Java
for
Web Development

Create Full-Stack Java Applications with Servlets, JSP Pages, MVC Pattern and Database Connectivity

SARIKA AGARWAL
VIVEK GUPTA

bpb

Java for Web Development

Create Full-Stack Java Applications with Servlets, JSP Pages, MVC Pattern and Database Connectivity

Sarika Agarwal

Vivek Gupta

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-55511-430

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

To View Complete
BPB Publications Catalogue
Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My beloved Son

Mr. Kushagra Agarwal

About the Authors

- **Sarika Agarwal** is a professor with 16+ years of experience teaching Java technology Python. She has imparted training on Java, Python, C, C++ for more than 5000 students. Her interest area is Java, Android, machine learning, Natural Language Processing.

  She has cleared many certifications, including SCJP (Sun Certified Java Programmer), OWCD (Oracle Web component Developer), ACAD (Android Certified Application Developer), Android ATC Certified Trainer, MTA certifications (Software Testing Fundamentals, Operating System, Database Management System), IBM RAD certification.

  She has worked in NIIT as a Java Faculty and Engineering College professor. She has published papers in Journals and Scopus Index conferences. She is the author of the Book -Java in Depth.

- **Vivek Gupta** has completed his B.Tech from Uttar Pradesh Technical University in 2006 CDAC (Post Graduate Diploma in Computer Science) in 2006. 15+ years of experience in Java Application support and maintenance at all levels (L1, L2, and L3). Currently, working with Ventiv Technology as Technical Lead. He has worked in HCL, Cognizant Technology Solutions

About the Reviewer

With about six years of experience in teaching, **Ms. Jyoti Kumari** is an Assistant Professor, an enthusiast in learning new technologies. She currently works at Keshav Mahavidyalaya, University of Delhi. She is also pursuing a Ph.D. in Information Technology at Amity University to carry her interest in research work. She also mentors graduate students in their project work such as creating android applications, web development, and so on. She was also a member of the editorial board of the magazine of the technical society of Computer Science department of the college.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my husband, Mr. Amit Kumar, for continuous support and encouraging me to write the book — I could have never completed this book without his support.

I am grateful to the excellent Java Community, from which I have learned and continue to learn a great deal. I would like to thank the reviewers who contributed many suggestions and improvements to my drafts. Thanks also go to many others with whom I have had conversations or email discussions over the course of writing the book. Any errors that remain are, of course, to be laid at my door.

Special thanks to BPB Publications team, for the valuable support provided throughout the entire process.

Finally, the writing of this book has been a great deal of work, and I could not have done it without the constant support of my family. My son, Kushagra Agarwal, has been very understanding, and I am looking forward to spending lots more time with all of them.

Preface

Java, the only pure object-oriented language available today, is now used in almost all applications, from simple home appliances control systems to complex space control systems. It has also revolutionized applications from Intranet to the Internet.

This book aims at imparting expertise in web application development using servlets and JavaServer Pages. You will learn to create a servlet, JSP pages, and connectivity with Database and deploy these applications on the Tomcat Server. Some Interview Questions with answers are also included. This book is meant for anyone who has an interest in Object-Oriented Programming and is aspiring to become a Java Programmer.

The book covers all topics with basic examples and analogies. The book covers all topics related to Servlets and JSP like building GUI applications, reusing JavaBeans in JSP, and using custom tag libraries. This book is written to serve as a textbook for GBTU and for those who want to learn basic and advanced level web application development in java with their efforts.

In each chapter, good worked Examples have been given. Chapter 1 aims to learn the connectivity of the Java Program with the Database; Chapter 2 covers Internationalization, which customizes an application according to specific languages and regions. Chapter 3 to 6 covers the Servlets, Inter Servlet Communications Sessions, etc. Chapter 7 to 10 Covers topics related to JSP pages, Custom tags, Directives, MVC architecture, and many more At the end of each chapter, some interview questions with answers are also given that may be useful to students of any discipline as MCA, B.Tech, M.Tech, M.Sc.

I am sure that the students and the faculty will find this book very useful.

Critical evaluation and suggestions for improving the book will be highly appreciated and gratefully acknowledged.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/0a0b7b

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Java-for-Web-Development**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

1. Database Connectivity 1

Introduction 1

Structure 1

Objectives 2

Database Management 2

ODBC Application Programming Interface (API) 3

JDBC-API 4

Categories of JDBC Drivers 4

JDBC-ODBC bridge + ODBC driver 5

Native-API driver 5

Network protocol driver 6

Thin Driver 6

Querying a database 7

Connecting to a database 7

The connection object 7

Loading the Driver and establishing the connection 7

The JDBC URL 7

A Sample JDBC URL 8

Processing querying in a database 8

Conclusion 19

Multiple Choice Questions 20

Answers 20

Fill in the blanks 21

State True/False 21

Questions 21

Interview Questions 22

2. Internationalization (I18N) 23
Introduction 23
Structure 23
Objective 24
Localization (L10N) 24
Locale 24
Constructors of Locale Class 24
Commonly used methods of Locale class 24
Resource Bundle 25
Constructor of ResourceBundle 26
Methods of ResourceBundle Class 26
Steps to develop the I18N-based application 27
Internationalizing Date and Time (I18N with Date and Time) 28
Methods of java.text.DateFormat 28
Internationalizing with Numbers (I18N with Numbers) ... 30
Conclusion 31
Multiple Choice Questions 32

3. Introduction to Java Servlets 37
Introduction 37
Structure 38
Objectives 38
Webserver 38
Introduction to Servlets 38
Characteristics of Servlets 39
Comparison between Servlets and Applets 39
Comparison between Servlets and other server-side scripting technologies 40
CGI scripts 40
Active Server Pages (ASP) 41
Working of Servlets 41

The GET and POST methods 42
The Javax.servlet package 42
Lifecycle of a Servlet *44*
Servlet Interface 44
Creating a Servlet *45*
Creating the deployment descriptor (web.xml file) *47*
Conclusion 50
Questions 50

4. HTTP Servlet 51
Introduction 51
Structure 51
Objectives 52
HTTP Servlet 52
Need of HTTPServlet class 52
HTTP Request and HTTP Response 52
The GET and POST methods 54
HttpServletRequest Interface 56
Method of ServletRequest Interface *57*
Conclusion 61
Questions 61
Multiple Choice Questions 61

5. Working with Servlet Sessions 63
Introduction 63
Structure 63
Objective 64
Session tracking 64
Techniques to keep track of sessions in servlets 64
URL Rewriting *64*
Hidden Form Fields *68*
Using the HttpSession Interface *71*

Cookies *76*
The javax.servlet.http.Cookie class *76*
Conclusion 81
Questions 81

6. Inter-Servlet Communication 83
Introduction 83
Structure 83
The RequestDispatcher Interface 84
Method of ServletContext Interface 84
Method to get the object of RequestDispatcher 84
Methods of RequestDispatcher interface 85
Implementing Inter servlet communication via a problem statement 86
Tasklist *86*
Client Interface *87*
index.html *87*
Code of FirstServlet *88*
SendRedirect *91*
Difference between forward() and sendRedirect() method 92
Conclusion 92
Questions 93

7. Java Server Pages (JSP) 95
Introduction 95
Structure 95
Objectives 96
Need for JSP 96
Difference between Servlet and JSP 97
Advantages of JSP *97*
The JSP request-response cycle *98*
Lifecycle of JSP *98*

Structure of a JSP Page *99*
The directory structure of the JSP Page *101*
Conclusion 101
Questions 102

8. Comment Tag and Scripting Element 103
Structure 103
Objectives 103
JSP Elements 104
Comment Tag *104*
Scripting Elements 104
Scriptlet tag *104*
Expression Tag *106*
Declaration tag *107*
Implicit Objects 108
Conclusion 116
Questions 116
Select the Correct Option 117

9. JSP Directives 119
Structure 119
Objective 119
Types Of Directives 120
JSP directives 120
The page directive *120*
Implicit Objects 122
contentType *122*
extends *122*
errorPage 122
isErrorPage *122*
import= "package list" *124*
language= "scripting language." *125*

Session=true/false *125*
info= "servlet information" *126*
Buffer 126
isELIgnored= "true/false" *126*
isThreadSafe *127*
autoFlush *127*
The include directive 127
The Taglib directive 128
Conclusion 128
Questions 129

10. JSP Action Element and Custom Tags 131
Introduction 131
Structure 131
Objectives 131
JSP Action Tags 132
jsp:useBean action tag *134*
jsp:setProperty and jsp:getProperty action tags *136*
Jsp:forward action tag *140*
jsp:include action tag *142*
JSP custom tags 147
Custom Tag Library *147*
Need of XML *148*
Custom Tags *150*
The structure of the TLD file *156*
The structure of the JSP File *158*
Expression Language(EL) 160
Model View Controller (MVC) Architecture in JSP 164
MVC Example in JSP *165*
Conclusion 170
Questions 171

11. Introduction to Struts 173
Introduction 173
Structure 174
Objective 174
Features of Struts2 174
Components of Struts2 175
Architecture of Struts2 176
Creating a Struts Application 177
Conclusion 182
Questions 182

Interview Questions 183

Index 189-196

Chapter 1
Database Connectivity

Introduction

Most web-based application programs need to interact with Database Management Systems (DBMS). This DBMS is used to retrieve information from the database repositories by applications. For example, the online shopping mail needs to keep track of its customers and the items sold. A search site like **www.yahoo.com** needs to keep track of the URLs of the web pages visited by the user.

Structure

In this chapter, we will cover the following topics:

- Techniques of database programming using Java
- Types of drivers to connect the Java program with database
- How databases are accessed using the JDBC-ODBC bridge and ODBC drivers

Objectives

In this chapter, you will be able to establish the connection between the database and Java program. You will be able to read and write the data from the database with the full support of the stored procedure and cursor.

Database Management

A database is a collection of related information, and a DBMS is software that provides you with a mechanism to retrieve, modify, and add the data to the database. There are many DBMS/RDBMS products available, for example, MS-Access, MS-SQL Server, Oracle, Sybase, Informix Progress, and Ingress. Each of these **Relational Database Management Systems** (**RDBMS**) stores the data in its form.

For example, MS-Access stores the data in the MDB file format, whereas MS-SQL stores the data in the. DAT file format.

Imagine you have been assigned a task to develop an application for a general store that allows a shop owner to maintain a record of the daily transactions. You design the database, install an MS-SQL server/Oracle on the shop owner's machine, and tell him to use it.

It will be a good idea for you to develop a customized front-end application in which the client is given options to retrieve, add, and modify the data at the touch of the key.

To accomplish this, you should have a mechanism of making the application work with the file format of the database, that is. MDB or. DAT files.

For your application to communicate with the database, it needs to have the following information:

- The location of the database
- The name of the database

Figure 1.1 illustrates a database application architecture:

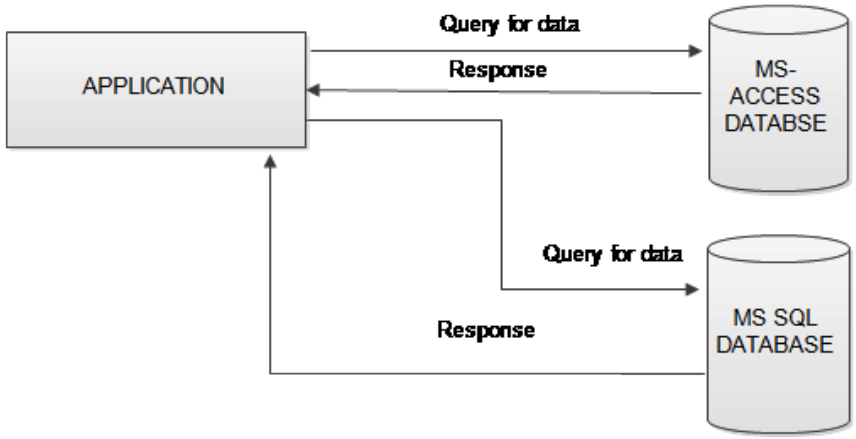


***Figure 1.1**: A database application architecture*

This means that the application you create would be able to work with only one kind of database and will be very difficult to code and port.

The preceding problems are solved by Microsoft's mechanism for efficient communication with the databases called **Open Database Connectivity** (**ODBC**).

ODBC Application Programming Interface (API)

ODBC API is a set of library routines that enable your programs to access a variety of databases. All you need to do is install a DBMS-specific ODBC driver and write your program to interact with a specified ODBC driver. *Figure 1.2* shows that the application communicates with the ODBC Driver Manager, which transfers the query to a database driver for whom the user wants to communicate.

