext package includes two important classes TagSupport and BodyTagSupport for implementing tag functionalities in a JSP page.

TagSupport

The TagSupport class acts as the base class for most tag handlers which supports, empty tag, tag with attributes, and a tag with body iterations. This class implements the Tag and IterationTag interfaces. Thus, it implements all the methods in these interfaces. It contains methods such as, doStartTag(), doEndTag(), and doAfterBody().

BodyTagSupport

The BodyTagSupport class can support tags that need to access and manipulate the body content of a tag. This class implements the BodyTag interface and extends the TagSupport class. It contains the following methods: setBodyContent() and doInitBody().

13.5.3 Exceptions

In addition to the interfaces and classes, the Tag handler classes use the exception classes which are defined in the javax.servlet.jsp package.

JspException

The JspException is thrown by a tag handler to indicate some unrecoverable error while processing a JSP page.

JspTagException

The JspTagException is a sub class of the JspException, which is thrown when an error is encountered inside the tag handler class while processing any tag.

13.6 Implementing Tag Interface

The Tag interface defines the life cycle and the methods to be invoked at start and end tag.

13.6.1 Methods of Tag Interface

The methods of the Tag interface are as follows:

doStartTag()

The tag handler invokes this method when a request is made to a tag or when the starting tag of the element is encountered.

It returns either of the two field constants, returns the SKIP_BODY constant, if there is no body to evaluate or returns the EVAL_BODY_INCLUDE constant.

Syntax:

```
public int doStartTag() throws JspException
```

- **setPageContext()**

This sets the current page context. This is called by the page implementation prior to doStartTag().

Syntax:

```
public void setPageContext(PageContext pc)
```

The parameter pc is the argument of type pageContext passed to the setPageContext() method.

- **doEndTag()**

The tag handler invokes this method when the JSP page encounters the end tag. It generally follows the execution of the doStartTag() if the tag is empty. This method returns the SKIP_PAGE constant if there is no need to evaluate the rest of the page or returns EVAL_PAGE constant if the rest of the page needs to be evaluated.

Syntax:

```
public int doEndTag() throws JspException
```

- **release()**

The container calls this method on the handler class when the tag handler object is no longer required.

Syntax:

```
public void release()
```

Code Snippet 12 demonstrates the methods of the Tag interface.

Code Snippet 12:

```
//The evaluation of start tag starts.  
public int doStartTag() throws JspException {  
    try {  
        pageContext.getOut().print("Creating Custom Tag by  
Implementing Tag Interface");  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    }  
  
    // The SKIP_BODY is returned  
    return SKIP_BODY;  
}  
}  
//The evaluation of end tag starts.  
public int doEndTag() throws JspException {  
    // the SKIP_PAGE is returned  
    return SKIP_PAGE;  
}  
//All the resources held by the tag handler is released.  
public void release() {  
}  
}  
//The current value of the pageContext is set  
public void setPageContext(PageContext pc) {  
    this.pageContext = pc;  
}  
//All the resources held by the tag handler is released.  
public void release() {  
}
```

13.6.2 Empty Tags

Empty tags are the simple tags without body content. To create an empty classic custom tag by implementing the Tag interface, we need to follow the three basic steps of creating a custom tag.

1. Create a Tag handler file

The tag handler file that defines an empty classic custom tag implements the Tag interface as this tag does not contain any body so only the start and the end tag of the custom tag needs to be processed.

Code Snippet 13 demonstrates the code that creates a tag handler class for the empty tag.

Code Snippet 13:

```
public class ImplementTag implements Tag {  
    private PageContext pageContext;  
    private Tag parent;  
  
    //The current value of the pageContext is set  
    public void setPageContext(PageContext pc) {  
        this.pageContext = pc;  
    }  
    //The instance of the parent tag is set here.  
    public void setParent(Tag tag) {  
        this.parent = tag;  
    }  
    //The instance of the parent tag is retrieved here.  
    public Tag getParent() {  
        return parent;  
    }  
  
    //The evaluation of start tag starts.  
    public int doStartTag() throws JspException {  
        try {  
            pageContext.getOut().print("Creating      Custom      Tag      by");  
            implementing Tag interface");  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
  
        // The SKIP_BODY is returned  
        return SKIP_BODY;  
    }  
    //The evaluation of end tag starts.  
    public int doEndTag() throws JspException {  
        // the SKIP_PAGE is returned  
        return SKIP_PAGE;  
    }  
    //All the resources held by the tag handler is released.  
    public void release() {  
    }  
}
```

2. Create a Tag library descriptor

We will create entry in tag library descriptor file for the empty. The TLD file will provide a detailed description of the Tag library and the custom tag as shown in Code Snippet 14.

Code Snippet 14:

```
<tag>
<name>Implement</name>
<tag-class>mytag.ImplementTag</tag-class>
<body-content>empty</body-content>
</tag>
```

3. Create a JSP page for embedding the tag

In the final step, the tag needs to be embedded in a JSP file as shown in Code Snippet 15.

Code Snippet 15:

```
<%--The tag is imported by the following directive.--%>
<%@taglib uri="/WEB-INF/tlds/TagInterface.tld"
prefix="taginterface"%><HTML>
<HEAD></HEAD>
<BODY>
<%--The custom tag is called here.--%>
<taginterface:ImplementTag></taginterface:Implement>
</BODY>
</HTML>
```

13.6.3 Empty Tag to Accept Attribute

Tags that have attributes are also called parameterized tags, these are basically empty tags.

The steps to create a classic custom tag to accept attribute are as follows:

1. Create a tag handler file

The tag handler class for the classic custom tag that accepts attribute need to implement the Tag interface. Attributes are passed from the tag handler class to the tag inside the JSP page. Only the start and the end tag of the custom tag needs to be processed.

Code Snippet 16 demonstrates the code for defining the attributes for the empty tag.

Code Snippet 16:

```
//The Tag interface is implemented to the TagAttribute class.

public class TagAttribute implements Tag {
private String name;
//The attribute, name is set by the setter method, setName()

public void setName(String name) {
this.name = name;
}
// The pageContext instance is set by the setPageContext()
method

public void setPageContext(PageContext pc) {
    this.pageContext = pc;
}
//The instance of the parent tag is retrieved here.
public Tag getParent() {
    return parent;
}
//The evaluation of start tag starts.

public int doStartTag() throws JspException {
try {
    pageContext.getOut().print(
        "This is my first tag with attribute!"+name);
} catch (IOException ioe) {
    throw new JspException("Error:
    IOException while writing to client"
    + ioe.getMessage());
}
    return SKIP_BODY;
}
//The evaluation of end tag starts.

public int doEndTag() throws JspException {
    return SKIP_PAGE;
}
//All the resources held by the tag handler is released.
public void release() {
}

}
```

2. Create a Tag library descriptor

We will create entry in tag library descriptor file for the empty. The TLD file will provide a detailed description of the Tag library and the custom tag along with the attributes passed to it within the xml elements.

Code Snippet 17 shows the tag library descriptor.

Code Snippet 17:

```
<tag>
<name>welcomparam</name>
<tagclass>tags.TagAttribute</tagclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>name</name>
<required>false</required>
<rteprvalue>false</rteprvalue>
</attribute>
</tag>
```

3. Create a JSP page for embedding the tag

In the final step, the tag needs to be embedded in a JSP file as shown in Code Snippet 18.

Code Snippet 18:

```
<%--The tag is imported by the following directive.--%>
<%@ taglib uri="/WEB-INF/jsp/TagAttribute.tld" prefix="first"%>
<HTML>
<HEAD>
</HEAD>
<BODY>
<%-- The tag is declared --%>
<first:welcomparam name="APTECH"/>
</BODY>
</HTML>
```

13.6.4 Non-empty Tags

Tags that include body inside them are non-empty tags. The two types of decision-making tags are:

We need to follow the three basic steps of creating a custom tag that has a body.

1. Create a tag handler file

The tag handler class for the classic custom tag that has body content implements the Tag interface. The `doStartTag()` method returns `EVAL_BODY_INCLUDE`.

The body is evaluated by the JSP container as any other java codes in the JSP page.

Code Snippet 19 demonstrates the code that creates the handler for the non-empty tag.

Code Snippet 19:

```
public class MyBodyTag implements Tag {  
    private String name=null;  
    private PageContext pc;  
    private Tag pt;  
    public void setName(String value) {  
        name = value;  
    }  
    public String getName() {  
        return(name);  
    }  
    public void setPageContext(PageContext pageContext) {  
        this.pc=pageContext;  
    }  
  
    public void setParent(Tag tag) {  
        this.pt=tag;  
    }  
  
    public Tag getParent() {  
        return pt;
```

2. Create a Tag library descriptor

The TLD file in this case will have the <body-content> as JSP as shown in Code Snippet 20.

Code Snippet 20:

```
<tag>  
<name> MyBodyTag </name>  
<tag-class>mytag.MyBodyTag </tag-class>  
<body-content>JSP</body-content>  
</tag>
```

3. Create a JSP page for embedding the tag

In the final step, the tag needs to be embedded in a JSP file as shown in **Code Snippet 21**.

Code Snippet 21:

```
<%--The tag is imported by the following directive.--%>
<%@taglib uri="/WEB-INF/tlds/UsingTag.tld"
prefix="usingtag"%>
<HTML>
<HEAD>
<TITLE>Tag with Body</TITLE>
</HEAD>
<BODY>
<%-- The tag is declared --%>
<usingtag:Interface>
<br>Current time: <%= new java.util.Date() %>
</usingtag:MyBodyTag>
</BODY>
</HTML>
```

13.7 Implementing IterationTag Interface

The **IterationTag** is implemented to the tag handler class to create tags, which repeatedly evaluates the body inside it and allows including the body content multiple times. This interface is an extension of the **Tag** interface.

13.7.1 Methods of IterationTag Interface

This interface has the method **doAfterBody()**. This method allows the user to conditionally re-evaluate the body of the tag. This method returns the constant **SKIP_BODY** or **EVAL_BODY_AGAIN**, if the **doStartTag()** method returns **EVAL_BODY_INCLUDE** then this method returns **EVAL_BODY_AGAIN** and the **doStartTag()** is evaluated once again. However, when the **doAfterBody()** returns **SKIP_PAGE**, the **doEndTag()** method is invoked.

Syntax:

```
public int doAfterBody() throws JspException
```

13.7.2 Iterative Tag Sample

Code Snippet 22 shows the `IterationTag` in a JSP page.

Code Snippet 22:

```
<%@ taglib prefix="iteration" uri="/WEB-INF/sampleIterationTagLib.tld" %>
<html><body>
<iteration:loop count="5" >
    Hello IterationTag!<br>
</iteration:loop>
</body></html>
```

For the mentioned `IterationTag`, we have to create corresponding java class and `<tag>` element in the TLD file.

13.8 BodyTag Interface

The `BodyTag` interface extends `IterationTag` and adds a new functionality that lets the tag handler evaluate its body content in a temporary buffer.

13.8.1 Methods of BodyTag Interface

The methods inside the `BodyTag` interface are as follows:

- `setBodyContent()`

This method sets the `bodyContent` object for the tag handler. The `bodyContent` object of the `BodyContent` class encapsulates the body content of the custom tag for processing. This method is automatically invoked by the JSP page before the `doInitBody()` method.

Syntax:

```
public void setBodyContent(BodyContent b) { }
```

- `doInitBody()`

This method is invoked immediately after the `setBodyContent()` method returns the `bodyContent` object and before the first body evaluation.

Syntax:

```
public int doAfterBody() throws JspException
```

13.8.2 Sample BodyTag

Code Snippet 23 shows the BodyTag in a JSP page.

Code Snippet 23:

```
<<%@ taglib prefix="bodyTagPrefix" uri="/WEB-INF/sampleBodyTagLib.tld" %>
<html>
<body>
    <bodyTagPrefix:convertTempconvert="C" >
        Temperature is:<%=temp%></br>
    </ bodyTagPrefix:loop>
</body>
</html>
```

For the mentioned BodyTag, we have to create corresponding java class and `<tag>` element in the TLD file.

13.9 Extending TagSupport and BodyTagSupport

The tag handler class needs to implement all the method declarations of the Tag or IterationTag interface when implementing these interfaces. In some case, we do not required to implement all these methods. The TagSupport class, which implements the Tag or IterationTag interface and provide a default implementation of all the methods. The programmer can extend TagSupport class and override the required method.

13.9.1 'TagSupport' Class

The TagSupport class implements the IterationTag interface, and provides default implementations for each of the methods of the Tag and IterationTag interfaces.

Some methods of TagSupport class are as follows:

- **doStartTag()**
This method is inherited from Tag interface and by default return value is SKIP_BODY.
- **doEndTag()**
This method is inherited from IterationTag interface and by default return value is SKIP_BODY.

Figure 13.3 depicts the hierarchy of TagSupport class.

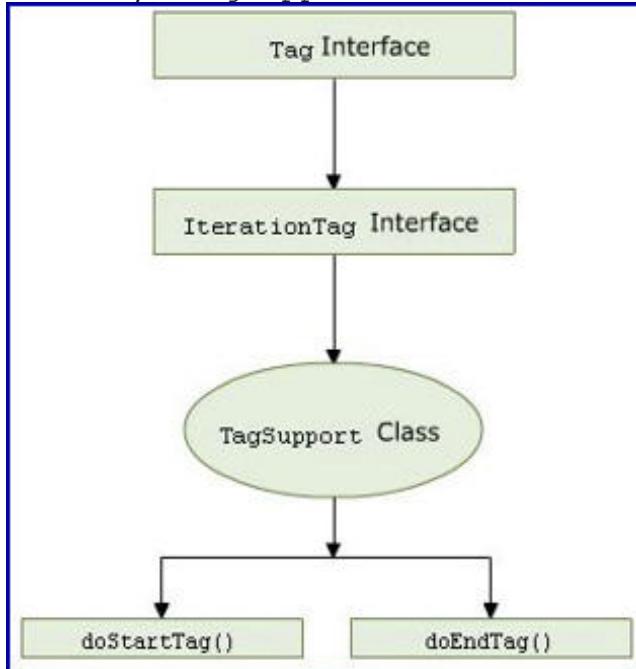


Figure 13.3: Hierarchy of TagSupport Class

13.9.2 BodyTagSupport Class

This class by default implements the BodyTag interfaces and also extends to the TagSupport class. So, the handler class only needs to override the methods of the BodyTagSupport class. All the methods of the TagSupport class and BodyTag interface are implemented by default in the handler class.

Figure 13.4 depicts the hierarchy of BodyTagSupport class.

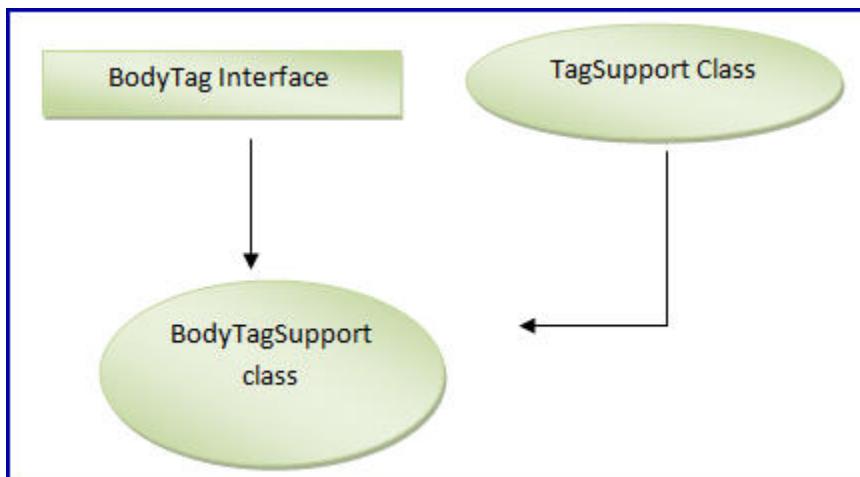


Figure 13.4: Hierarchy of BodyTagSupport Class

The `BodyTagSupport` class contains the following overridden methods:

getBodyContent()

This method retrieves the `bodyContent` object of the class `BodyContent`. This object contains the body of the tag which is accessed by the container for manipulation.

Syntax:

```
public BodyContent getBodyContent()
```

setBodyContent()

The `BodyContent` object is set by the `setBodyContent()` method.

Syntax:

```
public void setBodyContent(BodyContent b)
```

13.10 SimpleTag

Creating custom tag libraries with needs the programmer to create a Java tag handler, reference the class in a Tag Library Descriptor (TLD), and include the TLD in JSP.

The relationship between `SimpleTag` and other JSP tag interfaces is depicted in figure 13.5.

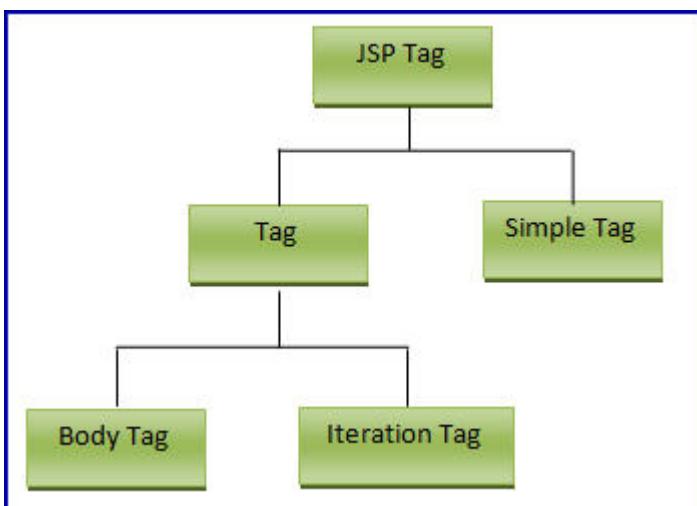


Figure 13.5: JSP Tag Interfaces Hierarchy

The `SimpleTag` interface provides `doTag()` method, which is supported by the classic tag handlers. The `doTag()` is called whenever there is a requirement of tag invocation. All the manipulations and evaluations are carried out by this method.

Following methods are present in **SimpleTag** interface:

doTag()

The method is called by the container to begin **SimpleTag** operation and used for handling tag processing. It retains body iteration after being processed.

Syntax:

```
public void doTag() throws JspException, IOException
```

setParent()

The method sets the parent of a specified tag.

Syntax:

```
public void setParent(JspTag parent)
```

where,

- parent is the new tag that needs to be set as a parent tag.

getParent()

The method returns the parent of the specified tag.

Syntax:

```
public JspTag getParent()
```

setJspContext()

The method sets the context to the tag handler for invocation by the container.

Syntax:

```
public void setJspContext(JspContext pc)
```

setJspBody()

The method is provided by the body of the specified tag, makes the body content available for tag processing.

Syntax:

```
public void setJspBody(JspFragment jspBody)
```

13.10.1 SimpleTagSupport

The **SimpleTagSupport** class acts as a base class for simple tag handlers. The class implements the **SimpleTag** interface. Some of the useful methods are as follows:

getJspContext()

This method returns the context passed into the container by `setJspContext()` method.

Syntax:

```
protected JspContext getJspContext()
```

getJspBody()

This method returns the `JspFragment` object for body processing in the tag.

Syntax:

```
protected JspFragment getJspBody()
```

13.10.2 Implementation of SimpleTag Interface

Code Snippet 24 shows the interface that is implemented by extending the `SimpleTagSupport` class and overriding the `doTag()` method.

Code Snippet 24:

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
public class Greeter extends SimpleTagSupport {
    public void doTag() throws JspException {
        PageContext pageContext = (PageContext) getJspContext();
        JspWriter out = pageContext.getOut();
    }
}
```

Code Snippet 25 shows the corresponding tag in TLD file.

Code Snippet 25:

```
<tag>
<name>greeter</name>
<tag-class>Greeter</tag-class>
<body-content>empty</body-content>
<description>
    Print this on the browser.
</description>
</tag>
```

Code Snippet 26 shows the corresponding JSP file.

Code Snippet 26:

```
<%@ taglib prefix="ui" uri="/WEB-INF/sampleTag.tld" %>
<html><body>
<ui: greeter />
</body></html>
```

Check Your Progress

1. Which of the statement will be executed when the starting end of an empty tag is encountered?

(A)	public int doStartTag() throws JspTagException {}
(B)	public int doEndTag() throws JspTagException {}
(C)	public int doStartTag() throws JspException {}
(D)	public void doEndTag() throws JspException {}

2. Which of these syntax are correct?

(A)	public void setJspBody(JspFragment jspBody)
(B)	public int doEndTag() throws JspException
(C)	public int doStartTag() throws JspException
(D)	public int doStartTag() throws JspTagException
(E)	public int doEndTag() throws JspTagException

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

3. Which of these statements are correct for the tags?

(A)	Nested custom tags are tags inside tags with the outermost tag acting as parent to other tags inside it.
(B)	In case of the nested tag, the tags inside the parent tag obtains reference to the parent class by using the <code>findAncestorWithClass()</code> method.
(C)	The parent tag definition comes first and the inner tags follow it.

Check Your Progress

(D) The `getParent()` method returns the parent tag for the tag, which is enclosed by the parent tag.

(A) B, C, D

(C) A, B, D

(B) A, C, D

(D) A, B, C

4. Which of the following options is NOT correct?

(A) The `getBodyContent()` method retrieves the `bodyContent` object of the `bodyContents` class.

(B) The `BodyTagSupport` class contains the `getBodyContent()` method.

(C) The `BodyTagSupport` class contains the `setBodyContent()` method.

(D) The `BodyTagSupport` class implements the `Tag` and `IterationTag` interfaces.

5. Which of the following is not a valid sub-element of the `<attribute>` element in a TLD?

(A) `<name>`

(B) `<class>`

(C) `<required>`

(D) `<rteexprvalue>`

6. Which of the options are correct about the differences between tag files and tag library descriptors?

(A) Tag files eliminate scriptlets.

(B) TLD needs to be defined earlier.

(C) Syntax of tag files is closer to HTML.

(D) Tag files are written easily.

Check Your Progress

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

7. Which of the following tag file directives control Tag processing?

(A)	tag
(B)	varReader
(C)	include
(D)	attribute

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

8. Which of the options are correct about attributes of <jsp:invoke> standard action?

(A)	fragment
(B)	var
(C)	varReader
(D)	begin

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

Answer

1.	C
2.	A
3.	B
4.	A
5.	B
6.	C
7.	B
8.	A

Summary

- Custom tags are action elements on JSP pages that are mapped to tag handler classes in a tag library.
- Tag libraries allow us to use independent Java classes to manage the presentation logic of the JSP pages, thereby reducing the use of scriptlets and leveraging existing code to accelerate development time.
- The tag without any body is called an empty tag and the tag accepting attributes is called a tag with attributes or a parameterized tag.
- A tag with JSP code or tag dependent strings is called a tag with body. If a tag contains several other tags inside it then it is called a nested tag.
- The Tag Library Descriptor (TLD) file contains the information that the JSP engine needs to know about the tag library in order to successfully interpret the custom tags on JSP pages.
- The SimpleTag interface provides the doTag() method, which is supported by the Classic Tag Handlers.



Welcome to the Session, **Internationalization**.

This session introduces the concept of internationalization and localization in a Web applications. The session explains the Unicode and Locale in internationalization. Further, the session explains how the Web pages should be customized for the clients, based on their local language and cultural formatting conventions. The session also describes the JSTL I18N tag library used for formatting values in JSP.

In this Session, you will learn to:

- Explain the concept of internationalization
- Explain the concept of localization
- Explain the role of Unicode character set in internationalization
- Explain the resource bundling mechanism and resource bundle for various locales
- Explain how to format dates in servlets for internationalization
- Explain how to format currency in servlets for internationalization
- Explain how to format numbers in servlets for internationalization
- Explain how to format percentages in servlets for internationalization
- Explain how to format messages in servlets for internationalization
- Explain various tags available in the JSTL internationalization tag library
- Explain how to format dates and currencies using JSTL I18N tags
- Explain how to format percentages and messages using JSTL I18N tags

14.1 Introduction

Internationalization can be defined as the method of designing an application that can be adapted to a region or a language without much change in the technology.

It helps in creating internationalized Web applications that standardize formatted numeric and date-time output. An internationalized application supports multiple natural languages.

14.1.1 Localization

Localization is a process of making a product or service language, culture and local 'look-and-feel' specific. Localizing a product does not only mean a language translation but also formatting of time, currencies, messages, and dates.

A Locale is a simple object, which identifies a specific language and a geographic region. To create international Java applications, the use of `java.util.Locale` class is a must. Locales are used in entire Java class libraries data for formatting and customization.

14.1.2 Unicode

As there are numerous geographical areas with various languages and conventions, a uniform standard is required for encoding these diversities. Unicode standard serves this purpose.

Unicode is a coding system that has codes for all the major language of the world. The coding system also includes punctuation marks, symbols, and classical as well as historical texts of many written languages.

Unicode in java is a 16-bit character encoding. By default java uses UCS-2. This is a fixed-width two byte encoding that simply encodes each code point from U+0000 to U+FFFF as itself. It has the advantage of being simple and fast but has the disadvantage of being byte-order dependent and of being incompatible with 7-bit ASCII. It allows java to handle international characters for most of the languages of the world.

Figure 14.1 depicts the Unicode table.



Figure 14.1: Unicode Table

14.1.3 Locale

An internationalized application can be adapted to any locale. The applications can be localized using resource bundles, which contain locale-specific objects. This helps in coding programs, which are independent of the user's locale. Locales represent specific geographical, political, or cultural region.

The locale is denoted by the standard format `xx_YY`, where, `xx` stands for two-letter language code in lower case, and `YY` stands for two-letter country code in upper case.

Some of the examples of locales are: zh_CN for Shanghai, China, ko_KR for Seoul, Korea, and en_US for English, US.

Figure 14.2 depicts the use of various locales.



Figure 14.2: Locales

14.2 Internationalizing Servlets

Content in Web application can be displayed in any particular language and this can be done easily from property files, which is also called the message resources.

Internationalization of an application can be achieved with the help of resource bundles. They are especially important for developing applications for different languages. Resource bundles contain locale-specific data.

Resource bundle is a set of related classes that inherit from the `ResourceBundle` . The subclass of `ResourceBundle` has the same base name with an additional component that identifies locales. For example, if resource bundle is named `DemoResources` and along with it, locale-specific classes can be related as in `DemoResources_en_US` .

The resource bundling helps to build server-side code that gives output based on the location and language of the user. It makes the job easier by avoiding the writing of multiple version of a class for different locales.

There are several methods in `ResourceBundle` class as follows:

- `getBundle()`

In order to get resource bundle for getting the locale-specific data, the `getBundle()` method is used. This method gets a resource bundle using the specified base name, the default locale and the class loader of the caller.

Syntax:

```
public static final ResourceBundle getBundle (String basename)
```

where,

- baseName is the name of the resource bundle.

Syntax:

```
public static final ResourceBundle getBundle (String baseName,  
Locale locale)
```

where,

- baseName is the base name of the resource bundle, a fully qualified class name.
- locale is the locale for which a resource bundle is desired.

Code Snippet 1 demonstrates the method to gets a resource bundle using the specified base name and locale.

Code Snippet 1:

```
/* This snippet creates Resource Bundle by invoking getBundle ()  
method, specifying the base name */  
  
ResourceBundle labels = ResourceBundle.  
getBundle ("DemoLabelsBundle");
```

□ getKeys()

This method returns an enumeration of the keys. The keys present in the property file are returned using this method.

Syntax:

```
public abstract Enumeration getKeys ()
```

Code Snippet 2 shows the use of the getKeys () method.

Code Snippet 2:

```
// Displays the value for the keys  
  
static void iterateKeys (Locale currentLocale) {  
  
    ResourceBundle labels = ResourceBundle.getBundle ("LabelsBundl  
e", currentLocale);  
  
    Enumeration bundleKeys = labels.getKeys ();  
  
    while (bundleKeys.hasMoreElements ()) {  
  
        String key = (String) bundleKeys.nextElement ();  
  
        String value = labels.getString (key);
```

```

        System.out.println("key=" + key + ", " + "value=" + value);
    }
}

```

□ **getLocale()**

This method returns the locale of this resource bundle. This method can be used after the calling of `getBundle()` to check whether the returned resource bundle really corresponds to the requested locale or not. It returns the locale for the current resource bundle.

Syntax:

```
public Locale getLocale()
```

Code Snippet 3 shows the use of the `getLocale()` method.

Code Snippet 3:

```

// Gets the user's Locale
Locale locale = request.getLocale();
ResourceBundle bundle = ResourceBundle.getBundle("i18n.
WelcomeDemoBundle", locale);
String welcome = bundle.getString("DemoWelcome");

```

□ **getObject()**

This method gets an object for the given key from the resource bundle. It tries to get the object from the resource bundle and if it fails, the parent resource bundle is called using parent's `getObject()` method. It throws a `MissingResourceException` if it fails.

Syntax:

```
public final Object getObject (String key)
```

where,

- key is the key for the desired object.

Example: `int[] myDemoIntegers = (int[]) myDemoResources.getObject ("intList");`

□ **getString()**

This method returns a string for the given key from the resource bundle or one of its parents.

Syntax:

```
public final String getString(String key)
```

where,

- key is the key for the desired string.

Code Snippet 4 shows the use of `getString()` method.

Code Snippet 4:

```
/** Retrieves the translated value from the ResourceBundle by
invoking the getString method as follows **/
String value = labels.getString(key);
```

14.3 Formatting in Servlets

The formatting of the numbers, currency, date, and percentage helps the programmer to format these values based on the locale or the region of the user.

14.3.1 Formatting Dates

There are several formats in which dates can be displayed. They are as follows:

Predefined Formats

The `DateFormat` style is predefined and locale-specific and is easy to use. The styles are as follows:

- **SHORT** - it is completely numeric, such as 11.14.50 or 3:30pm.
- **MEDIUM** - it is longer, such as Jan 10, 1954.
- **LONG** - it is longer, such as January 10, 1954 or 3:50:32pm.
- **FULL** - it is completely specified, such as Tuesday, April 14, 1954 AD or 3:50:42pm PST.

There are two steps in formatting of date using the `DateFormat` class. The first step is to create a formatter with the `getDateInstance()` method. The second step consists of invoking the `format()` method, which returns formatted date in the form of string.

Customizing Formats

Most of the time, the predefined formats are enough but sometime customized format is required. To do that `SimpleDateFormat` class is used. A `SimpleDateFormat` object is created with specified pattern strings.

The format for the date is determined by the contents of the pattern string. The code in the snippet formats the date as per the pattern string. The formatted date is contained in string, which returned by the `format()` method.

There are several classes for formatting dates. They are as follows:

DateFormat

This is an abstract class, which formats and parses the dates and time in a language independent manner.

Code Snippet 5 demonstrates how to format the date.

Code Snippet 5:

```
// Snippet formats the date
String date = DateFormat.getDateInstance(DateFormat.FULL,
    DateFormat.SHORT, locale).format(new Date());
```

DateFormat.Field

This is a nested class that is used as attribute keys in the `AttributedCharacterIterator`.

The `AttributedCharacterIterator` allows iteration through both text and related attribute information. It is returned from `DateFormat.formatToCharacterIterator` and is also used as field identifiers in `FieldPosition`.

DateFormatSymbol

It is a public class, which encapsulates localizable date-time formatting data, such as names of months, days of week, and the time zone data. Both `DateFormat` class and `SimpleDateFormat` class use `DateFormatSymbols` to encapsulate localizable date-time formatting data information.

DateFormatter

The `DateFormatter` class formats the date as per the format of the current locale.

SimpleDateFormat

The `SimpleDateFormat` is a class that formats and parses dates in a locale-sensitive manner.

Code Snippet 6 demonstrates the use of the `SimpleDateFormat` class.

Code Snippet 6:

```
// Formats the date as per the specified style
SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd",
    Locale.US);
```

Figure 14.3 depicts the various date formats.

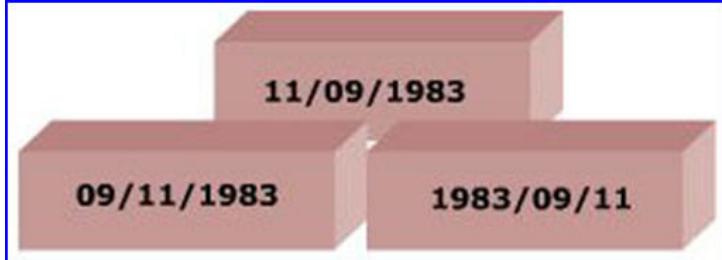


Figure 14.3: Date Formats

14.3.2 Formatting Currency

The formatting of the currency is necessary so that the users can be benefited by the availability of the locale specific formatting. Conversion of one currency to another requires some conversion in value. The formatting of currency value is much more involved compared to formatting date and time. Similarly, formatting of numbers as per the locale or particular region is also very necessary.

Some of the classes used for formatting currency are as follows:

- **Currency**

This class represents currency. The currencies are identified by ISO 4217 currency codes.

- **NumberFormat**

This is an abstract base class for all number formats. `NumberFormat` provides methods to determine the number formats of the locales and their name. It helps to format and parse numbers for any locales.

The class `Currency` represents currencies. These currencies are identified by their ISO 4217 currency codes.

Some of the methods of currency class are as follows:

- **getCurrencyCode()**

This method gets the currency code of the currency as per the ISO 4217 codes.

Syntax:

```
public String getCurrencyCode ()
```

Code Snippet 7 shows the use of the `getCurrencyCode()` method.

Code Snippet 7:

```
/** This snippet returns the currency code of the currency as per  
the ISO 4217 codes **/  
  
public String getCurrencyCode () {  
    return currency.getCurrencyCode () ;  
}
```

□ **`getSymbol()`**

This method gets the currency symbol for the default locale.

Syntax:

```
public String getSymbol ()
```

Code Snippet 8 shows the use of `getSymbol()` method.

Code Snippet 8:

```
// This snippet gets the currency symbol for default locale  
  
public String getCurrencySymbol () {  
    return currency.getSymbol () ;  
}
```

□ **`getInstance(Locale locale)`**

This method returns the `Currency` instance for the country of the specified locale.

Syntax:

```
public static Currency getInstance (Locale locale)
```

where,

- `locale` is the locale representing the country for which a `Currency` instance is required.

Code Snippet 9 shows the use of `getInstance()` method.

Code Snippet 9:

```
// This snippet gets the currency instance
import java.text.NumberFormat;
import java.util.Currency;
public class CurrencyClass {
    public static void main ( String args [] ) {
        // Returns locale-specific currency instance
        Currency localeCurrency = Currency.getInstance ( locale );
    }
}
```

14.3.3 Formatting Numbers

Some of the methods of `NumberFormat` class are as follows:

- **`format(double number)`**

This method is the specialization of the format.

Syntax:

```
public final String format ( double number )
```

Code Snippet 10 demonstrates the use of `format()` method.

Code Snippet 10:

```
// This snippet formats the number in percentage
NumberFormat nft = NumberFormat.getPercentInstance ( locale );
String formatted = nft.format ( 0.51 );
```

- **`getCurrency()`**

This method provides the number format while formatting currency values. The value derived initially is locale dependent. If no valid currency is determined and no currency has been set using `setCurrency` the returned value may be `null`.

Syntax:

```
public Currency getCurrency ()
```

Code Snippet 11 demonstrates the use of `getCurrency()` method.

Code Snippet 11:

```
// This snippet display the currency as per the locale  
import java.text.NumberFormat;  
import java.util.Currency;  
public class CurrencyClass{  
    public static void main ( String args [] ) {  
        NumberFormat formatter = NumberFormat.getInstance ( ) ;  
        System.out.println ( formatter.getCurrency ( ) ) ;  
    }  
}
```

`getInstance()`

This method returns the default number format for the current default locale.

Syntax:

```
public static final NumberFormat getInstance ()
```

Code Snippet 12 demonstrates the use of `getInstance()` method.

Code Snippet 12:

```
// This snippet display the currency as per the locale  
import java.text.NumberFormat;  
import java.util.Currency;  
public class CurrencyClass{  
    public static void main ( String args [] ) {  
        NumberFormat formatter = NumberFormat.getInstance ();  
        System.out.println ( formatter.getCurrency () ) ;  
    }  
}
```

`parse()`

This method parses text from the beginning of the specified string to produce a number.

Syntax:

```
public Number parse ( String str ) throws ParseException
```

where,

- str is a string whose beginning should be parsed.

For example,

```
NumberFormat nf = NumberFormat.getInstance();
```

```
Number myDemoNumber = nf.parse(myDemoString);
```

□ setCurrency()

This method sets the currency used by the number format when formatting currency values. This does not update the minimum or maximum number of fraction digits used by the number format.

Syntax:

```
public void setCurrency(Currency currency)
```

where,

- currency is the new currency to be used by this number format.

For example,

```
// sets the currency to new amountCurrency
```

```
NumberFormat formatter = NumberFormat.getInstance();
```

```
formatter.setCurrency(amountCurrency);
```

14.3.4 Formatting Percentages

Data such as dates, currencies, and numbers are locale dependent. They must conform to the end user language and region. Therefore, formatting is necessary.

The format for displaying percentages can be changed using `getPercentInstance()` method of `NumberFormat` class. For example, with this format a fraction as 0.82 can be displayed as 82%.

There are two methods for formatting percentage as follows:

□ getPercentInstance()

This method returns a percentage format for the current default locale.

Syntax:

```
public static final NumberFormat getPercentInstance()
```

□ getPercentInstance(Locale inLocale)

This method returns a percentage format for the specified locale.

Syntax:

```
public static NumberFormat getPercentInstance(Locale inLocale)
```

where,

- `inLocale` is the specified locale according to which percentage is to be formatted.

Figure 14.4 depicts the code for formatting of percentages.

```
static public void displayPercent(Locale currentLocale) {
    Double percent = new Double(0.75);
    NumberFormat percentFormatter;
    String percentOut;

    percentFormatter = NumberFormat.getPercentInstance(currentLocale);
    percentOut = percentFormatter.format(percent);
    System.out.println(percentOut + " " + currentLocale.toString());
}
```

Output:

75% en_US

Figure 14.4: Formatting Percentages

14.3.5 Formatting Messages

Message formatting is required so that the end user is benefited. The formatting provides message in language neutral way. Formatting also helps provide messages in user's language.

In order to format a message the `MessageFormat` object is used. The array of objects using the format specifiers embedded in the pattern is formatted by `MessageFormat.format()` method. It returns the result as a `StringBuffer`.

The classes for formatting the message are as follows:

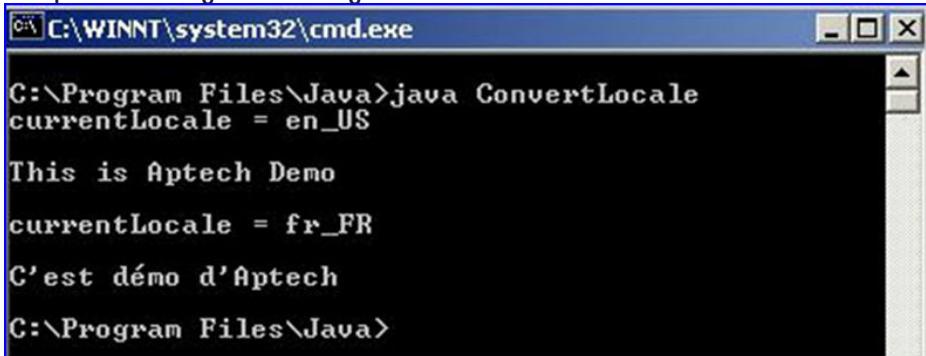
□ **MessageFormat**

This class provides a means to generate related messages in a language-neutral way. `MessageFormat` takes a set of objects, formats them, then inserts the formatted strings into the pattern at the appropriate places. It can also be used to construct messages displayed for end users.

□ **MessageFormat.Field**

This class defines constants that are used as attribute keys in the `AttributedCharacterIterator` object returned from `MessageFormat.formatToCharacterIterator()`.

Figure 14.5 depicts message formatting.



The screenshot shows a Windows Command Prompt window titled 'C:\WINNT\system32\cmd.exe'. The command entered is 'java ConvertLocale'. The output displays two locale configurations: 'currentLocale = en_US' followed by 'This is Aptech Demo' and 'currentLocale = fr_FR' followed by 'C'est démo d'Aptech'. The prompt at the end is 'C:\Program Files\Java>'. This demonstrates how Java's `MessageFormat` class can be used to format messages based on the current locale.

Figure 14.5: Formatting a Message

14.4 Internationalizing JSP Pages

You may need to apply various internationalization formats while creating Web applications using JSP. For this, JSP Standard Tag Library (JSTL) provides a set of internationalization or I18N tags that are used for applying various internationalization formats.

The internationalization or I18n tag library helps reduce and manage the complexities of internationalized applications.

There are several tags available in I18n tag library that are as follows:

formatDate

This tag formats a date value as a date using a locale. A style or pattern for the presentation of the date can be specified.

formatNumber

This tag formats a number as a currency using a locale. The default text is used if the currency value is null. If no locale is specified, the parent `<i18n:locale>` tag is used.

message

This tag retrieves a message from a resource bundle and optionally uses the `java.util.MessageFormat` class to format the message. The arguments to `MessageFormat` can be supplied in the form of an object array or as sub tags within the message tag body.

14.4.1 Formatting Dates in JSP

The tag `<fmt:formatDate>` is used for formatting dates and/or time in JSP for internationalization. The `value` attribute or the body content of the `<fmt:formatDate>` tag formats the date value. The formatted date is written to the JSP's writer. The value can also be stored in a string named `var` and an optional `scope` attribute.

□ **<fmt:formatDate>**

Syntax:

```
<fmt:formatDate value="date" [type="{time|date|both}"]  
[dateStyle="{default|short|medium|long|full}"]  
[timeStyle="{default|short|medium|long|full}"]  
[pattern="customPattern"]  
[TimeZone="TimeZone"]  
[var="varName"]  
[scope="{page|request|session|application}"]/>
```

where,

- `value` is the date and/or the time to be formatted.
- `type` specifies whether time, date or both are to be formatted.
- `dateStyle` is the predefined formatting style for dates.
- `timeStyle` is the predefined formatting style for times.
- `pattern` is custom formatting style for dates and times.
- `TimeZone` is the time zone in which to represent the formatted time.

Example: <fmt:formatDate value="\${FinishDate}" type="date" dateStyle="full"/>

14.4.2 Formatting Currencies in JSP

The currencies can be formatted in JSP for internationalization using JSTL I18N tags. The `<fmt:setLocale>` stores the specified locale in the `javax.servlet.jsp.jstl.fmt.locale` configuration variable. The formatting is done as per the set locale. The tag `<fmt:formatNumber>` can be used to format the currencies.

The number is specified to be formatted either with an EL expression in `value` attribute or as the tag's body content. The desired formatting is specified by the `type` attribute.

□ **<fmt:setLocale>**

Syntax:

```
<fmt:setLocale value="locale" [variant="variant"]  
[scope="{page|request|session|application}"]/>
```

where,

- `value` is a string value, which is interpreted as the printable representation of a locale. Language and country codes must be separated by hyphen ('-') or underscore ('_').
- `variant` is vendor or browser-specific variant.
- `scope` is the scope of the locale configuration variable.

□ `<fmt:formatNumber>`

Syntax:

```
<fmt:formatNumber value="numericValue"
  [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [currencyCode="currencyCode"]
  [currencySymbol="currencySymbol"]
  [groupingUsed="{true|false}"]
  [maxIntegerDigits="maxIntegerDigits"]
  [minIntegerDigits="minIntegerDigits"]
  [maxFractionDigits="maxFractionDigits"]
  [minFractionDigits="minFractionDigits"]
  [var="varName"]
  [scope="{page|request|session|application}"]/>
```

where,

- `value` is the numeric value to be formatted.
- `type` specifies whether the value is to be formatted as number, currency, or percentage.
- `pattern` is custom formatting pattern.
- `currencyCode` is ISO 4217 currency code. Applied only when formatting currencies; ignored otherwise.
- `currencySymbol` is the currency symbol. Applied only when formatting currencies; ignored otherwise.
- `groupingUsed` specifies whether the formatted output will contain any grouping separators.
- `maxIntegerDigits` maximum number of digits in the integer portion.

- `minIntegerDigits` minimum number of digits in the integer portion.
- `maxFractionDigits` maximum number of digits in the fractional portion.
- `minFractionDigits` minimum number of digits in the fractional portion.
- `var` is name of the exported scoped variable.
- `scope` scope of var.

Code Snippet 13 demonstrates the formatting of currency in JSP.

Code Snippet 13:

```
//formatting for currency
<fmt:setLocale value="en_GB"/>
Formatting salary with Locale <B>en_GB</B> becomes :
<fmt:formatNumber type="currency" value="${salary}" /><BR>
```

14.4.3 Formatting Percentages in JSP

The `<fmt:formatNumber>` tag formats a number in integer, decimal, currency, and percentage. By specifying the `type` attribute in `<fmt:formatNumber>` percentage of a number can be obtained. This happens when the value is multiplied with hundred.

Example: `<fmt:formatNumber value="0.82" type="percent"/>`

14.4.4 Formatting Messages in JSP

The `<fmt:message>` tag retrieves a message from a resource bundle and optionally uses the `java.util.MessageFormat` class to format the message. The `key` attribute specifies the message key. If the `<fmt:message>` tag occurs within a `<fmt:bundle>` tag, the key is appended to the bundle's prefix, if there is one.

If the `<fmt:message>` tag occurs outside of a `<fmt:setbundle>` tag, the `bundle` attribute must be present and must be an expression that evaluates to a `LocalizationContext` object. Most often, a variable is initialized with the `<fmt:setBundle>` tag.

Syntax:

```
<fmt:message key="messageKey"
[bundle="resourceBundle"] [var="varName"]
[scope="{page|request|session|application}"]/>
```

where,

- `key` is the message key to be looked up.
- `bundle` is the localization context in whose resource bundle the message key is looked up.
- `var` is the name of the exported scoped variable string.
- `scope` is the scope of `var`.

Code Snippet 14 demonstrates the formatting of messages in JSP.

Code Snippet 14:

```
//this snippet formats the message//  
<fmt:message key="welcome">  
  <fmt:param value="${userNameString}" />  
</fmt:message>
```

Check Your Progress

1. Which among the following statements relating to the internationalization and localization are true?

(A)	Internationalization can be defined as the method of designing an application that can be adapted to only English speaking region without much change in the technology.
(B)	Internationalization helps in creating Web applications that overlooks formatting of numeric and date-time output.
(C)	Localization is a process of making a product or service language, culture and local 'look-and-feel' specific.
(D)	A Locale is a simple object, which identifies a specific language and a geographic region.
(E)	Locales are used in entire Java class libraries data for formatting and customization.

2. Which among the following statements relating to the Unicode and resource bundle are true?

(A)	Unicode is a coding system that has codes for all the major language of the world.
(B)	Unicode coding system includes punctuation marks, symbols, and classical as well as historical texts of many written languages.
(C)	Resource bundle helps in a great way in internationalization as it contains data, which are very locale specific.
(D)	The applications can be internationalized using the resource bundles, which contain locale-specific objects.
(E)	In a resource bundle, the locale is denoted by the standard format xx_YY. Where, xx stands for two-letter country code in lower case and YY stands for two-letter language code in upper case.

3. Which among the following options return the enumeration of keys present in the property file?

(A)	<pre>static void iterateKeys(Locale currentLocale) { ResourceBundle labels = ResourceBundle.getBundle("LabelsBundle",currentLocale); while (bundleKeys.hasMoreElements()) { String key = (String)bundleKeys.nextElement(); String value = labels.getString(key); System.out.println("key = " + key + ", " +"value = " + value); } }</pre>
-----	---

Check Your Progress

(B)

```
static void iterateKeys(Locale currentLocale) {
    ResourceBundle labels =
        ResourceBundle.
    getBundle("LabelsBundle",currentLocale);
    Enumeration bundleKeys = labels.getKeys();
    while (bundleKeys.hasMoreElements()) {
        String key = (String)bundleKeys.
    nextElement();
        String value = labels.getString(key);
        System.out.println("key = " + key + ", "
    +"value = " + value);
    }
}
```

(C)

```
static void iterateKeys(Locale currentLocale) {
    ResourceBundle labels =
        ResourceBundle.
    getBundle("LabelsBundle",currentLocale);
    Enumeration bundleKeys = labels.getKeys();
    while (bundleKeys.hasMoreElements()) {
        String key = (String)bundleKeys.
    nextElement();
        System.out.println("key = " + key + ", "
    +"value = " + value);
    }
}
```

(D)

```
static void iterateKeys(Locale currentLocale) {
    Enumeration bundleKeys = labels.getKeys();
    while (bundleKeys.hasMoreElements()) {
        String key = (String)bundleKeys.
    nextElement();
        String value = labels.getString(key);
        System.out.println("key = " + key + ", "
    +"value = " + value);
    }
}
```

4. Match the classes for formatting dates against their corresponding description.

	Description	Class
(A)	This is an abstract class, which formats and parses the dates and time in a language independent manner.	(1) DateFormatSymbols
(B)	It is public class, which encapsulates localizable date-time formatting data, such as names of months, days of week, and the time zone data.	(2) SimpleDateFormat

Check Your Progress

(C)	This class formats the date as per format of current locale.	(3)	DateFormat
(D)	This class allows formatting and parsing dates in a locale-sensitive manner.	(4)	DateFormatter
(A)	A-3, B-1, C-4, D-2	(C)	A-2, B-3, C-4, D-1
(B)	A-4, B-1, C-2, D-3	(D)	A-4, B-3, C-2, D-2

5. Match the methods for formatting currencies against their corresponding description.

Description		Method	
(A)	This method gets the currency code of the currency as per the ISO 4217 codes.	(1)	getCurrency()
(B)	This method provides the number format while formatting currency values.	(2)	getCurrencyCode()
(C)	This method returns the default number format for the current default locale.	(3)	getInstance (Locale locale)
(D)	This method returns a Currency instance for the specified locale.	(4)	getInstance()
(A)	A-3, B-1, C-4, D-2	(C)	A-2, B-3, C-4, D-1
(B)	A-4, B-1, C-2, D-3	(D)	A-4, B-3, C-2, D-2

6. Which of the following options will output 277%?

(A)	double d = 2.769351; String P1= NumberFormat. getPercentInstance(). format(d); System.out.println(P1);	(C)	double d = 2.769351; String P1= NumberFormat. getPercent Instance(locale); System.out. println(P1);
-----	---	-----	---

Check Your Progress

(B)

```
double d = 2.769351;
String P1=
NumberFormat.
getInstance
(locale).format(d);
System.out.println(P1);
```

(D)