Later, if the database is upgraded to a newer version of RDBMS or ported to a different RDBMS product, you will need to change only the ODBC driver and not the program, as shown in *Figure 1.2*:

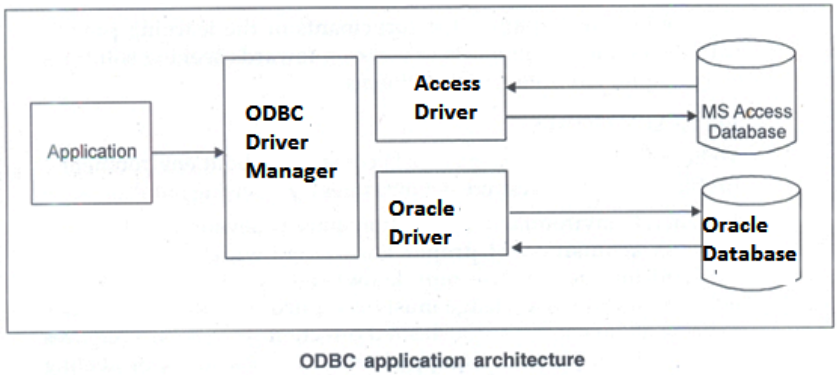


***Figure 1.2**: ODBC application architecture*

JDBC-API

Java Database Connectivity (**JDBC**) provides a database programming API for Java programs. Since the ODBC API is written in the C language and makes use of the pointers and other constructs that Java does not support, a Java program cannot directly communicate with an ODBC driver.

Sun Microsystems created the JDBC-ODBC bridge driver that translates the JDBC-API to the ODBC.API. It is used with the ODBC drivers available.

Categories of JDBC Drivers

There are several categories of the JDBC drivers available, which are as follows:

- JDBC-ODBC bridge + ODBC Driver
- Native API – Partly a Java driver
- JDBC – Net pure Java Drivers / N/W Protocol
- Nature Protocol pure Java drivers / thin Driver

JDBC-ODBC bridge + ODBC driver

The JDBC-ODBC bridge driver uses the ODBC driver to connect to the database. The **JDBC-ODBC Bridge Driver** converts the JDBC method calls into the ODBC function calls. This is now discouraged because of the thin Driver. Refer to *Figure 1.3*, which illustrates the JDBC application architecture:

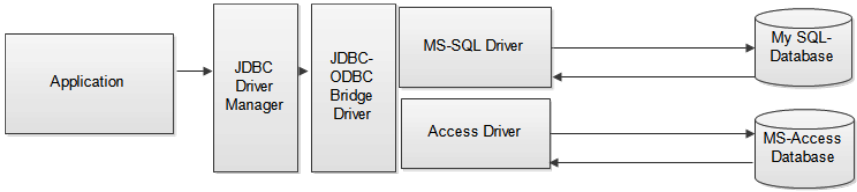


***Figure 1.3**: JDBC Application Architecture*

JDBC Driver Manager

The JDBC Driver Manager is the backbone of the JDBC architecture. The function of the JDBC driver manager is to connect a Java application to the appropriate Driver specified in your Java Program.

JDBC-ODBC Bridge

As a part of JDBC, Sun Microsystems provides a driver to access the ODBC data sources. This Driver is called the JDBC-ODBC bridge. The JDBC-ODBC bridge is implemented as the JDBC ODBC class, and the nature library is used to access the ODBC driver. In the Windows platform, the native library is **JdbcOdbc.dll**

The following are the advantages:

- Easy to use
- It can be easily connected to any database

The following are the disadvantages:

- Performance degraded because the JDBC method call is converted into the ODBC function calls
- The ODBC driver needs to be installed on the client machine

Native-API driver

The Native-API Driver uses the client-side libraries of the database. The Driver converts the JDBC method calls into native calls of the database API. It is not written entirely in Java.

The following is the advantage:

- Performance upgraded, better than the JDBC-ODBC bridge driver

The following are the disadvantages:

- The Native Driver needs to be installed on each client machine
- The vendor-client library needs to be installed on the client machine

Network protocol driver

It uses middleware (application server) that converts the JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in Java.

The following is the advantage:

- No client-side library is required

The following are the disadvantages:

- Network support is required
- Requires database-specific coding to be done in the middle tier
- Maintenance of Network Protocol becomes costly because it requires database-specific coding

Thin Driver

The thin Driver converts the JDBC calls directly into the vendor-specific database protocol. That is why it is known as the thin Driver. It is fully written in the Java language.

The following are the advantages:

- Better performance than all the other drivers
- No software is required at the client-side or server-side

The following is the disadvantage:

- Drivers depend on the database

Querying a database

Now that you have understood the JDBC architecture, you can write a Java application that can work with a database. In this section, you will learn about the packages and classes available in Java that allow you to send the queries to a database and process the query results.

Connecting to a database

The **`java.sql`** package contains the classes that help in connecting to a database, sending embedded SQL statements to the database, and processing the query results.

The connection object

The connection object represents a connection with a database. You may have several connection objects in an application that connects to one or more databases.

Loading the Driver and establishing the connection

To establish a connection with a database, complete the following steps:

1. Register the ODBC JDBC/thin Driver by calling the **`forName()`** method from the Class **`class`**.

   For example, `Class.forName("oracle.JDBC.driver.OracleDriver");`

2. Call the **`getConnection("JDBC URL")`** method from the **`DriverManager`** class.

The **`getConnection()`** method of the Driver Manager class attempts to locate the Driver that can connect to the database represented by the JDBC URL passed to the **`getConnection()`** method.

The JDBC URL

The JDBC-URL is a string that provides a way of identifying a database. A JDBC URL is divided into the following three parts:

```
<protocol>: <sub protocol> : <subname>
```

In this, **<protocol>** in a JDBC URL is always **Jdbc<subprotocol>**, which is the name of the database connectivity mechanism. If the mechanism of retrieving the data is ODBC-JDBC bridge, the subprotocol must be ODBC. **<Subname>** is used to identify the database.

A Sample JDBC URL

The following is a sample JDBC URL:

```
String url = "jdbc:oracle:thin:@localhost:1521:xe";

// URL For thin Driver

//"jdbc:oracle:thin:@localhost:1521:xe"

Class.forName ("oracle.jdbc.driver.OracleDriver")

Connection con = DriverManager. getConnection
(url,"system","password");
```

Processing querying in a database

Once a connection with the database is established, you can query the database and process the result set. JDBC does not enforce any restriction on the type of SQL statements that can be sent, but as a programmer, it is your responsibility to ensure that the database can process the statements.

JDBC provides three classes for sending the SQL statements to a database, which are as follows:

- **The Statement object**: You can create the statement object by calling the **create statement ( )** method from the connection object.
- **The Prepared Statement object**: You can create the Prepared statement object by calling the **preparedStatement ()** method from the connection object. The prepared statement object contains a set of methods that can be used for sending queries with the INPUT parameters.
- **The Callable Statement object**: You can create the Callable Statement object by calling the **prepareCall( )** method from the connection object. The CallableStatement object contains the functionality for calling a stored procedure. You can

handle both the INPUT as well as the OUTPUT parameters using the Callable Statement Object.

Using the Statement object

You can use the statement object to send simple queries to the database as shown in the following sample Query-App program:

```
//Query App. java
import java.sql. *;
public class QueryApp {


  public static void main (String a [] )
  {
    try  {
  Class. forName("oracle.jdbc.driver.OracleDriver");


  Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system","sail_
boat1");
  Statement stat = con.createStatement();
  stat.executeQuery ("select * from emp");
  }
  catch(Exception e)
  {System.out.print ("Error" + e);
  }
  }
  }
```

In the preceding Query App example, the following happens:

1. The thin/Oracle driver is loaded.
2. The connection object is initialized using the **`getConnection()`** method.
3. The statement object is created using the **`createStatement()`** method.

4. Finally, a simple query is executed using the **executeQuery()** method of the statement object.

The Statement Object

The **Statement** object allows you to execute the simple queries. It has three methods that can be used for the purpose of querying, which is as follows:

- The **executeQuery()** method executes a simple select query and returns a single **ResultSet** object.
- The **executeUpdate()** method executes the SQL INSERT, UPDATE, and DELETE statement.
- The **execute()** method executes an SQL statement that may return multiple results.

The ResultSet Object

The ResultSet object provides you with the methods to access the data from the table. Executing a statement usually generates a ResultSet object. It maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. The **next()** method moves the cursor to the next row; you can access the data from the ResultSet rows by calling the **getxxx()** method, where xxx is the data type of the parameter. The following code queries the database and processes the ResultSet:

```
import java.sql. *;
public class QueryApp {

  public static void main (String a [] )
  { ResultSet result;
    try  {
  Class.forName("oracle.jdbc.driver.OracleDriver");

  Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system","sail_
boat1");
  Statement stat = con.createStatement();
  result=stat.executeQuery ("select * from emp");
  while(result.next())
```

```
  {
    System.out.println(result.getString(2));/* retrieving
data from 2nd Col. of emp table*/
  }
  }
  catch(Exception e)
  {System.out.println ("Error" + e);}  }}
```

In the preceding Query App example, the following happens:

1. The ResultSet object is returned by the **`executeQuery()`** method.
2. All the rows in the ResultSet object are processed using the **`next()`** method in a `while` loop.
3. The values of the second column are retrieved using the **`getString()`** method.

The output is as follows:

```
Suresh
Radhika
```

You can modify the same program to display the content of the table in a window:

```
import java.awt.*;
import java.awt.event.*;
import java.sql. *;
public class QueryApp extends Frame implements
ActionListener{
  TextField eid,ename;
  Button next;
  Panel p;
  static ResultSet result;
  static Connection con;
  static Statement stat;
  public QueryApp(){
    super("The Query Application");
```

```
    setLayout(new GridLayout(5,1));
    eid=new TextField();
    ename=new TextField();
    next=new Button("Next");
    setSize(500,500);
    p=new Panel();
    add(new Label("Employee ID"));
    add(eid);
    add(new Label ("Employee Name: "));
    add(ename);
    add(p);
    p.add(next);
    next.addActionListener(this);
    setVisible(true);
  }
  public static void main (String a [] )
  {
    QueryApp ab=new QueryApp();
    try  {
  Class. forName("oracle.jdbc.driver.OracleDriver");
  System.out.println("after Driver");
  con = DriverManager.getConnection ("jdbc:oracle:thin:@
localhost:1521:xe","system","sail_boat1");
  stat = con.createStatement();
  System.out.println("after statement");
  result=stat.executeQuery("select * from emp");
    }
  catch(Exception e)
  {System.out.print ("Error in sql" + e);}}
  @Override
  public void actionPerformed(ActionEvent event) {

    if(event.getSource()==next)
```

```
    {
      try{
        while( result.next())
        {
          eid.setText(eid.getText()+"  "+(result.
getString(1)));
          ename.setText(ename.getText()+"  "+(result.
getString(2)));
        }
      }catch(Exception e){System.out.println(e);}
    }
    }
    }
```

In the preceding example, the following happens:

1. The **`main()`** method creates an object of the QueryApp class.
2. The constructor of the Query App class places the controls on the window.
3. The query is then executed.
4. When the user clicks the **`Net`** button, the action **`Performed()`** method moves the **`Resultset`** object to the next record and displays that record.

The output of the preceding program is shown in *Figure 1.4*:

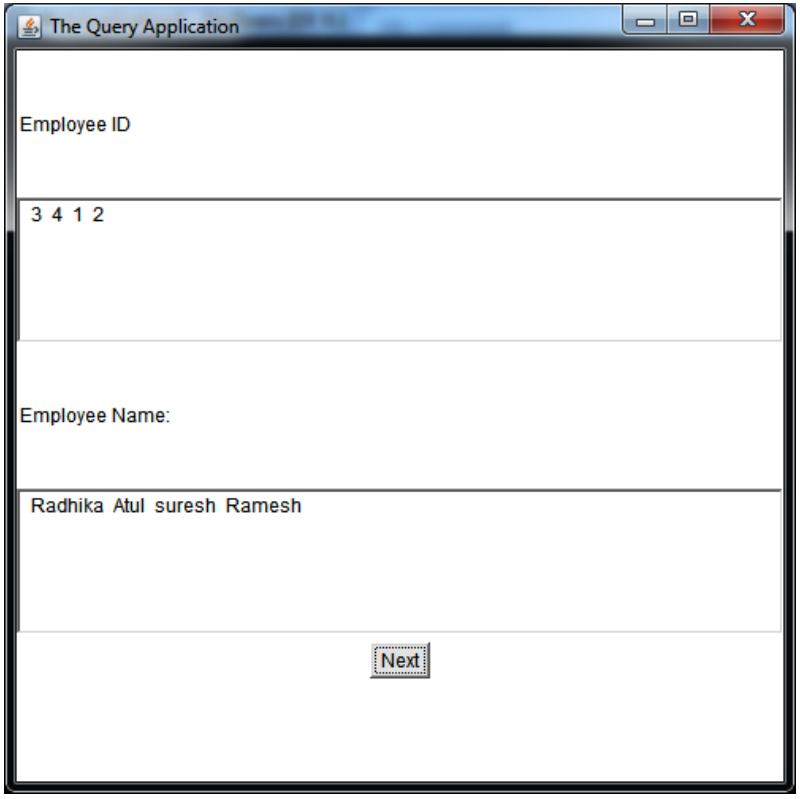


***Figure 1.4**: Output of the preceding program*

Prepared Statement Object

You have to develop an application that queries the database according to the search criteria specified by a user. For example, the user supplies the employee-ID-eid and wants to see the details of that employee.

Select * from employee where eid=? **You do not know the id**

To make it possible, you must prepare a query statement that receives an appropriate value in the where clause at run time.

Using the Prepared Statement object

The Prepared Statement object allows you to execute the parameterized queries. The Prepared Statement object is created using the **`preparedStatement()`** method of the connection object, as follows:

```
PreparedStatement stat = con.preparedStatement("select *
from emp where eid = ?");
```

The **preparedStatement()** method of the connection object takes an SQL statement as a parameter. The SQL statement can contain the placeholders that can be replaced by the INPUT parameters at runtime.

NOTE: The "?" Symbol is a placeholder that can be replaced by the INPUT parameters at runtime.

Passing Input Parameters

Before executing a **preparedStatement** object, you must set the value of each**?** parameter. This is done by calling a **setxxx()** method, where xxx is the datatype of the parameter. Look at the following code example:

```
Stat. setString (1, eid.getText());
ResultSet result = stat. executeQuery ();
```

The following code makes use of the Prepared Statement Object:

```
import java.sql.*;
import java.awt.*;
import java.awt. event.*;
public class PreparedQuery extends Frame implements
ActionListener
{
TextField eid, ename;
Button query;
Panel p;
/* These variables are declared as static because they
have to be accessed in a static method*/
static ResultSet result;
static Connection con;
static PreparedStatement stat;
/* The constructor of the Prepared Query App class */
public  PreparedQuery()
{
  super("The Query Application");
```

```
  setLayout(new GridLayout(5,1));
  eid=new TextField();
  ename=new TextField();
  query=new Button("QUERY");
  setSize(500,500);
  p=new Panel();
  add(new Label("Employee ID"));
  add(eid);
  add(new Label ("Employee Name: "));
  add(ename);
  add(p);
  p.add(query);
  query.addActionListener(this);
  setVisible(true);
}
/* The main method creates an object of the class and
displays the first record*/
public static void main (String a [] )
{
  PreparedQuery obj = new PreparedQuery( );
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
  con = DriverManager.getConnection ("jdbc:oracle:thin:@
localhost:1521:xe","system","sail_boat1");
  stat = con.prepareStatement ("select * from emp where
id= ?");
}
catch(Exception e)
{
System. out. print ("Error in sql" + e);
}
}
@Override
public void actionPerformed(ActionEvent event) {

  if(event.getSource()==query)
```

```
  {
    try{
      stat.setString(1,eid.getText());
      result=stat.executeQuery();
      while(result.next())
      {
        eid.setText(result.getString(1));
        ename.setText(result.getString(2));
      }
    }catch(Exception e){System.out.println(e);}
  }
  }
}
```

In the preceding example, the following happens:

1. The **`PreparedStatement`** object is created using the **`prepared statement()`** method.
2. The parameters of the **`PreparedStatement`** object are initialized when the user clicks on the **`Query`** button.
3. The query is then executed using the **`execute Query()`** method and the result is displayed in the corresponding controls, as shown in *Figure 1.5*:

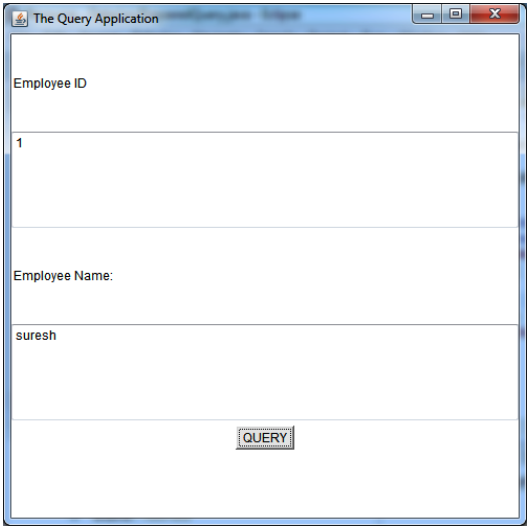


***Figure 1.5**: The output of the preceding program*

Adding Records

You can use the **executeUpdate()** method of the statement object to execute simple INSERT statements, such as the following:

```
Stat.executeUpdate("insert <tablename> values ( )");
```

The return value of the **executeUpdate()** method is the number of rows affected by the query, as follows:

```
public  void  addRecord (){try
{
stat. executeUpdate (“Insert into publishers values
(“1020”, “New Employee”);
} catch (Exception e) {  }
```

NOTE: The PreparedStatement object can be used for sending the parameterized INSERT statements to the database.

Modify Records

You can use the **executeUpdate()** method of the Statement object to execute simple UPDATE statements, such as the following:

```
stat.executeUpdate ("update <tablename> set <Expr>");
```

The return value of the **executeUpdate()** method is the number of rows affected by the Query, as follows:

```
public void modifyRecord ( ){
try {
stat.executeUpdate (“update Employee set
EName=‘NewEmployee' Where eid= '1234'”),}
catch(Exception e)
{System.out.println("exception ”+e); }
}
```

NOTE: The Prepared Statement object can be used for sending the parameterized UPDATE Statements to the database.

Deleting Records

You can use the **executeUpdate()** method of the statement object to execute simple Delete statements, such as the following:

```
stat.executeUpdate ("delete <tablename> where <Expr>");
```

The return value of the **`executeUpdate()`** method is the number of rows affected by the query, as follows:

```
public void  deleteRecord()
{try
{
stat.executeUpdate(“delete Employee where Eid = '1234');
}catch (Exception e)  {System.out.println("Exception”);}
}
```

NOTE: The Prepared Statement object can be used for sending the parameterized DELETE Statements to the database.

Conclusion

The database is a repository of information used by the applications. ODBC API is a set of library routines that enables your programs to access a variety of databases. JDBC provides a database-programming API for Java programs. The JDBC-ODBC bridge driver translates the JDBC API to the ODBC API. 5. There are several categories of JDBC drivers available, which are as follows:

- JDBC-ODBC bridge + ODBC driver
- Native API, partly Java driver
- JDBC-Net, pure Java driver
- Thin Driver

The **`java.sql`** package contains the classes that help in connecting to a database, sending the embedded SQL statements to the database, and processing the query results. The connection object represents a connection to a database. The Prepared Statement object allows you to execute the parameterized queries. The **`ResultSet`** object provides you with the methods to access the data from a table.

Many people do not understand English and want messages, currencies, and time in their languages. Internationalization helps to do the same. In the next chapter, we will study Internationalization, which is the process of customizing an application according to specific languages and regions.

Multiple Choice Questions

1. **The JDBC-ODBC bridge is which of the following?**
   a. Single-Threaded
   b. Multi-Threaded
   c. None of these
2. **DriverManager is which of the following?**
   a. Interface
   b. Class
   c. Method
3. **Commit is a method of which of the following?**
   a. Connection
   b. ResultSet
   c. Statement
4. **Which of the following methods are needed for loading a database driver in JDBC?**
   a. Class.forName()
   b. RegisterDriver()
   c. None of these
5. **How many JDBC driver types does Sun define?**
   a. One
   b. Four
   c. Three

Answers

1. **b**
2. **b**
3. **a**
4. **a**
5. **b**

Fill in the blanks

1. _____ is an open-source DBMS product that runs on UNIX, Linux, and Windows.

   Ans: MYSQL

2. _____Statement can execute the parameterized queries.

   Ans: Prepared Statement

3. The_____ object is returned by the execute Query () method.

   Ans: ResultSet

4. The _____Symbol is a placeholder that **can be** replaced by the INPUT parameters at runtime.

   Ans: "?"

State True/False

1. JDBC is an API to connect to relational-, object- and XML data sources.

   Ans: False

2. The JDBC-ODBC Bridge supports multiple concurrent open statements per connection.

   Ans: True

3. JDBC is an API to access the relational databases.

   Ans: False

4. java.sql and javax.sql packages contain the JDBC Classes and Interfaces.

   Ans: True

Questions

1. What is database and database driver?
2. How many types of drivers are available? Explain.

3. Write the steps to connect with the database.
4. What is ResultSet? What is the return type of ResultSet?

Interview Questions

1. What are the main steps in Java to make the JDBC connectivity?
2. What are the different types of Statements?
3. What is JDBC?
4. Which type of Statement can execute the parameterized queries?
5. Does the JDBC-ODBC Bridge support multiple concurrent open statements per connection?

Chapter 2
Internationalization (I18N)

Introduction

Many people do not understand English and want messages, currencies, and time in their languages. Internationalization helps in doing the same. Internationalization is the process of customizing an application according to specific languages and regions. Internationalization has 18 characters in it, which is why it is abbreviated as I18N.

Structure

In this chapter, we will cover the following topics:

- Localization
- Locale
- Resource bundle
- Steps to develop I18N-based application
- Internationalization

Objective

After studying this chapter, you will be able to develop an application and display the messages according to specific languages and regions.

Localization (L10N)

It is a mechanism to create an application in a specific language or region. This can be done via the Locale object. An object of the Locale class represents a geographical or cultural region. Locale class is used to get the information about the country language, variant, and so on. A variant represents operating system-specific information.

Locale

It is a class in the **`java.util`** package that provides information about the country and the region.

Constructors of Locale Class

There are three constructors of the locale class, which are as follows:

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

Commonly used methods of Locale class

The following are the commonly used methods of the Locale class:

- **`public static Locale getDefault()`**: It returns the instance of current Locale.
- **`public static Locale[] getAvailableLocales()`**: It returns an array of available locales.
- **`public String getDisplayCountry()`**: It returns the country name of this locale object.
- **`public String getDisplayLanguage()`**: It returns the language name of this locale object.
- **`public String getDisplayVariant()`**: It returns the variant code of this locale object.

- **`public String getISO3Country()`**: It returns the three-letter abbreviation for the current locale's country.
- **`public String getISO3Language()`**: It returns the three-letter abbreviation for the current locale's language.

The program to get the information of India is as follows:

```
import java.util.Locale;
public class LocaleTest {
  public static void main(String[] args) {
    Locale locale=new Locale("hi","IN");
    System.out.println(locale.getDisplayCountry());
    System.out.println(locale.getDisplayLanguage());
    System.out.println(locale.getDisplayName());
    System.out.println(locale.getISO3Country());
    System.out.println(locale.getISO3Language());
    System.out.println(locale.getLanguage());
    System.out.println(locale.getCountry());
}}
```

The output is as follows:

```
India
Hindi
Hindi (India)
IND
hin
hi
IN
```

Resource Bundle

It is a good practice if the messages or labels used inside the application are visible in the user's language, that is, if the user belongs to India, then the message is in Hindi, or if the users belong to the United States the messages are in English. Resource Bundle stores the text and components that are Locale sensitive and loads this information from the properties file that contains the messages.

The **java.util.ResourceBundle** class is an abstract class; we cannot make the object of the **ResourceBundle** class. Its method helps us manage locale-sensitive resources.

Constructor of ResourceBundle

- **ResourceBundle()**: This is the default constructor mainly designed for use by the subclasses and the factory methods.

Methods of ResourceBundle Class

The following are the methods of the ResourceBundle class:

- **public static ResourceBundle getBundle(String basename)**: It loads the resource bundle with the given name and returns the instance of the **ResourceBundle** class.
- **public static ResourceBundle getBundle(String basename, Locale locale)**: It loads the resource bundle with the given name and the specified locale and returns the instance of the **ResourceBundle** class.
- **public String getString(String key)**: It returns the value for the corresponding key from this resource bundle.
- **Locale getLocale()**: It returns the Locale associated with the current bundle.
- **static final clearCache()**: It deletes all the resource bundles from the cache that were loaded by the default class loader.
- **boolean containsKey()**: It returns true if the passed string argument is a key within the invoking resource bundle.
- **protected void setParent()**: It sets the passed bundle as the parent of the invoking bundle. In the case of a lookup, if the key is not found in the invoking object, then it is looked up in the parent bundle.
- **final Object getObject()**: It retrieves and returns the object associated with the key passed as an argument, either from the current resource bundle or the parent.

NOTE:

- **The name of the properties file should be filename_ languagecode_country code, for example, MyMessage_ hi_IN.properties.**
- **The properties file containing the Locale sensitive information is also called resource bundle.**

Steps to develop the I18N-based application

The following are the steps to develop the I18N-based application:

1. Identify the Locale's sensitive information for which you want to customize the application.
2. Create properties files for each Locale, and the name of the properties files must be **`filename_languagecode_country code`**.
3. Get the Resource Bundle instance.
4. Read the message from the properties file with the help of the ResourceBundle methods.

The following are the examples:

1. Create **`MessageBundle_hi_IN.properties`**, as follows:

```
welcome=Namaskar,Kaisehaiaap?
```

2. Create **`MessageBundle_en_US.properties`**, as follows:

```
welcome=Hello, How are you?
```

3. Create a Java class to get the instance of **`ResourceBundle`** and read the message from the properties file according to their Locale, as follows:

```
import java.util.Locale;
import java.util.ResourceBundle;
public class I18NDemo {
public static void main(String[] args) {
//Default Locale is the US
ResourceBundle bundle = ResourceBundle.
```

```
getBundle("MessageBundle", Locale.US);
System. out.println("Message in " + Locale.US + ": "
+ bundle.getString("welcome"));

// Now the default locale change to India
Locale.setDefault(new Locale("hi", "IN"));
bundle = ResourceBundle.getBundle("MessageBundle");
System.out.println("Message in " + Locale.
getDefault() + ": " + bundle.getString("welcome"));


}}
```

The output is as follows:

```
Message in en_US: Hello, How are you?
Message in hi_IN: Namaskar, Kaise hai aap?
```

Internationalizing Date and Time (I18N with Date and Time)

Whenever Locale is changed, the format of the date and time is also changed. This means that the format of the date and time is different from region to region.