```
double d = 2.769351;
String P1=
NumberFormat.
getInstance();
System.out.
println(P1)
```

7. Which among the following statements relating formatting message is incorrect?

(A)

MessageFormat class provides a means to generate related messages in a language-neutral way.

(C)

In MessageFormat.Field the array of objects using the format specifiers embedded in the pattern is formatted by MessageFormat.format() method.

(B)

MessageFormat takes a set of objects, formats them, then inserts the formatted strings into the pattern at the appropriate places.

(D)

MessageFormat.Field class defines constants that are used as attribute keys in the AttributedCharacterIterator returned from MessageFormat.formatToCharacterIterator.

8. Which among the following options is not a JSTL internationalization tag?

(A)

formatDate

(C)

formatCurrency

(B)

formatNumber

(D)

message

Answer

1.	B
2.	A
3.	B
4.	A
5.	C
6.	C
7.	C
8.	C

Summary

- Internationalization can be defined as the method of designing an application that can be adapted to a region or a language without much change in the technology
- Localization is a process of making a product or service, language, culture, and local 'look-and-feel' specific.
- Internationalization of server-side code reduces the task of writing multiple versions of a class for different locales.
- Resource bundles are used to achieve the locale specific output.
- Internationalization requires formatting of dates, numbers, currencies, and messages with the help of resource bundle.
- The internationalization or I18N tags in JSTL help to reduce and manage the complexities of internationalized applications. There are several tags available in I18N.



Visit

Frequently Asked Questions

@

www.onlinevarsity.com



Welcome to the Session, **Securing Web Applications**.

This session explains the concepts of security in Web application. The session describes about the various authentication mechanisms and security methods used in securing Java Web components.

In this Session, you will learn to:

- Explain the need and features of securing Web applications
- Describe the HTTP basic, digest, client, and form-based authentication method of ensuring security
- Explain how to configure users in Tomcat
- Explain how to specify authentication mechanisms using web.xml
- Describe the seven steps to implement declarative security
- Explain the concept and five steps to implement programmatic security
- Describe the HttpServletRequest methods for identifying users

15.1 Introduction

A Web application is an application, which is accessed using a Web browser over a network, such as Intranet or the Internet. The reason of popularity of Web application is the ability to maintain the application and easy accessibility to information.

Easier accessibility to Web applications opens door to attackers to access or modify the confidential information. Hence, it is necessary to secure the Web application to keep the information safe using some of the security mechanisms.

Figure 15.1 depicts unauthorized access on the Internet.

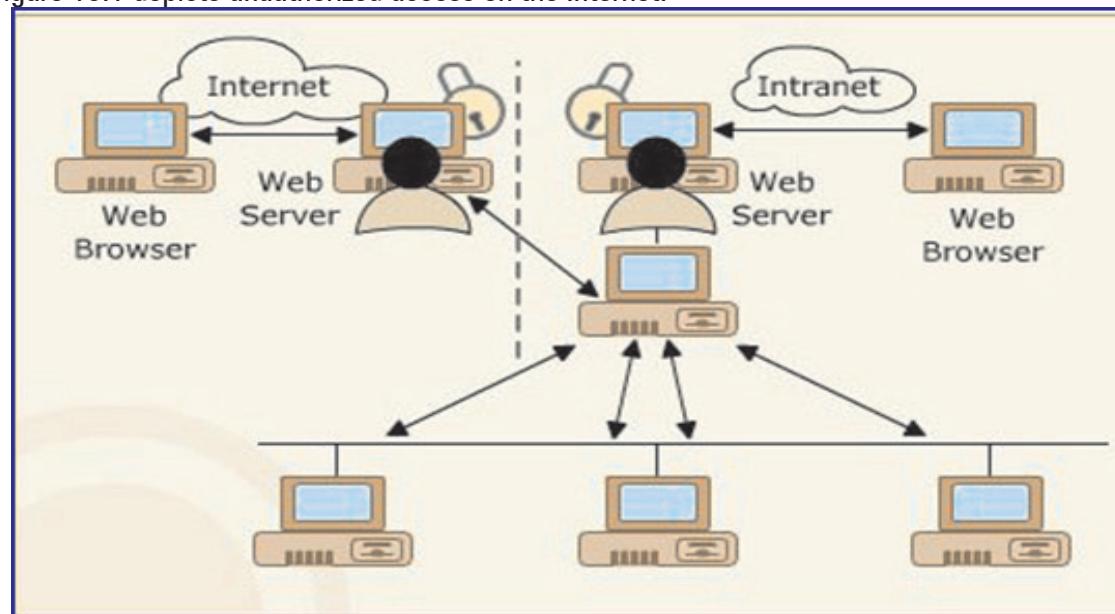


Figure 15.1: Unauthorized Access on the Internet

Security is the constraint that restrict users from accessing sensitive information freely. However, few pre-defined users can access sensitive information.

There are three important issues to be considered in case of security:

- Authentication – This enables client verification through correct username and password.
- Confidentiality – This keeps information hidden from people other than the involved client and server.
- Integrity – This ensures that the content of the communication is not changed during transmission.

15.1.1 Understanding Authentication and Authorization

When the user accesses the Web application resources, he/she needs to be identified. The process of identifying the user allows the Web application to understand the following:

- Identity of the user
- Operations that can be performed by the user
- Maintaining integrity and confidentiality of the accessed resources

Java platform supports various runtime security mechanisms that can be performed while responding to a method invocation by the client.

Figure 15.2 shows the various runtime security mechanism applied in a Web application.

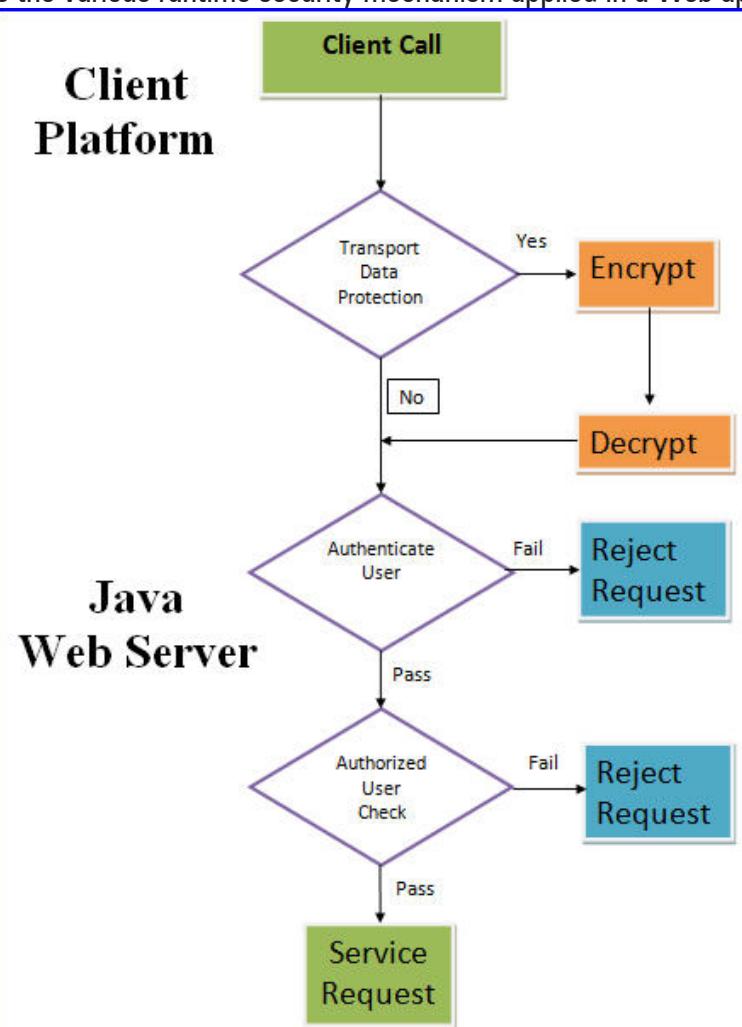


Figure 15.2: Security Mechanisms in Web Application

As shown in figure 15.2, there are various security mechanisms that are applied at the client as well as server-side to protect the Web application resources from wrong users.

- **Transport-Level protection** - Transportation security ensures that a protected transport-layer connection is used for all URL patterns and HTTP methods accessed by the client. For example, if users use a normal HTTP connection to access the security constraint URLs, then the Servlet or JSP page must redirect them to HTTPS protocol layer.
- **Authentication** – Authentication is the process of identifying the user and verifying his/her access to the Web application. Most of the Web applications provide username and password system to identify the users. However, digital certificate technology can also be used to handle identification and encryption of information between the Web server and clients.
- **Authorization** – Authorization is the process of allocating permissions to the authenticated users. This means after the user logs in the application with proper credentials, the authorization checks his access to the resources and services provided by the Web application.
Authorization to the users can be enforced by providing the access control mechanism to the users.

15.1.2 Securing Web Applications

Web applications are created by application developers who set up the security for the application by using annotations or deployment descriptors. This information is then used by the application deployers who assign method permissions for security roles, set up authentication and transport mechanisms.

Some of the common terms used in applying security to Web applications are:

- **User** – A user is an identity that has been defined in the Web server. A user is associated with a set of roles that allows him to access all the resources defined by the role.
- **Group** – A group is a set of authorized users classified with common features such as job title or department. Group is defined in the Web server and allows easier access to large number of users. A group is designated for the entire Web server.
- **Role** – A role is a set of permissions that is applied to a resources in an application. A role is associated with a specific application in the Web server.
- **Realm** – The resources can be partitioned based on their authentication scheme or authorization details containing users and groups. A realm is a database of users and groups identifying valid users for the Web applications. It is controlled by authentication policy. Some of the pre-defined realm available on the GlassFish server includes: file, admin-role, and certificate.

15.1.3 Configuring Users in Tomcat

For Tomcat 7, you create a user with the manager-script role and a password for that user.

Figure 15.3 shows how to register the Tomcat server with NetBeans IDE.

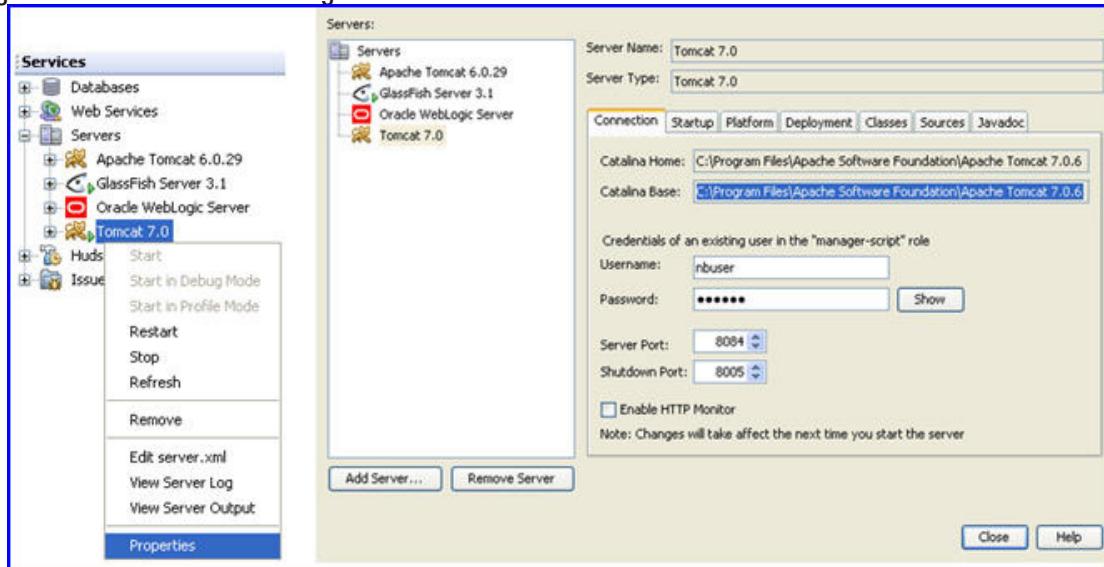


Figure 15.3: Configuring Tomcat Web Server in NetBeans IDE

Tomcat is flexible to configure it as per your needs. By configuring Tomcat, you can create username and password for user authentication. The basic users and roles for the Tomcat server are in `tomcat-users.xml` which is available in `<CATALINA_BASE>\conf` directory under the installed directory of the machine.

Code Snippet 1 demonstrates how to add users and roles in the `tomcat-users.xml` file.

Code Snippet 1:

```
<!-- C:\Program Files\Apache Software Foundation\Tomcat 7.0\conf\tomcat-users.xml file data, -->
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>

// The purpose of role is to give access to a designated user. If the rolename is manager, then the access is given to that area, which is specified for manager.