The **`java.text.DateFormat`** class has the following methods that help in changing the date and time according to the region.

Methods of java.text.DateFormat

The following are the methods for java.text.DateFormat:

1. **`public static DateFormat getDateInstance(int style, Locale locale)`**: This method returns the object of **`DateFormat`** based on the specified style and the **`Locale. This`** object is used to format the date and not the time.
2. **`public static DateFormat getTimeInstance(int style, Locale locale)`**: This method is used to format the time according to the specified style and Locale.
3. **`public String format(java.util.Date date)`**: This method formats the date into the date-time string.

NOTE: The DateFormat class has the following styles to display the date and time:

- `static final int  SHORT`
- `static final int LONG`
- `static final int FULL`
- `static final int MEDIUM`
- `static final int DEFAULT`

The following is an example:

```
import java.util.Locale;
import java.text.DateFormat;
public class DateFormatDemo {

  public static void main(String[] args) {
    // TODO Auto-generated method stub
    String lang="en";
    String country="US";
Locale l=new Locale(lang,country);
DateFormat df= DateFormat.getDateInstance(DateFormat.
LONG,l);
String dt=df.format(new java.util.Date());
System.out.println(dt);
// Time

DateFormat df2= DateFormat.getTimeInstance(DateFormat.
LONG,l);
String time=df2.format(new java.util.Date());
System.out.println(time);
  }}
```

The following is the output:

```
May 10, 2020
7:33:47 PM IST
```

Internationalizing with Numbers (I18N with Numbers)

Whenever a Locale is changed, the format of the number also changes. The format of the numbers is also different from region to region. The **`java.text.NumberFormat`** class is used to format the number and the currency percent according to the specific locale.

The following are the methods of the NumberFormat:

- **`public static NumberFormat getNumberInstance(Locale locale)`**: This returns the NumberFormat object and is used to format the number according to the locale.
- **`public static NumberFormat getInstance(Locale locale)`**: This returns a general-purpose number format for the current default locale.
- **`public static NumberFormat getPercentInstance(Locale l)`**: This returns a percentage format for the specified locale.
- **`public static NumberFormat getCurrencyInstance(Locale l)`**: This returns a currency format for the current default locale.

The following is an example:

```
import java.text.NumberFormat;
import java.util.Locale;
public class FormatextDemo {
  public static void main(String[] args)
{
    Locale l=new Locale(args[0],args[1]);
    NumberFormat nf=NumberFormat.getNumberInstance(l);
    String num=nf.format(123456789.12345);
    System.out.println("Number for "+args[0] +" "
+args[1]+" : "+ num);
    //Currency Format
    NumberFormat nf2=NumberFormat.getCurrencyInstance(l);
    String num2=nf2.format(123456789.12345);
    System.out.println("Currency of  "+args[0] +" "
```

```
+args[1]+" : "+ num2);
    //Percent Format

    NumberFormat nf3=NumberFormat.getPercentInstance(l);
    String num3=nf3.format(123456789.12345);
    System.out.println("Percent Format of   "+args[0] +" "
+args[1]+" : "+ num3);
  }    }
```

The following is the output (for the US region):

```
Number for en US : 123,456,789.123
Currency of  en US : $123,456,789.12
Percent Format of   en US : 12,345,678,912%
```

The following is the output (for France):

```
Number for fr FR : 123 456 789,123
Currency of fr FR: 123 456 789,12 €
Percent Format of fr FR : 12 345 678 912 %
```

The following is the output (for Japan):

```
Number for ja JP : 123,456,789.123
Currency of  ja JP : ?123,456,789
Percent Format of   ja JP : 12,345,678,912%
```

Conclusion

1. **Localization**: It is a mechanism to create an application in a specific language or region. This can be done via the **`Locale`** object.

2. **Locale:** It is a class in **`java.util`** package that provides information about the country and the region.

3. Resource Bundle stores the text and components that are Locale sensitive and loads this information from the properties file that contains the messages. The **`java.util.ResourceBundle`** class is an abstract `class.` **`Properties`** file that contains the Locale sensitive information and is also called resource Bundle.

4. The name of the properties file should be the **filename_languagecode_country** code, for example, **MyMessage_hi_IN.properties**.
5. Whenever a Locale has changed, the format of the number also changes. The format of the numbers also differs from region to region. The **java.text.NumberFormat** class is used to format the following according to the specific locale:
    - Format the number
    - Format the currency
    - Format the percent

Multiple Choice Questions

1. **Given:**

```
Date = new Date(); //line no 12
df.setLocale(Locale.ITALY); //line no 13
String s = df.format(date); // line no14
```

The variable df is an object of the type DateFormat that has been initialized inline 11. What is the result if this code is run on December 14, 2000?

A. The value of s is 14-dec-2004.
B. The value of s is Dec 14, 2000.
C. An exception is thrown at runtime.
D. Compilation fails because of an error in line 13.

Answer: D

2. **Given:**

```
NumberFormat nf = NumberFormat.getInstance(); //
line no 12
nf.setMaximumFractionDigits(4); // line no 13
nf.setMinimumFractionDigits(2); // line no 14
String a = nf.format(3.1415926); // line no 15
String b = nf.format(2); // line no 16
```

Which two statements are true about the result if the default locale is Locale.US? (Choose two.)

A. The value of b is 2.

B. The value of a is 3.14.

C. The value of b is 2.00.

D. The value of a is 3.141.

E. The value of a is 3.1415.

F. The value of a is 3.1416.

G. The value of b is 2.0000.

Answer: C, F

3. **Given:**

   d is a valid, non-null Date object.

   df is a valid, non-null DateFormat object set to the current locale.

   What outputs the current locale's country name and the appropriate version of d's date?

   A. Locale loc = Locale.getLocale();

      System.out.println(loc.getDisplayCountry()+ " " + df.format(d));

   B. Locale loc = Locale.getDefault();

      System.out.println(loc.getDisplayCountry()+ " " + df.format(d));

   C. Locale loc = Locale.getLocale();

      System.out.println(loc.getDisplayCountry()+ " " + df.setDateFormat(d));

   D. Locale loc = Locale.getDefault();

      System.out.println(loc.getDisplayCountry()+ " " + df.setDateFormat(d));

   Answer: B

4. Given:

```
import java.text.*;
class DateOne {
public static void main(String[] args) {
Date d = new Date(1123631685981L);
DateFormat df = new DateFormat();
System.out.println(df.format(d));
. }
. }
```

And given that 1123631685981L is the number of milliseconds between Jan. 1, 1970, and sometime on Aug. 9, 2005, what is the result? (Note: the time of day in option A may vary.)

A. 8/9/05 5:54 PM

B. 1123631685981L

C. An exception is thrown at runtime.

D. Compilation fails due to a single error in the code.

E. Compilation fails due to multiple errors in the code.

Answer:

-> E is correct. The Date class is located in **`java.util`** package, so it needs an import, and the **`DateFormat`** objects must be created using a static method such as **`DateFormat.getInstance()`** or **`DateFormat.getDateInstance()`**.

->A, B, C, and D are incorrect based on the above.

5. Which of the following statements are true? (Choose all that apply.)

A. The DateFormat.getDate() is used to convert a String to a Date instance.

B. Both the DateFormat and the NumberFormat objects can be constructed to be Locale specific.

C. Both the Currency and the NumberFormat objects must be constructed using the static methods.

D. If a NumberFormat instance's Locale is to be different from the current Locale, it must be specified at the creation time.

E. A single instance of the NumberFormat can be used to create the number objects from the strings and create the formatted numbers from the numbers.

Answer:

-> B , C, D, and E are correct.

->A is incorrect, DateFormat.parse() is used to convert a String to a Date.

Questions

1. How will you use a specific Locale in Java?
2. Differentiate between Localization and internationalization?
3. How do I go about internationalizing an existing program?
4. What is ResourceBundle? How can one extract the strings to the Resource Bundle files?
5. Explain how Java application uses multiple locales?

Chapter 3
Introduction to Java Servlets

Introduction

Consider a situation where you would like to register yourself with a website to get a free newsletter subscription. The information you are required to provide is your first name, last name, and your organization's name. In addition, you can choose an ID for yourself to log in to the website. After entering the details and clicking on the Submit button, the entered data is forwarded to the web server for processing. Processing in this situation could consist of checking whether the ID you have entered is already in use or checking for invalid values. If the ID is already in use, the user is asked to choose a different ID. Validations such as checking for blank fields and negative values could be done on the client's side itself to reduce the overhead of the server. Small programs that are written for such purposes are called client-side scripts. VB scripts and Java scripts are two languages that can be used for client-side scripting. The programs that take care of processing the web server's data are called server-side scripts. The web applications also demand more functionality from the webserver. This has triggered the development of tools that enable efficient server-side programming, such as Servlets, Java Server Pages (JSP), Active Server Pages (ASP), and so on.

Structure

In this chapter, we will cover the following topics:

- Webserver
- Servlets and their characteristics
- Comparison between Servlet and Applet
- Working of Servlet
- Lifecycle of Servlet
- Methods of Servlet
- Deployment descriptor

Objectives

After studying this chapter, you will learn how to communicate with the webserver with the help of Servlets. You will understand the lifecycle methods of Servlets. You will also learn how the web container gets the information about Servlets with the help of the deployment descriptors.

Webserver

A webserver takes the client request, finds the resource (HTML page, picture, sound file, records), and returns something to the user. When the server does not find a requested resource, it gives 404 "Not Found Error." The webserver is a combination of the hardware and the software. Hardware can be any physical machine, and software is a web server application.

Introduction to Servlets

Servlets are Java programs that can be deployed on a Java-enabled webserver to enhance and extend the functionality of the webserver. For example, you can write a Servlet to add a messenger service to the Earnest Bank website. Servlets can also be used to add dynamic content to web pages. For example, you can use a Servlet to retrieve the latest gift offers provided by the Earnest Bank from an information database and display it on the bank's home page.

Characteristics of Servlets

Servlets can be used to develop a variety of web-based applications. As Servlets are written using Java, they can use the extensive power of the Java API, such as networking and URL access, multithreading, database connectivity, internationalization, **remote method invocation** (**RMI**), and object serialization. The characteristics of Servlets that have gained them widespread acceptance are as follows:

- **Servlets are efficient**: The initialization code for a Servlet is executed only when the Servlet is executed for the first time. Subsequently, the Servlet's requests are processed by its `service()` method. This helps increase the efficiency of the server by avoiding the creation of unnecessary processes.
- **Servlets are robust**: Servlets are based on Java; they provide all the powerful features of Java, such as exception handling and garbage collection, which make them robust.
- **Servlets are portable**: Servlets are also portable because they are developed in Java. This enables easy portability across the web servers.
- **Servlets are persistent**: Servlets increase the system's performance by preventing frequent disk access. For example, if a customer logs on to **www.EarnestOnline.com**, the customer can perform many activities, such as checking for the balance, applying for a loan, and so on. In every stage, the customer needs to be authenticated by checking for the account number against the database; instead of checking for the account number against the database every time, Servlets retain the account number in the memory till the user logs out of the website.

Comparison between Servlets and Applets

Applets are Java programs that are embedded in web pages. When a web page containing an Applet is opened, the byte code of the applet is downloaded to the client's computer. This process becomes time-consuming as the size of the applet is too large. As the Servlets execute on the web server, they help overcome problems with the

download time faced while using applets. Servlets do not require the browser to be Java enabled, unlike the applets, because they execute on the webserver, and the results are sent back to the client or the browser. The applet is a depreciated technology.

Comparison between Servlets and other server-side scripting technologies

Common Gateway Interface (**CGI**) scripts, JSP, and ASP are alternatives to Servlets and have their advantages and disadvantages.

CGI scripts

A CGI script is a program that is written in C, C++, or Perl. A CGI script gets executed in a server when a server receives a request from the client for processing the data. The server passes the request to the CGI script. The CGI script processes the request and sends the output in the form of HTML to the server. The server, in turn, passes the request to the client.

The disadvantages of using a CGI script are as follows:

- Whenever a CGI script is invoked, the server creates a separate process for it. The server has a limitation on the number of processes that can be created simultaneously. If the number of requests is too high, the server will not be able to accept the requests. In addition, the creation of too many processes will also bring down the efficiency of the server.
- The most popular platform for writing a CGI script is Perl. Even though Perl is a very powerful language for writing CGI applications, the server needs to load the Perl interpreter for each request that it receives. Thus, for an incoming request, the executable file of the CGI script and the Perl interpreter is loaded, which brings down the efficiency of the server.

Unlike the CGI scripts, the Servlet initialization code is executed only once. In the case of Servlets, each request is handled by a separate thread in the webserver more efficiently by preventing the creation of unnecessary processes.

Active Server Pages (ASP)

ASP is a server-side scripting language that has been developed by Microsoft. ASP enables a developer to combine HTML and a Scripting language on the same web page. JavaScript and VBScript are two scripting languages that are supported by ASP. VBScript and JavaScript can also be used for server-side scripting by using the run at (**`<script runat=server>`**) tag. The limitation of ASP is that it is not compatible with all the web servers. The other web servers need specific plug-ins to be installed to support ASP. However, adding a plug-in can decrease the performance of the system.

Working of Servlets

The client or the browser passes the requests to the server using the GET or the POST methods. For example, a Servlet could be invoked by clicking a user-interface component, such as a button on a form or following a hyperlink on a web page. After the Servlet processes the request, the output is returned as an HTML page to the client.

The client request consists of the following components:

- The protocol used for the communication between the server and the client, such as HTTP.
- The request type can be GET or POST.
- The query string contains additional information such as login name, password, and registration details.

The following is an example: GET

```
http://www EarnestBank.com/login.html?username="sarika"&-
passwd="3445H"
```

The preceding URL is used to display a user's mailbox called **`"Sarika"`**.

Table 3.1 describes the different components of the URL:

Component	Description
HTTP	It is the protocol that is used for communication between the server and the client.
www.EarnestBank.com	It is the name of the website.
login.html	It is the name of the form that is displayed to the user.
Username="sarika"&-passwd="3445H	These are the values that are passed to the server-side program.

***Table 3.1**: Different components of the URL*

The GET and POST methods

When a client sends a request to the server, the client can also pass additional information with the URL to describe what exactly is required as the output from the server by using the **`GET`** method. The additional sequence of characters appended to the URL is called a **Query String**. However, the length of the query string is limited to 240 characters. Moreover, the query string is visible on the browser and is called, therefore, a security risk.

To overcome these disadvantages, the **`POST`** method can be used. In the POST method, a large amount of data can be sent through a separate socket connection. The complete transaction is invisible to the client.

The disadvantage of this method is that it is slower than the GET method because the data is sent to the server as separate packets.

The Javax.servlet package

Java supports the implementation of Servlets through the **`javax.servlet`** and **`javax.servlet.http`** packages. The **`javax.servlet`** interface provides the general framework for creating a Servlet. A Servlet can directly implement this interface or indirectly implement the same by extending the **`javax.servlet.GenericServlet`** or the **`Javax.Servlet.http.HttpServlet`** classes.

The **`GenericServlet`** class of the **`javax.Servlet`** package is used to create Servlets that can work with any protocol. The **`javax.`**

`Servlet.http` package is used to create the HTTP Servlets that provide the output in the form of HTML pages. The class that is used to create the HTTP Servlets is called **`HttpServlet`** and is derived from the **`GenericServlet`** class. Serialization is also made possible in Servlet and is derived from the **`GenericServlet`** class through the serializable interface. Serialization is the process of writing an object into a persistent storage medium, such as a hard disk. The hierarchies of the classes that are used to create a Servlet are shown in *Figure 3.1*:

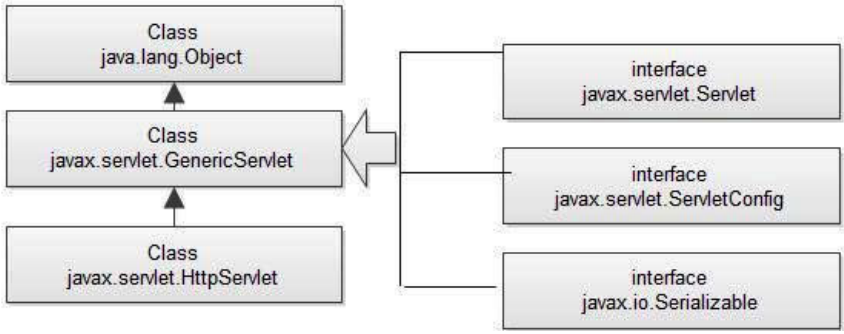


***Figure 3.1**: The Servlet class Hierarchy*

Table 3.2 describes the classes and interfaces used for the creation of Servlets:

Class/Interface Name	Description
`HttpServlet` `class`	Provides an HTTP-specific implementation of the Servlet interface. This class extends the `GenericServlet` class that provides a framework for handling the other types of network and web services.
`HttpServletRequest` `Interface`	It extends the `ServletRequest` interface to provide the methods to process the requests from the clients. For example, assume that the client browser consists of a form with two fields. When the values are submitted to the server for processing, they are extracted using the methods in the `HttpServletRequest` interface.
`HttpServletResponse` `interface`	It extends the `ServletResponse` interface to provide the methods to send a response. For example, it has methods to access the HTTP headers and cookies.

***Table 3.2**: Classes and Interface to create a Servlet*

Lifecycle of a Servlet

The lifecycle of a Servlet gives the states of Servlet. There are three states of Servlet – new, ready, and end. The states of the servlets change with the help of the calling of servlets method.

Servlet Interface

The servlet interface provides a common functionality in all the Servlets. The servlet interface defines the methods that all the Servlets must implement. HttpServlet and GenericServlet implement servlet interface directly or indirectly. The servlet interface provides three lifecycle methods used to implement any Servlet.

A Servlet is loaded only once in the memory and is initialized in the **`init()`** method. After the Servlet is initialized, it starts accepting a request from the client and processes them through the **`service()`** method until it is shut down by the **`destroy()`** method. The **`service()`** method is executed for every incoming request. The lifecycle of a Servlet is depicted as follows:

1. Servlet class is loaded.
2. Servlet instance is created.
3. The init method is invoked.
4. The service method is invoked.
5. The destroy method is invoked.

Refer to *Figure 3.2,* which illustrates the lifecycle of a Servlet:

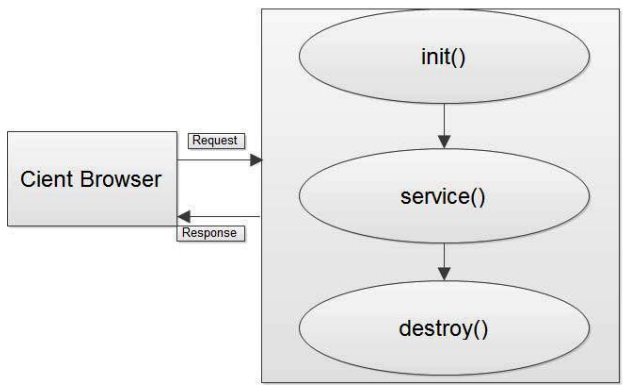


***Figure 3.2**: Lifecycle of a Servlet*

Table 3.3 describes the methods of the Servlet interface:

S.No	Method name	Description
1	`public void init(ServletConfig config) throws Servlet Exception`	It contains all initialization codes for the Servlet and is invoked when the Servlet is first loaded and created.
2	`public void service(ServletRequest request,ServletResponse response);`	It receives all the requests from clients, identifies the type of requests, and dispatches them to the `doGet()` or `doPost()` methods for processing.
3	`public void destroy()`	It executes once when the Servlet is removed from the server. The `cleanup` code for the Servlet must be provided in this method.
4	`public ServletConfig getServletConfig();`	It returns the object of `ServletConfig`.
5	`public String getServletInfo()`	It returns the information about the Servlet, such as writer, copyright, version, and so on.

***Table 3.3**: Methods of Servlet Interface*

Creating a Servlet

Along with three lifecycle methods(int(),service() and destroy()), two more methods are also used to create a Servlet.

Table 3.4 also describes two methods that are used in creating a Servlet:

S.No	Method name	Description
1	`ServletResponse.getWriter ()`	It returns a reference to a `PrintWriter` object. The `PrintWriter` class is used to write the formatted objects as a text-output stream onto the client.
2	`ServletResponse.setContentType (String type)`	It sets the type of content sent as a response to the client browser. For example, `SetContentType ("text/html")` is used to set the response type as text.

***Table 3.4**: Methods that are used in creating a Servlet along with the life cycle method*

The Servlet can be created in three of the following ways:

- By implementing the **`Servlet`** interface
- By inheriting **`GenericServlet`** class
- By inheriting **`HttpServlet`** class

1. **The following is an example of a servlet created by implementing the Servlet Interface**:

```
import java.io.*;
import javax.Servlet.*;
public class FirstServlet implements Servlet{
ServletConfig config=null;
public void init(ServletConfig config)
{
this.config=config;
System.out.println("Servlet is initialized");
}
public void service(ServletRequest
req,ServletResponse res)
throws IOException,ServletException
{
res.setContentType("text/html");
PrintWriter out=res.getWriter();
out.print("<html><body>");
out.print("<b>This is First Servlet through Servlet
Interface</b>");
out.print("</body></html>");
}
public void destroy()
{
System.out.println("Servlet is destroyed");
}
public ServletConfig getServletConfig()
{    return config;
}
public String getServletInfo()
```

```
{
    return "Institute of Learning- Advance Java
First Servlet via Servlet interface";
}
}
```

Compile a Servlet

For compiling the Servlet, a JAR file is required to be loaded. Different servers need different jar files.

Table 3.5 provides the list of different servers and JAR files:

Jar File	Server
javaee.jar	Glassfish/JBoss
weblogic.jar	Weblogic
servlet-api.jar	Apache Tomcat

***Table 3.5**: List of different servers and JAR files*

Creating the deployment descriptor (web.xml file)

The deployment descriptor is an XML file, from which the web container gets the information about the Servlet to be invoked. The web container uses the parser to get the information from the **web.xml** file. There are many XML parsers such as SAX, DOM, and Pull. For a Servlet or an HTML page (that might contain a link to a Servlet) to be accessible from the client, it has to first be deployed on the webserver.

NOTE: The load-on-startup element of Servlet in web.xml is used to load the Servlet at the time of deploying the project or the server to start. This saves time for the response of the first request. If you pass the positive value, the lower integer value Servlet will be loaded before the higher integer value Servlet. In other words, the container loads the Servlets in the ascending order of the integer values. The 0 value will be loaded first, then 1, 2, 3, and so on.

The description of the basic element used in the **web.xml** file is as follows:

- **<web-app>** represents the whole application.
- **<Servlet>** is a sub-element of **<web-app>** and represents the Servlet.
- **<Servlet-name>** is a sub-element of **<Servlet>** and represents the name of the Servlet.
- **<Servlet-class>** is a sub-element of **<Servlet>** and represents the class of the Servlet.
- **<Servlet-mapping>** is a sub-element of **<web-app>**. It is used to map the Servlet.
- **<url-pattern>** is a sub-element of **<Servlet-mapping>**. This pattern is used on the client-side to invoke the **Servlet. web.xml**, as follows:
- **<load-on-startup>1</load-on-startup>.** Servlet with less number will be loaded first

Web.xml

```
<web-app>
    <Servlet>
        <Servlet-name>FirstServlet</Servlet-name>
        <Servlet-class>FirstServlet</Servlet-class>
    </Servlet>
    <Servlet-mapping>
        <Servlet-name>FirstServlet</Servlet-name>
        <url-pattern>/FirstServlet</url-pattern>
    </Servlet-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

Now, start the server and run the Servlet.

The output of the Servlet is shown in *Figure 3.3*:

***Figure 3.3**: Output of the Servlet*

2. **The following is an example of a servlet created by extending the GenericServlet class**

   GenericServlet class: It is an abstract class that implements Servlet, ServletConfig, Serializable interface. It has given body of all the methods of servlet interface except service() method. Therefore, a programmer has to give the body of service() method. It is protocol-independent.

Program:

```
import java.io.*;
import javax.Servlet.*;
public class FirstServlet implements Servlet
{
public void service(ServletRequest req,ServletResponse
res)
throws IOException,ServletException
{
res.setContentType("text/html");
PrintWriter out=res.getWriter();
out.print("<html><body>");
out.print("<b>This is First Servlet through Servlet
Interface</b>");
out.print("</body></html>");
}
}
```

HttpServlet: HTTP is a web-specific protocol and has rules for the conversation between the browser requests and the webserver responses. In the next chapter, you will learn about HttpServlet

Conclusion

In this chapter, you learned that Servlets are server-side Java programs that can be deployed on a web server. The Servlet interface provides the basic framework for coding the Servlets. Servlets are portable, extensible, persistent, and robust. The lifecycle of a Servlet is composed of the **`init()`**, **`service()`**, and **`destroy()`** methods. Servlets can be deployed in Glassfish, Weblogic, and Apache Tomcat.

In the next chapter, you will learn about **`HTTPServlet`**. HTTP is a web-specific protocol and has rules for the conversation between the browser requests and the webserver responses. HTTP adds the header information to the top of whatever content is in the response sent by the server. The browser uses that header information to help process the HTML page. **`HTTPServlet`** provides a framework for handling the HTTP protocol, whereas **`GenericServlet`** is protocol-independent.

Questions

1. What is different between the web server and the application server?
2. What are the advantages of Servlets over CGI?
3. Which class is used by a Servlet to receive a request from the client?
4. What is the difference between Applet and Servlet?
5. How do we compile the Servlet?
6. Where does the Servlet store, and where does it run?
7. What are the life cycle methods for a Servlet?
8. What is the use of URL-pattern in web-app.xml?

Chapter 4
HTTP Servlet

Introduction

This chapter focuses on the things related to what the HTTP servlets can do. HyperText Transfer Protocol (HTTP) is a stateless protocol used to make the communication between the browser and the webserver. For example, when a user sends a request to open a specific page, the rules in the HTTP protocol are to be followed. HTTPServlet has implemented all the rules of the HTTP protocol.

Structure

In this chapter, we will cover the following topics:

- HTTP Servlet
- HTTP Request and Response
- Methods of HTTP Servlet
- GET and POST Methods

Objectives

After studying this chapter, you will learn about the HTTP Protocol. You will also understand how HTTPServlet helps us communicate with the web container and handle the HTTP protocols.

HTTP Servlet

It is an abstract class that extends the generic servlet and adds the functionality of the HTTP protocol, as shown in *Figure 4.1*:

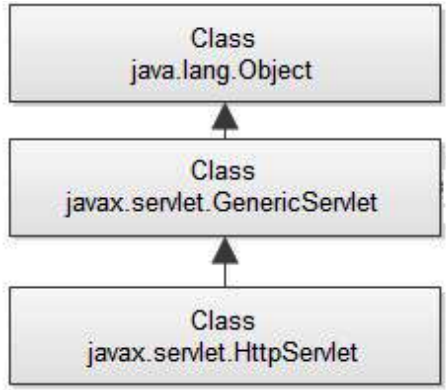


***Figure 4.1**: Hierarchy of HttpServlet class*

Need of HTTPServlet class

A web server takes a client request and gives something back to the browser in the HTML format to display it. When a web server sends an HTML page to the client, it sends it using **HyperText Transfer Protocol** (**HTTP**). HTTP is a web-specific protocol and has rules for conversation between browser requests and web-server responses. HTTP adds the header information to the top of whatever content is in the response sent by the server. The browser uses that header information to help process the HTML page. HTTPServlet provides a framework for handling the HTTP protocol, whereas GenericServlet is protocol-independent.

HTTP Request and HTTP Response

The browser sends a client's request inside the HTTP Request, and the server passes the data inside the HTTP Response, as shown in *Figure 4.2*:

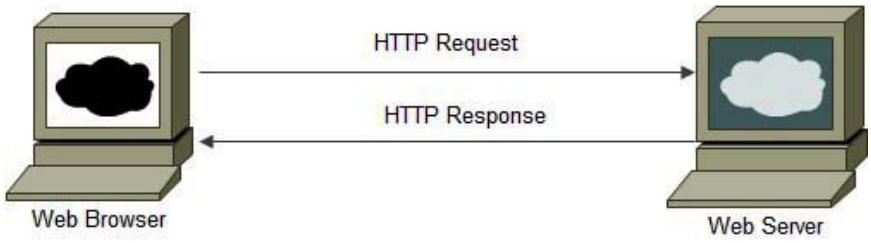


Figure 4.2: *HTTP Request and Response*

HTTP Protocol has several methods. This method name tells the server the kind of request being made. For example, the client sends an **`HTTP Get`** request to the server asking to get the page from the specified resource. When a user sends an **`HTTP Post`** request to the server, the user sends the data to the server or updates a resource. HttpServlet provides various methods of HTTP specific, which are as follows:

- **`public void service(ServletRequest req, ServletResponse res)`**: Dispatches the request to the protected service method by converting the request and response object into the HTTP type.
- **`protected void service(HttpServletRequest req, HttpServletResponse res)`**: Receives the request from the service method and dispatches the request to the **`doXXX()`** method depending on the incoming HTTP request type.
- **`protected void doGet(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`GET`** request. The web container invokes it.
- **`protected void doPost(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`POST`** request. The web container invokes it.
- **`protected void doHead(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`HEAD`** request. The web container invokes it.
- **`protected void doOptions(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`OPTIONS`** request. It is invoked by the web container.
- **`protected void doPut(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`PUT`** request. It is invoked by the web container.

- **`protected void doTrace(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`TRACE`** request. It is invoked by the web container.
- **`protected void doDelete(HttpServletRequest req, HttpServletResponse res)`**: Handles the **`DELETE`** request. It is invoked by the web container.
- **`protected long getLastModified(HttpServletRequest req)`**: Returns the time when **`HttpServletRequest`** was last modified since midnight January 1, 1970 GMT.

The GET and POST methods

When a client sends a request to the server, the client can also pass additional information with the URL to describe what exactly is required as the output from the server by using the **`GET`** method. The additional sequence of characters that are appended to the URL is called a **query string**. However, the length of the query string is limited to 240 characters. Moreover, the query string is visible on the browser and can therefore be a security risk.

To overcome these disadvantages, the post method can be used. The **`POST`** method sends the data as packets through a separate socket connection. The complete transaction is invisible to the client. The disadvantage of this method is that it is slower than the **`GET`** method because the data is sent to the server as separate packets.

Table 4.1 shows the difference between the GET and POST methods:

Get	Post
Data is limited to 240 characters	A large amount of data can be sent.
Not Secured	Secured.
Can be bookmarked	Cannot be bookmarked.
Idempotent	Non-Idempotent.
Efficient	It is less efficient and slow.

***Table 4.1**: Difference between Get and Post Method*

An **example** to develop an application to keep track of the number of users visiting your web application or website is as follows:

```
import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
  /**Counter to keep track  of the number of users visiting
the web site */
public class HitCountServlet extends HttpServlet {
static int count;
  public void init(ServletConfig config) throws ServletException
  {
super.init(config);
   }
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        count++;
        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>Servlet HitCountServlet</
title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1> You are user Number"+String.
valueOf(count)+"Visiting our  website</h1>");
            out.println("</body>");
            out.println("</html>");
        }    }

    @Override
    public String getServletInfo() {
        return "Hit count servlet";}
}
```

The output for the preceding example is shown in *Figure 4.3:*

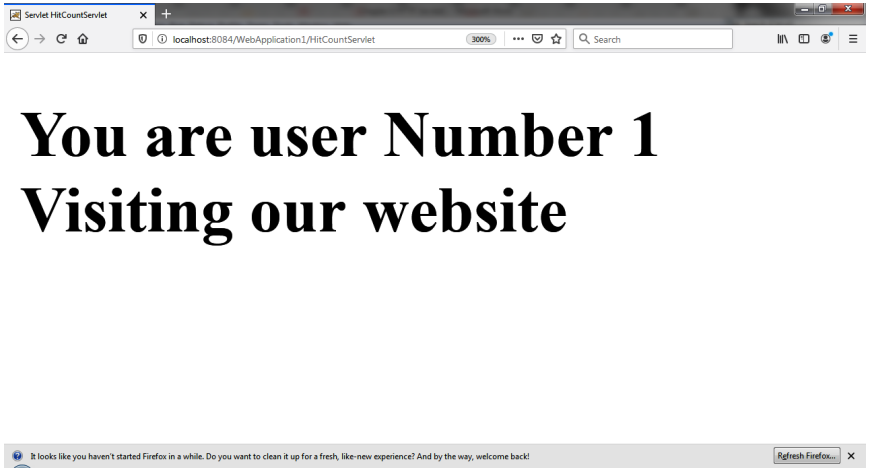


***Figure 4.3**: Result of the preceding program*

In this example, the following methods are to be used:

- The **`init()`** method needs to be coded in the ***HitCountServlet*** class to initialize the hit counter to zero. This method gets invoked automatically when the servlet is loaded into the memory. The servlet gets automatically loaded on the web container either when the server is started or installed manually using some administrative tools.
- The **`doGet()`** method needs to be coded to increment the hit counter whenever a client browser requests for the **http://localhost:8084/WebApplication1/HitCountServlet** home page.

NOTE: If the client does not explicitly specify the request type, then by default, the doGet() method is invoked.

HttpServletRequest Interface

This interface extends the **`ServletRequest`** interface to provide the request information for a servlet, such as content type, content length, parameter names and values, header information, attributes, etc. It is in the **`Javax.servlet.http`** package.

Method of ServletRequest Interface

Table 4.2 shows the methods of the ServletRequest interface:

Method	Description
`public String getParameter(String name)`	It is used to obtain the value of a parameter by name.
`public String[] getParameterValues(String name)`	It returns an array of String containing all the values of the given parameter name. It is mainly used to obtain the values of a multi-select list box.
`java.util.Enumeration getParameterNames()`	It returns an enumeration of all of the request parameter names.
`public int getContentLength()`	It returns the size of the request entity data or -1 if not known.
`public String getCharacterEncoding()`	It returns the character set encoding for the input of this request.
`public String getContentType()`	It returns the Internet Media Type of the request entity data or null if not known.
`public ServletInputStream getInputStream() throws IOException`	It returns an input stream for reading binary data in the request body.
`public abstract String getServerName()`	It returns the host name of the server that received the request.
`public int getServerPort()`	It returns the port number on which this request was received.

***Table 4.2**: Methods of ServletRequest Interface*

With the help of the **`ServletRequest`** interface, the server can get the parameter pass by the user, process it, and send the result back to the user.

This can be explained with the help of an **example** of the login form where the user enters the user name and password. Servlet has used the **`getParameter`** method that returns the value username and password and checks whether the username and password are valid or not.

Example

index.html:

```
<!DOCTYPE html>
<html>
<body>
<h1>The Login Form</h1>
<form method="post" action="/LoginForm/actionpage">
<label for="fname">User name:</label>
<input type="text" id="uname" name="Uname"><br><br>
<label for="lname">Password:</label>
<input type="text" id="pwd" name="pwd"><br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Actionpage.java:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Actionpage extends HttpServlet {
 public void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use
following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Actionpage</title>");
```

```
            out.println("</head>");
            out.println("<body>");
            String name=request.getParameter("Uname");
            String pwd=request.getParameter("pwd");
            if(name.equals("Sarika") && pwd.equals("vivek"))
                out.println("<h1>You are successfully
login </h1>");
            else
                out.println("<h1>User Name or Password is
Incorrect </h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
     @Override
    public String getServletInfo() {
        return "Login Form ";
    }}
```

Web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
    <servlet>
        <servlet-name>Actionpage</servlet-name>
        <servlet-class>Actionpage</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Actionpage</servlet-name>
        <url-pattern>/actionpage</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

The following is the output of `index.html` as shown in *Figure 4.4*:

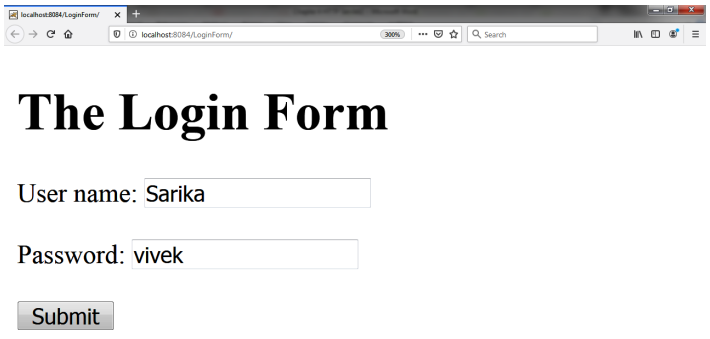


***Figure 4.4**: Output of index.html*

The following is the output of the action page as shown in *Figure 4.5 and Figure 4.6*:

ActionPage

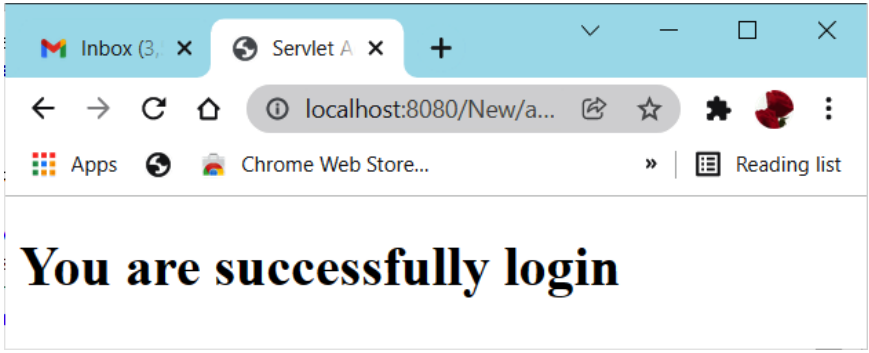


***Figure 4.5**: Output of Action Page*

Action Page: (If either user name or password is incorrect):

***Figure 4.6**: Output of Action Page*

Conclusion

In this chapter, you learned that HTTP stands for HyperText Transfer Protocol, and it is a network protocol used on the web. It runs on top of TCP/IP. HTTPServlet is an abstract class that extends the generic servlet and adds the functionality of the HTTP protocol. HTTP uses the request/response model – the user makes a request, and the webserver gives an HTTP response to the browser, and the browser displays it to the user. If the response from the server is an HTML Page, the HTML is added to the HTTP Response. HTTPServletRequest and HTTPServletResponse are two interfaces used to exchange the information between the user and the webserver. An HTTP request includes the request URL. The additional sequence of characters that are appended to the URL is called a query string. The GET request appends the query string at the end of the URL. The POST method appends the query string or data in the body of the request.

In the next chapter, you will learn about the sessions as HTTP cannot remember the activity of the previous page. HTTP is a stateless protocol. This means it does not have any information regarding the HTTP page. To remember the state or send the information from one page to another, we need to maintain the sessions.

Questions

1. What is the difference between the Get and Post methods?
2. What is the difference between GenericServlet and HttpServlet?
3. How is PrintWriter different from ServletOutputStream?
4. Which HTTP method is non-idempotent?

Multiple Choice Questions

1. **What type of servlets use these methods – doGet(), doPost(),doHead, doDelete(), doTrace()?**
    a. Generic Servlets
    b. HttpServlets
    c. All of the above
    d. None of the above

    Answer: HttpServlets

2. **Web server is used for loading the init() method of the servlet.**
   a. True
   b. False

   Answer: True

3. **Which packages represent the interfaces and classes for servlet API?**
   a. javax.servlet
   b. javax.servlet.http
   c. Both A & B
   d. None of the above

   Answer: javax.servlet

4. **Which class can handle any type of request so that it is protocol-independent?**
   a. GenericServlet
   b. HttpServlet
   c. Both A & B
   d. None of the above

   Answer: Generic Servlet

5. Which HTTP request method is non-idempotent?
   a. GET
   b. POST
   c. BOTH A & B
   d. None of the above

   Answer: POST

6. **Which object is created by the web container at the time of deploying the project?**
   a. ServletConfig
   b. ServletContext
   c. Both A & B
   d. None of the above

   Answer: Servlet Context

Chapter 5
Working with Servlet Sessions

Introduction

HTTP cannot remember the activity of the previous page. HyperText Transfer Protocol (HTTP) is a stateless protocol. This means it does not have any information regarding the HTTP page. To remember the state or send the information from one page to another, we need to maintain the sessions.

Structure

In this chapter, we will cover the following topics:

- Session tracking
    - URL rewriting
    - Hidden form fields
    - Cookies
    - HTTPSession interface

Objective

After studying this chapter, you will understand how to make sessions and remember each user preference.

Session tracking

A session is a group of activities performed by a user while accessing a particular website. The process of keeping track of the settings across the sessions is called **session tracking**. Consider the example of an online shopping mall. The user can choose a product and add it to the shopping cart. When the user moves to a different page, the details in the shopping cart are still retained so that the user can check the items in the shopping cart and then place the order.

Session tracking can also be used to keep track of the user's preferences. For example, if the user selected the novels, then more novels are displayed to the user.

Techniques to keep track of sessions in servlets

By default, the data across the sessions cannot be stored by using HTTP because it is a stateless protocol. However, certain techniques help store the session data by using HTTP.

They are as follows:

- URL Rewriting
- Hidden form Fields
- Cookies
- HTTPSession interface

URL Rewriting

This is a technique by which the URL is modified to include the session ID of a particular user and is sent back to the client. In any subsequent transaction, the client is forced to use the session ID when it sends a request to the server. If a session ID does not exist, then a session ID is created and used in subsequent communication between the client and the server.

The following is an example:

Original URL:

```
http://<host address>:<port number >/servletcontext/
sampleservlet
```

We can send the parameter values and names with the original URL.

Syntax:

```
Original URL?name1=value1&name2=value2
```

The following is an example:

```
http://<host address>:<port number>/servletcontext/
sampleservlet?name=sarika&pwd=sailboat
```

The advantages of URL Rewriting are as follows:

- It will always work whether a cookie is disabled or not (browser independent).
- No extra form is required.

The disadvantages of URL Rewriting are as follows:

- Works with links.
- Text information can only be sent.

The following is an example:

User Name is available on the **`next2`** page by using URL rewriting.

`Index.html`:

```
<!DOCTYPE html>
<html>
<body>
<h1>The Login Form</h1>
<form method="post" action="/Sessions/next">
<label >User name:</label>
<input type="text" id="uname" name="Uname"><br><br>
<label for="lname">Password:</label>
<input type="password" id="pwd" name="pwd"><br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Next Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class URLRewritingExample extends HttpServlet {
    protected void doPost(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>Servlet
URLRewritingExample</title>");
            out.println("</head>");
            out.println("<body>hello    ");
            String name=request.getParameter("Uname");
            String pwd=request.getParameter("pwd");
             if(name.equals("Sarika") && pwd.
equals("vivek"))
             {
out.println("<h1>"+name+" are successfully login </h1>");
              out.print("<a
href='next2?name="+name+"'>visit</a>");
}
            else
                out.println("<h1>User Name or Password is
Incorrect </h1>");
            out.println("</body>");
            out.println("</html>");
        }   }}
```

Next2Servlet:

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

public class NEXT2 extends HttpServlet {
    protected void doGet(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>Servlet NEXT2</title>");
            out.println("</head>");
            out.println("<body>");
        String n=request.getParameter("name");
            out.println("<h1>Hello "+n +"</h1>");
            out.println("</body>");
            out.println("</html>");
        } }}
```

The following is the result for index.html, as shown in *Figure 5.1*:

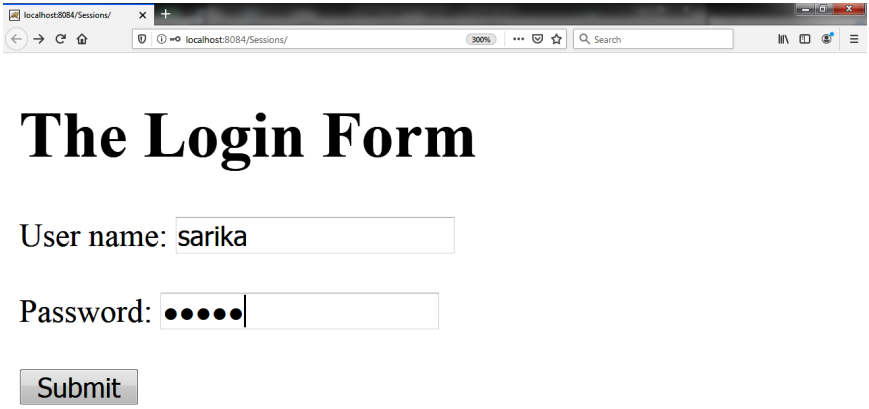
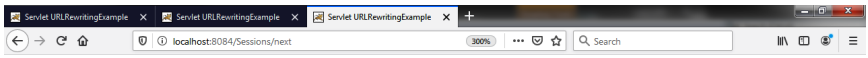


***Figure 5.1**: Output of the preceding program (index.html)*

The following is the result for the next servlet, as shown in *Figure 5.2*:

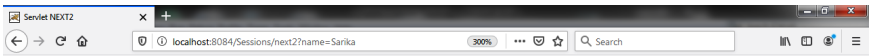


Hello Sarika ! you are successfully login

visit

***Figure 5.2**: Output of Next servlet*

When you click on a visit, you receive the following output, as shown in *Figure 5.3*:



Hello Sarika

***Figure 5.3**: Output of Next 2 Servlet*

Hidden Form Fields

This is one of the techniques that can be used to keep track of the users by placing the hidden fields in a form. The values that have been entered in these fields are sent to the server when the user submits the form. As far as the server is concerned, there is no difference between the hidden form fields and the other fields in the form.

Let us consider the same example of a shopping mall. The items that the user selects can be recorded by using the hidden form fields and submitted to the server for processing the details.

The following is an example:

`Hello.html`:

```
<HTML>
        <TITLE>A Form with Hidden Fields></TITLE>
        <BODY>
```

```
          <FORM>
              <INPUT TYPE = “HIDDEN”  NAME= “text1”></FORM>
       </BODY>
</HTML>
```

In this program, we store the user's name in a hidden text field and get that name from the second servlet, as shown as follows:

index.html

```
<html>
<head>
<title>Hidden Form Field</title>
</head>
<body>
<form action="first">
 Name:
<input type="text" name="userName"/><br/>
<input type="submit" value="submit"/>
</form>
</body></html>
```

First Servlet(ActionServlet1):

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ActionServlet1 extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ActionServlet1</
```

```
title>");
            out.println("</head>");
            out.println("<body>");
            String name=request.getParameter("userName");
            out.print("Welcome "+name);
            out.print("<form action='second'>");
            out.print("<input type='hidden' name='uname'
value='"+name+"'>");
            out.print("<input type='submit'
value='Next'>");
            out.print("</form>");
            out.println("</body>");
            out.println("</html>");
        }}}
```

Second Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Second extends HttpServlet {
    protected void doGet(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>Servlet Second</title>");
            out.println("</head>");
            out.println("<body>");
                 String name=request.getParameter("uname");
                out.print("Hello "+name);
```

```
        out.println("</body>");
        out.println("</html>");          }}}
```

The output for **`index.html`** is shown in *Figure 5.4*:

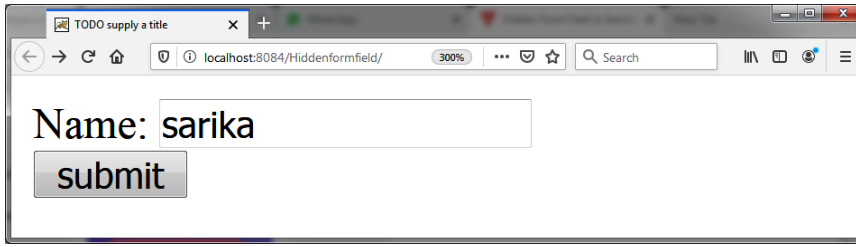


***Figure 5.4**: Output of the above program*

The First Servlet stores the name using the hidden form field, as shown in *Figure 5.5*:

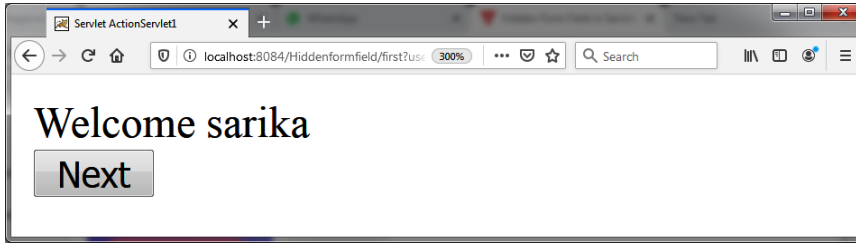


***Figure 5.5**: Output of ActionServlet1 (first) Servlet*

The Second Servlet accesses the name from the hidden field, as shown in *Figure 5.6*:

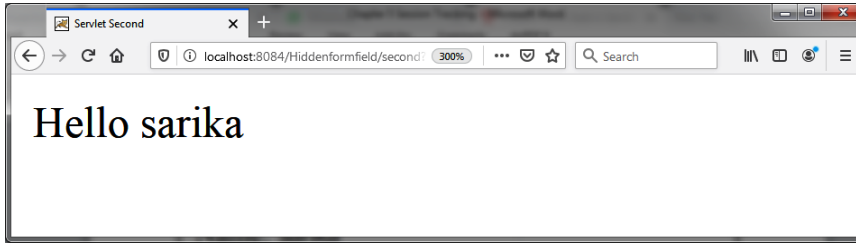


***Figure 5.6**: Output of the Second servlet*

Using the HttpSession Interface

The Servlet API consists of a few classes and interfaces that help implement session tracking in servlets. The Java Servlet API provides

an interface called **`HttpSession`** that can be used to keep track of the sessions in the current servlet context.

Every user who logs onto a website is automatically associated with an **`HttpSession`** object. The servlet can use this object to store the information about the user's session. The **`HttpSession`** object enables the user to maintain two types of data – state and application.

The state data is used to maintain and retrieve the details about the user's connection. A user's connection details could be the time at which the session was created or last accessed. The application data is used to store the details, such as the items that were added to the shopping basket by the user. The application data can be manipulated by using the **`getValue()`** and the **`putValue()`** methods of the **`HttpSession`** interface, as shown in *Table 5.1*:

Method name	Functionality
`getSession()`	This method is used to retrieve the current HTTP session that is associated with the user. If a session does not exist, then a session can be created by using `getSession(true)`.
`Object getValue (String name)`	This function is used to retrieve the value in a session object. For example, if you have a session object named item selected, then you can retrieve the value in the session object by using `getValue(itemselected)`.
`Void putValue (String name, Object value)`	This function is used to add an item to the session.
`boolean isNew ()`	This function returns a true value if a new session ID has been created and has not been sent to the client.
`String getlD ()`	This function returns the session ID. If URL rewriting is used, the session can be retrieved using this function and padded onto the client.
`long getCreationTime ()`	This function returns the time in milliseconds when the session was first created.

`long getLastAccessedTime ( )`	This function returns the previous time a request was made with the same session ID. The return value of this function is used by the session manager to optimize the activity of the server.