<role rolename="manager"/>
<role rolename="admin"/>
<user username="admin" password="app_tech" roles="admin,manager"/>
</tomcat-users>
```

```
// To include a new user,
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <user username="Tom" password="tempo" roles="admin,manager"/>
    <user username="admin" password="aptech" roles="admin,manager"/>
</tomcat-users>
```

The roles named **admin** and **manager** are created. Then, the users named **Tom** and **admin** are created with the password and the specified role.\

15.2 Authentication Mechanisms

The security mechanisms are based on authentication and authorization techniques. Authorization mechanism specifies who is allowed to access the resources in the application. Authentication is used to verify the users' identity. It authenticates the user by some reusable methods, which do not require any modification.

Some of the security mechanisms used in Web applications are as follows:

HTTP basic authentication method

In HTTP basic authentication, the Web browser pops-up a login page, which prompts the user for credentials in the form of username and password. A user will be allowed to access information only if the credentials match a stored pair of username and password.

HTTP digest authentication method

In HTTP digest authentication, the server has the password based on a hash function. By using the hash function, the database stored the hash value rather than the password in plain text, so that it is difficult to decipher the data.

Form-based authentication method

In form-based authentication, the browser provides a login form, which appears in response to a request. The user can enter a username and password in the form. If the entered credentials match a stored pair of username and password, the user will be allowed to access information. The user will be directed to an error page if the login fails.

HTTPS client authentication method

In HTTPS client authentication, the browser uses HTTPS protocol, which is identical with the http but the URL indicates to use a default TCP port and an extra authentication layer in between HTTP and TCP. This extra security layer conform the client's authentication.

15.2.1 HTTP Basic Authentication

In HTTP basic authentication, the Web browser pops up a login page in response to a request. The login page prompts the user for credentials, such as username and password. This authentication method works on the assumption that the client-server communication is reliable.

This mechanism does not provide any protection for the information communicated between the client and the server. The credentials are transmitted in plain text, any attacker can easily retrieve the data. To prevent this, the data is encoded before the transmission. The username and password are encoded using base 64 encoding scheme.

The advantage of using basic authentication is that it is browser friendly. However, there is no logout mechanism specified.

15.2.2 HTTP Digest Authentication

The HTTP digest authentication builds over the basic authentication, which authenticate the user identity without sending the password as plain text to the server. The password is first encrypted and then sent.

This type of authentication is based on a hash function. By using the hash function, the database stores a hash value rather than the password. This process is a one-way system because it is difficult to know the original password according to the output. So it is impossible for the attacker to breach this authentication mechanism.

Figure 15.4 depicts digest authentication.

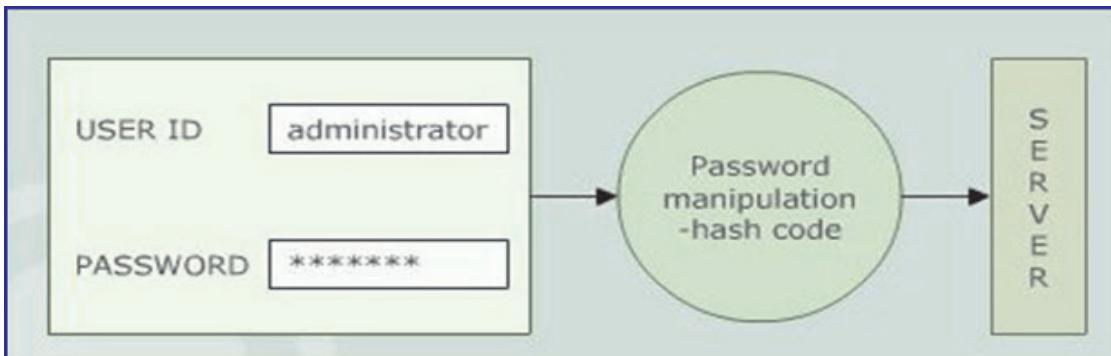


Figure 15.4: Digest Authentication

15.2.3 HTTP Client Authentication

The HTTPS client authentication is a secured client authentication technique, which is based on Public Key Certificates. This authentication is similar to http but uses the https, which is HTTP over Secure Socket Layer (SSL). The URL instructs the browser to use a default TCP port with an extra authentication layer in between HTTP and TCP.

Figure 15.5 depicts HTTPS client authentication.

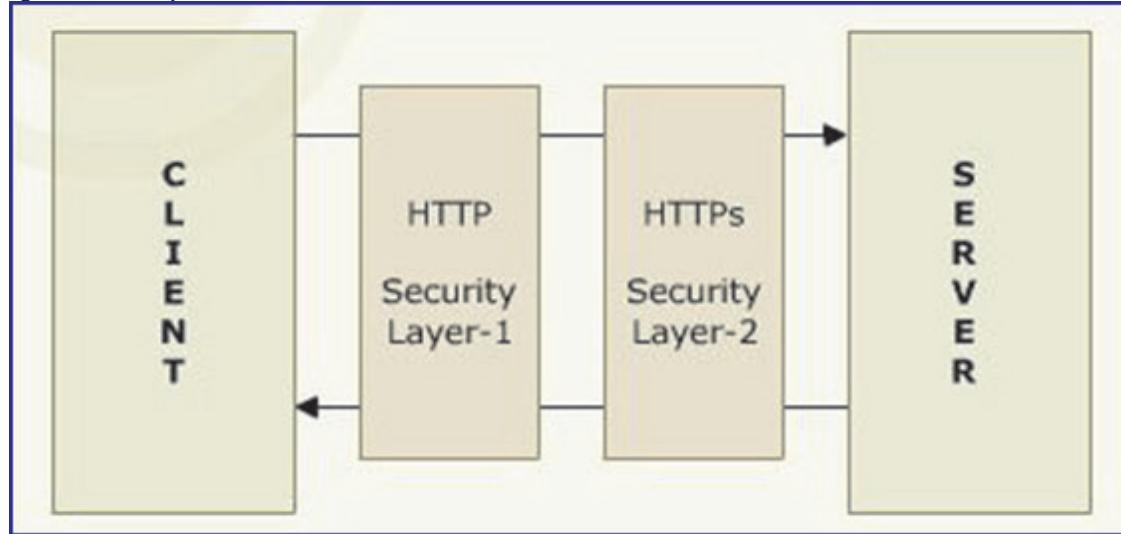


Figure 15.5: HTTPS Client Authentication

15.2.4 Form-based Authentication

The form-based authentication provides a login page, which appears in response to a request from the user. An error page is also available if the login fails from the user side.

After getting the username and password validated by the server, the access is provided to the user. Here, you can define how to protect the Web content for the URLs you are planning to protect.

Figure 15.6 depicts form-based authentication.

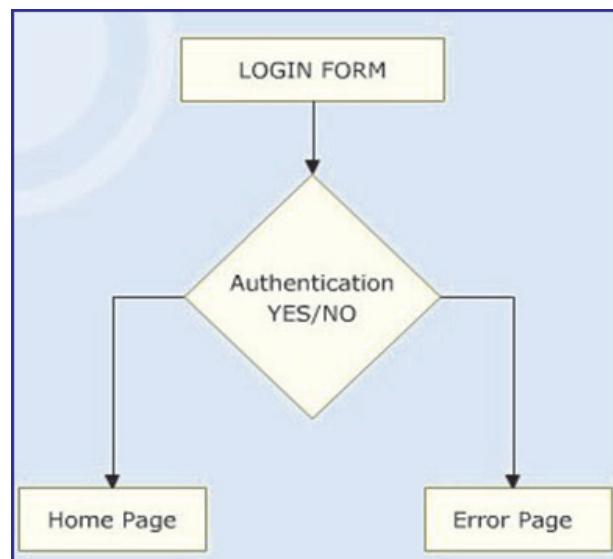


Figure 15.6: Form-based Authentication

15.3 Configuring Authentication in web.xml

web.xml is the most important Java EE configuration file for Java EE Web applications. Web.xml is also known as deployment descriptor. Security configuration in this descriptor leads the operation of Web container security.

You must specify the authentication mechanism in the web.xml file. If you are using basic authentication, the browser verifies the user or it forwards the login page if you are using form-based authentication. To know the use of basic authentication in an application, use the login-config element and its sub elements in the web.xml file. An Excel add-in can be activated by performing the following steps:

The sub elements of login-config element are as follows:

- <auth-method>**
This sub element names the authentication method used in the element.
- <realm-name>**
This sub element names the Web resource where the <login-config> maps.
- <form-login-config>**
This sub element is optional, and is specified only when using form-based authentication that is the <auth-method> value is set to FORM.
- <form-login-page>**
This sub element specifies the form to display when a request is made to a protected Web resource in the Web application. The form-login page is usually an HTML or JSP file, but it can also be a servlet.

Code Snippet 2 uses the form-based authentication mechanism.

Code Snippet 2:

```
<web-app>
. . .
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/login.htm</form-login-page>
        <form-error-page>/loginerror.htm</form-error-page>
    </form-login-config>
</login-config>
. . .
</web-app>
```

15.4 Authorization Strategies

The main purpose of the security model is to control access to business services and resources in the application. The Java security model two strategies for providing access control namely declarative access control and programmatic access control.

15.4.1 Implementing Declarative Security

The declarative security provides security to a resource with the help of the server configuration. Declarative security works as a different layer from the Web component with which it works.

Advantages of declarative security are as follows:

- It gives scope to the programmers to ignore the constraints of the programming environment.
- Updating the mechanism does not require total change in security model.
- It is easily maintainable.

Disadvantages of declarative security are as follows:

- Access is provided to all or denied.
- Access is provided by the server only if the password matches.
- All the pages use same authentication mechanism, which means it cannot use both form-based and basic authentication for the same page.

To implement the declarative security, perform the following eight steps:

1. Set Up Usernames, Passwords, and Roles

When a user tries to access a secured resource in a Web application, which uses form-based authentication, the system uses a form to ask for a username and a password. After verifying the password, the system gives access to the user. Then, it determines the role, such as user, administrator or executive, which is defined by the user.

As discussed in Code Snippet 1, the users and roles are created in tomcat-users.xml file for the Tomcat Web server.

2. Tell the Server That You Are Using Form-Based Authentication and Designate Locations of Login and Login Failure Pages

After validating the user and the role of user, it uses a form-based authentication, which supplies a value for the auth-method sub element, and uses the form-login-config sub element to give the locations of the login and login-failure pages. These pages can consist of either JSP or HTML.

As discussed in Code Snippet 2, the part of web.xml indicates that the container is using form-based authentication.

3. Create a Login Page

The `login-config` element informs the server to use form-based authentication for creating a login page, and also redirect unauthenticated users to a designated error page. The login page requires a form for security check, a text field named `username` and a password field named `password`.

Code Snippet 3 shows the `login.jsp` page.

Code Snippet 3:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>...</TITLE></HEAD>
<BODY>
...
<FORM ACTION="securityServlet" METHOD="POST">
<TABLE>
<TR><TD>User name: <INPUT TYPE="TEXT" NAME="j_username">
<TR><TD>Password: <INPUT TYPE="PASSWORD" NAME="j_password">
<TR><TH><INPUT TYPE="SUBMIT" VALUE="Log In">
</TABLE>
</FORM>
...
</BODY></HTML>
```

4. Create a Page to Report Failed Login Attempts

The `login-failure` page contains a link to an unrestricted section of the Web application. The link shows 'login failed' or 'username and password not found' message.

5. Specifying URLs That Should Be Password Protected

Specifying the URLs and describing the protection they should have are the purposes of the `security-constraint` element. The `security-constraint` element should come before the `login-config` element in `web.xml`, and contains four sub elements, `display-name`, `web-resource-collection`, `auth-constraint`, and `user-data-constraint`.

Code Snippet 4 shows the use of security-constraint.

Code Snippet 4:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
...
<web-app>
<!-- ... -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Sensitive</web-resource-name>
        <url-pattern>/sensitive/*</url-pattern>
    </web-resource-collection>

    <auth-constraint>
        <role-name>administrator</role-name>
        <role-name>executive</role-name>
    </auth-constraint>
</security-constraint>
<login-config>...</login-config>
</web-app>
```

The code instructs the server to require passwords for all URLs of the form `http://localhost/.../sensitive/`. Users who supply passwords and belong to the administrator or executive logical roles should be granted access; all others should be denied access.

The sub-element `web-resource-collection` subelement of `security-constraint` identifies the resources that should be protected. Each `security-constraint` element must contain one or more `web-resource-collection` entries; all other `security-constraint` subelements are optional. The `web-resource-collection` element consists of a `web-resource-name` element that gives an arbitrary identifying name, a `url-pattern` element that identifies the URLs that should be protected, an optional `http-method` element that designates the HTTP commands such as GET, POST, and so on to which the protection is applied. The default is all methods and an optional `description` element providing documentation.

The `web-resource-collection` element designates the URLs that should be protected, whereas the `auth-constraint` element designates the users that should have access to protected resources. It should contain one or more `role-name` elements identifying the class of users that have access and, optionally, a `description` element describing the role.

6. Specifying URLs That Should Be Available Only with SSL

If your server supports SSL, you can define that certain resources are available only through encrypted HTTPS (SSL) connections. Use of SSL does not obstruct the basic way that form-based authentication works. None of your servlets or JSP pages need to be modified or moved to different locations when you enable or disable SSL. This is an extra advantage of declarative security.

The `user-data-constraint` subelement of `security-constraint` can mandate that certain resources be accessed only with SSL. The `user-data-constraint` element, if used, must contain a `transport-guarantee` subelement (with legal values `NONE`, `INTEGRAL`, or `CONFIDENTIAL`) and can optionally contain a `description` element. A value of `NONE` for `transport-guarantee` puts no restrictions on the communication protocol used. Since `NONE` is the default, there is little point in using `user-data-constraint` or `transport-guarantee` if you specify `NONE`. A value of `INTEGRAL` means that the communication must be of a variety that prevents data from being changed in transit without detection. A value of `CONFIDENTIAL` means that the data must be transmitted in a way that prevents anyone who intercepts it from reading it. Although in principle (and perhaps in future HTTP versions) there may be a distinction between `INTEGRAL` and `CONFIDENTIAL`, in current practice they both simply mandate the use of SSL.

Code Snippet 5 instructs the server to permit only https connections to the associated resource.

Code Snippet 5:

```
<security-constraint>
  <!-- ... -->
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

7. Turning Off the Invoker Servlet

The most portable approach of turning off the invoker servlet is to remap the servlet pattern in the Web application so that all requests that include the pattern are sent to the same servlet. To remap the pattern, first create a simple servlet that prints an error message. Then, use the `servlet` and `servlet-mapping` elements to send requests that include the servlet pattern to that servlet.

8. Write Authentication Filter

Write the Servlet filter and mention in `web.xml`. This Filter class filters the HTTP requests based on its type.

15.5 Programmatic Security

When declarative security alone is not sufficient to express the security model of the application, then the programmatic security model can be used by security-aware applications.

15.5.1 Concept of Programmatic Security

Programmatic security authenticates the users and grants access to the users. The servlet or JSP page either authenticates the user or verifies that the user has authenticated earlier, to check the unauthorized access. To ensure the safety of network data, the servlet or JSP page redirects the HTTP connection to HTTPS of the URLs.

The advantages of programmatic security are as follows:

- Ensures total portability
- Allows password matching strategies

The disadvantages are of programmatic security are as follows:

- Much harder to code and maintain
- Every resource must use the code

15.5.2 HttpServletRequest Methods for Identifying User

Some of the methods in the HttpServletRequest interface are used for identifying a user by the help of cookies. Cookies are texts that are sent by the server to a browser, and then sent back unchanged by the browser each time it accesses the server.

There are various methods available in HttpServletRequest interface as follows:

getAuthType()

This method returns the authentication scheme name.

Syntax:

```
public java.lang.String getAuthType()
```

getCookies()

It returns an array, which have all the information of the Cookie objects sent by the client.

Syntax:

```
public Cookie[] getCookies()
```

getHeader()

It returns the value of the header as a String, which is requested.

Syntax:

```
public java.lang.String getHeader(java.lang.String name)
```

 getRemoteUser()

If the user is authenticated it returns the login name of the user, else it returns null.

Syntax:

```
public java.lang.String getRemoteUser()
```

 getRequestedSessionId()

This method returns the session ID that is defined by the client.

Syntax:

```
public java.lang.String getRequestedSessionId()
```

 getSession()

This method returns the current session or creates one if there is no session.

Syntax:

```
public HttpSession getSession()
```

 isUserInRole()

It returns a boolean value, which indicates whether the authenticated user is included in the logical 'role'.

Syntax:

```
public boolean isUserInRole(java.lang.String role)
```

 getUserPrincipal()

This method returns a java.security.Principal object.

Syntax:

```
public java.security.Principal getUserPrincipal()
```

isRequestedSessionIdValid()

It checks the validity of the requested session ID.

Syntax:

```
public boolean isRequestedSessionIdValid()
```

15.5.3 Implementing Programmatic Security

The six steps to implement programmatic security are as follows:

1. Check whether there is an authorization request header

It checks whether the header exists or not.

2. Get the string, which contains the encoded username/password

If the authentication header exists it returns the string, which contains the username and password in base64 encoding format.

3. Reverse the base64 encoding of the username/password string

Reverse the base64 encoding of username or password by using the `decodeBuffer` method of `BASE64Decoder` class.

4. Check the username and password

The common approach of checking the username and password is to use database to get the original username and password. In other way, put the information of the password in the servlet. You will get access if the incoming username and password matches the reference username and password pairs.

5. If authentication fails, send the proper response to the client

It returns a 401 response code, which is an unauthorized one in the form of, `WWW-Authenticate: BASIC realm="some-name"`. This response pop-up a dialog box requesting the user to enter a name and password for some-name.

6. Servlet Annotations

Based on J2EE API, `javax.servlet.annotation` package provides the number of classes and interfaces.

Using of these classes and interfaces declare the filters, servlets, and listeners to authenticate the http requests.

15.6 ServerAuthModule Interface

Different clients sent different requests to server. For application security, it is necessary to validate the client requests. Different type of security methods are used for validating the client requests. `ServerAuthModule` interface helps to validate the client requests.

15.6.1 ServerAuthModule Interface Methods

`ServerAuthModule` interface have five methods that are as follows:

- **Initialize():** This method is used for initialize the authentication module and declare all the necessary objects for authentication. This method has four arguments namely, `messagepolicy1`, `messagepolicy2`, `callbackhandler`, and `Map` objects.
- **getSupportedMessageTypes():** This method returns the array of objects and it contains the messages.
- **ValidateRequest():** This method is call by servlet container and validates the client sent http Servlet requests. This method has argument `messageinfo`, `subject 1` and `subject2` objects.
- **SecureResponse():** This method also called by Servlet container. This method returns the response code.
- **CleanSubject():** This method is used to clean the subject of requests.

Check Your Progress

1. Identify the correct options for different security mechanisms used in Web application.

(A)	In Web application, it is difficult to maintain the data without changing in the client systems through the server.
(B)	In HTTP digest authentication, the server has the password based on a hash function.
(C)	In basic security mechanism, the Web browser never pops-up a login page.
(D)	In form-based authentication, an error page is shown if the login fails from the users' side.
(E)	In HTTPS client authentication, the browser uses a default TCP port and an extra authentication layer in between HTTP and TCP.

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

2. Identify the correct option from the following statements for basic authentication.

(A)	The user credentials are transmitted in plain text, so it is secure from any attacker.
(B)	This mechanism provides protection for the information communicated between client and server.
(C)	The login page prompts the user for credentials, such as username and password.
(D)	In basic authentication the login page directly prompts the user to the home page.

3. Identify the correct options from the following statements for HTTP digest authentication.

(A)	Digest authentication authenticates the user identity without sending the password as plain text to the server.
(B)	The hash value is vulnerable for the attacker to breach.
(C)	In digest authentication, the password is first encrypted and then sent.
(D)	It is quite easy to know the original password according to the output.
(E)	Digest authentication is based on a hash function.

Check Your Progress

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

4. Identify the correct option from the following statements for client authentication.

(A)	This authentication is similar to TCP and Secure Socket Layer (SSL).
(B)	The HTTPS client authentication is a secured client authentication technique.
(C)	HTTPS client authentication based on Private Key Certificate.
(D)	The URL instructs the browser not to use a default HTTP port with an extra authentication layer.

5. Identify the wrong option for configuring and specifying users in Tomcat.

(A)	To configure user in Tomcat, first select the Tomcat 6.0 from Apache Software Foundation in your C: drive.
(B)	In URL, put http://localhost you will get Apache Tomcat home page.
(C)	To know the use of basic authentication in an application, use the login-config element.
(D)	When browser loads a resource, which is secured by web.xml file, the browser responds in different ways.

6. Identify the correct options for declarative security from the following statements.

(A)	The login-config element tells the server to use form-based authentication and to redirect unauthenticated users to a designated error page.
(B)	Use of SSL hampers the basic way that form-based authentication works.
(C)	The login page requires a form for security check, a text field named username and a password field named password.
(D)	Specifying URLs and describing the protection they should have is not the purpose of the security-constraint element.
(E)	In declarative security, any updating does not require the total change in the security model.

Check Your Progress

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

7. Identify the correct option for programmatic security.

(A)	Programmatic security has not authenticated the users it grants access directly to the users.
(B)	The programmatic security allows password matching strategies.
(C)	To check the unauthorized access, the HTTP servlet verifies that the user has been authenticated earlier.
(D)	To ensure the safety of network data, the JSP page redirects the HTTP connection to the password URLs.

8. Identify which is not a correct step for programmatic security.

(A)	It checks whether there is an authorization request header.
(B)	If authentication fails, it redirects the client to the home page.
(C)	It does not check the username and password.
(D)	It uses the base46 encoding of the username/password string.

Answer

1.	C
2.	C
3.	B
4.	B
5.	B
6.	B
7.	B
8.	A

Summary

- A Web application is an application, which is accessed with the help of a Web browser.
- The reason of popularity of Web application is the ability to maintain it without changing the client computers. When you access the Web, hackers can get your information. So to keep your information secret, it is necessary to secure Web applications.
- There are four authentication Mechanisms available. These are HTTP Basic Authentication, HTTP Digest Authentication, HTTPS Client Authentication, and Form-Based Authentication.
- To configure a user in Tomcat, first include the Tomcat 7.0 from Apache Software Foundation. When browser loads a resource, which is secured by web.xml file, the browser responds in two ways.
- The browser challenges the user if you are using basic authentication or forwards the login page if you are using form-based authentication.
- The declarative security provides security to a resource with the help of server configuration.
- Programmatic security authenticates the users and grants access to the users.