***Table 5.1**: Methods of HttpSession Interface*

Program

In this program, the attribute was set in the first servlet's session scope and got that value from the session scope by the second servlet. To set the attribute in the session scope, we used the **`setAttribute()`** method of the **`HttpSession`** interface, and to get the attribute, we used the **`getAttribute`** method, as shown as follows:

```
<html>
<head>
<title>Http Session</title>
</head>
<body>
<form action="First">
                Name:
<input type="text" name="userName"/><br/>
<input type="submit" value="submit"/>
</form>
</body>
</html>
```

First Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class First extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
```

```
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use
following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet First</title>");
            out.println("</head>");
            out.println("<body>");
            String name=request.getParameter("userName");
            out.print("Welcome "+name);
            HttpSession session=request.getSession();
            session.setAttribute("uname",name);
            out.print("<br><a href='Second'>visit</a>");
            out.println("</body>");
            out.println("</html>");
        }}}
```

Second Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class Second extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use
following sample code. */
            out.println("<!DOCTYPE html>");
```

```
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Second</title>");
        out.println("</head>");
        out.println("<body>");
        HttpSession session=request.getSession(false);
        String n=(String)session.getAttribute("uname");
        out.print("Hello "+n);
        out.println("</body>");
        out.println("</html>");
    }}}
```

The following is the result for `index.html`, as shown in *Figure 5.7*:

***Figure 5.7**: Output of the preceding Program (index.html)*

The following is the First Servlet output, as shown in *Figure 5.8*:

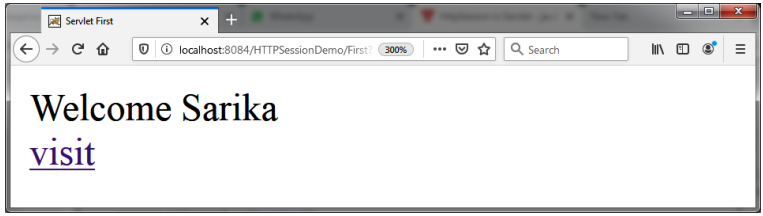


***Figure 5.8**: Output of First Servlet*

The following is the Second Servlet output, as shown in *Figure 5.9*:

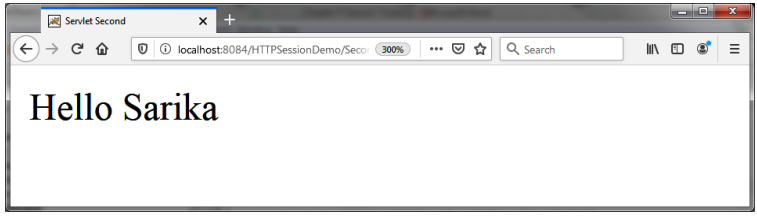


***Figure 5.9**: Output of Second Servlet*

Cookies

Cookies are small text files used by a web server to keep track of the users. A cookie has values in the form of *key-value* pairs. They are created by the server and sent to the client with the HTTP response headers. The client saves the cookies in the local hard disks and sends them along with the HTTP request headers to the server. If a cookie with the same name already exists, then the key is overwritten with the new value. A server can send one or more cookies to the client. A web browser, which is the client software, is expected to support 20 cookies per host, and the size of each cookie can be a maximum of 4 bytes each.

The following are the characteristics of cookies:

- Cookies are only sent back to the server that created them and not to any other server. For example, if a cookie was created by the webserver and sent to the client or the browser, the cookie can be sent back to the same server only.
- The server can use cookies to find out the computer name, IP address, or any other details of the client computer.

The following are the advantages of Cookies:

- Cookies are the simplest technique of maintaining the state.
- Cookies are maintained at the client-side.

The following are the disadvantages of Cookies:

- It will not work if the cookie is disabled from the browser.
- Only textual information can be set in the Cookie object.

The javax.servlet.http.Cookie class

The Servlet API provides a class called Cookie, which is used to represent a cookie. The **`Cookie`** class is used for implementing session tracking in servlets. The values of the cookies are stored in the client computers. As discussed earlier, a cookie has a name, which is referred to as a key, and the data stored in the cookie is referred to as value.

A cookie is sent by the client through an **`HttpServletRequest`** object. The servlet sends the cookie to the client through an **`HttpServletRequest`** object.

The constructor of the Cookie class

- `Cookie()`: Constructs a new cookie.
- `Cookie(String name, String value)`: Constructs a cookie with a specified name and value.

Table 5.2 lists some of the methods of **`javax.servlet.http.Cookie`**, **`HttpServletResponse`**, and **`HttpServletRequest`** class that are used to implement the concept of session tracking using cookies:

Method Name	Functionality
`Cookie.Cookie (String. String)`	The constructor of the Cookie is a class used to create a cookie and assign a value to it.
`Cookie.getValue (String name)`	Each cookie that gets created by a servlet is given a name and value. This function returns the value stored in the given cookie.
`Cookie.setValue (string)`	This function is used to assign a value of a type string to the given cookie.
`Cookie.getName()`	This method is used to retrieve the name of a cookie.
`Cookie.setMaxAge (int)`	This method is used to specify the maximum amount of time for which the client browser retains the cookie value.
`Cookie.setName(String name)`	It changes the name of the cookie and sets the name of the cookie.
`HttpServletResponse. addcookie ()`	Cookies are created by the server and passed onto the client through an `HttpServletResponse` object. The `addCookie ()` method is used to add a cookie to the response. This method can be called more than once to add different cookies to the response that is sent to the client.

Method Name	Functionality
`HttpServletRequest. getCookie()`	The client sends the data to the server in the form of a request received by the server in the form of an `HttpServletRequest` object. This method is used to retrieve the cookie values in the request.
`public Cookie[] getCookies()`	It is a method of the `HttpServletRequest` interface used to return all the cookies from the browser.

***Table 5.2**: Method of Cookie, HttpServletResponse, and HttpServletRequest class*

The following is an example:

In this program, we store the user's name in the cookie object by the First servlet and access it in the second servlet.

`Index.html`:

```
<html>
<head>
<title> Cookie Demo</title>
</head>
<body>
<form action="First">
 Name:
<input type="text" name="userName"/><br/>
<input type="submit" value="submit"/>
</form>
</body>
</html>
```

First Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class First extends HttpServlet {
    protected void doGet(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>Servlet First</title>");
            out.println("</head>");
            out.println("<body>");
           String name=request.getParameter("userName");
            out.print("Welcome "+name+"  in the Cookie Demo");
            Cookie ck=new Cookie("uname",name);//creating
cookie object
             response.addCookie(ck);//adding cookie in the response
            out.print("<form action='Second'>");
             out.print("<input type='submit'
value='next'></form>");
            out.println("</body>");
            out.println("</html>");}}}
```

Second Servlet:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Second extends HttpServlet {

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

```
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use
following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Second</title>");
            out.println("</head>");
            out.println("<body>");
             Cookie ck[]=request.getCookies();
             out.print("Hello "+ck[0].getValue());
            out.println("</body>");
            out.println("</html>");
        }}}
```

The following is the result for index.html, as shown in *Figure 5.10*:

***Figure 5.10**: Output of above program (index.html)*

The following is the First Servlet output, as shown in *Figure 5.11*:

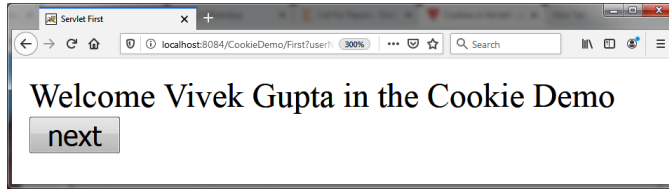


***Figure 5.11**: Output of First Servlet*

The following is the Second Servlet output, as shown in *Figure 5.12*:

***Figure 5.12**: Output of Second Servlet*

Conclusion

A session is a group of transactions that happens between the server and client over some time. Any of the following methods can implement session tracking:

- URL Rewriting
- Hidden Form fields
- Cookies
- Using the HttpSession interface

In the next chapter, we will learn about Inter-servlet communication. It means communication between servlets of a web application in the same server. Servlets that are present on the same web server can communicate and share resources, such as variables, amongst each other. For example, when a user logs on to a website, the user authentication can be done by a servlet.

Questions

1. Explain Session and its importance?
2. What is Session Tracking? What are the different types of Session Tracking?
3. Why do you use Session Tracking in HttpServlet?
4. What is the advantage of Cookies over URL rewriting?
5. What is the use of Cookie and Session? What is the difference between them? How can you destroy the Session in Servlet? (Hint: You can call invalidate() method on the session object to destroy the Session. For example, session.invalidate();)
6. How do you track a user session in Servlets?

Chapter 6
Inter-Servlet Communication

Introduction

Inter-servlet communication means communication between servlets of a web application in the same server. Servlets on the same web server can communicate and share resources, such as variables, amongst each other. For example, when a user logs on to a website, user authentication can be done by a servlet. A different servlet can perform information processing after the user is authenticated. Dividing the tasks across the servlets also helps in implementing a structured approach to implementing tasks. Communication between servlets can be implemented by using the RequestDispatcher interface.

Structure

In this chapter, we will cover the following topics:

- RequestDispatcher
- Servlet Context
- Method to get the object of RequestDispatcher

- Methods of RequestDispatcher interface
- sendRedirect() method of HttpServletResponse interface

The RequestDispatcher Interface

The **`RequestDispatcher`** interface is used to forward or delegate a request from a servlet to other resources, such as a servlet, an HTML file, or a JSP page. In this case, the source servlet does some processing and delegates the request to another servlet. Moreover, this task delegation happens within servlets in the same servlet context.

A servlet context is a directory in which the servlets are deployed in the webserver. Servlets that execute in the same server belong to the same servlet context. However, few web servers also enable the creation of more than one servlet context. You can get the reference of **`ServletContext`** with the help of the method of the **`servletConfig`** interface.

Syntax: **`public ServletContext getServletContext();`**

This function **`getServletContext()`** method of the **`ServletConfig`** interface is used to obtain a reference to the servlet context in which a servlet executes.

Method of ServletContext Interface

The following are the methods of the ServletContext interface:

1. **`public abstract void setAttribute(String name, Object object)`**: This function is used to set a value to an attribute available in the servlet Context. For example, **`setAttribute("name," "Ram")`** is used to set the value of an attribute called **`name`**. In addition, this function can also be used to create an attribute that can be accessed by all the servlets in the same servlet context.
2. **`Public abstract Object getAttribute(String name)`**: This function is used to obtain the value of an attribute.

Method to get the object of RequestDispatcher

The **`RequestDispatcher`** interface encapsulates the URL of a resource (a servlet, a JSP page, or a **`.html`** file) that exists in a particular

servlet context. The **`getRequestDispatcher()`** method of the **`ServletRequest`** interface returns the object of **`RequestDispatcher`**.

Syntax: **`public RequestDispatcher getRequestDispatcher(String urlpath);`**

This method is used to get a reference to a servlet through a URL specified as the parameter. The dispatcher that is returned is used to invoke the servlet. If the dispatcher cannot be obtained for the URL specified, this function returns null.

Methods of RequestDispatcher interface

The following are the methods of the RequestDispatcher interface:

- **`public abstract void forward(HttpServletRequest request, HttpServletResponse response)`** throws ServletException, IOException: This method is used to delegate a task to the resource encapsulated by a particular interface object. It forwards a request from one servlet to another. This method must be used when the output is completely generated by the second servlet or the servlet that is invoked. If the **`PrintWriter`** object is already accessed by the first servlet, then this method throws an exception.
- **`public abstract void includes (ServletRequest request, ServletResponse response)`** throws ServletException, IOException: This function is also used to invoke one servlet from another like the **`forward ()`** function; however, you can also include the output of the first servlet with the current output. The first servlet can make use of the **`PrintWriter`** object even after calling the **`include()`** method.

NOTE:

1. **Servlet chaining is a technique by which multiple servlets are executed in a specific sequence to complete a transaction. A servlet can invoke another servlet to perform the next step in a transaction. The output of the previous servlet is sent back to the browser or the client. This technique can be compared to the concept of pipes in Linux, where the output of one command is redirected to the other.**

2. **In the process of Servlet Chaining: Firstly, we must create a RequestDispatcher for a resource that has to be chained. Then, must set the attribute values for the request if required. Next, we need to call the forward() method or the include() method on a RequestDispatcher object.**

Implementing Inter servlet communication via a problem statement

You need to deposit the money into your account. Create a servlet that accepts and validates the account number and PIN of a user. The first servlet is used to validate the account number and the PIN. If the values are valid, an attribute called account number is created and assigned and the account number is entered. Otherwise, an error message is displayed to the user and includes the index.html. If the account number and PIN are valid, then it forwards the request to the second servlet and is invoked.

Tasklist

- **Write the client interface (`index.html`)**: The HTML form accepts the account number and PIN from the user.
- **Write the code of two servlets**: After clicking on the submit button, the servlet will be invoked, validating the account number and PIN, and forwarding the request to the second servlet. Then, the second servlet grants the permission to deposit or withdraw the amount.
- **Compile and deploy the servlets**: Make the entry of the servlet in deployment descriptor **`web.xml`**. Compile and run the program.
- **Verify the functionality of servlets**: Verify whether the program displays a valid result or not.

To solve the problem, we will use the **`RequestDispatcher`** interface. The problem states that the first servlet needs to accept the account number and the PIN from the user and validate them. If the account number is valid, then the second servlet needs to be invoked.

Client Interface

The HTML form contains two textboxes for accepting the account number and the PIN, respectively. Upon clicking on the submit button, the first servlet is invoked.

index.html

```
<html> <head>
        <title>Our Bank</title>
        </head>
    <body><div><CENTER>
            <H1>MY BANK </h1></CENTER>
            <FORM action="firstServlet">
                <table>
                    <tr>  <td>   Enter the account Number   </td>
                        <td>  <input type="text"
name="accnum">  </td>
                    </tr>
                      <tr> <td>   Enter the Pin Number </td>
                        <td> <input type="password"
name="pinnum"> </td>
                    </tr>                 </table>
                <br> <center>  <input type=SUBMIT
value="SUBMIT"></center>
        </div></body></html>
```

The following is the output, as shown in *Figure 6.1*:

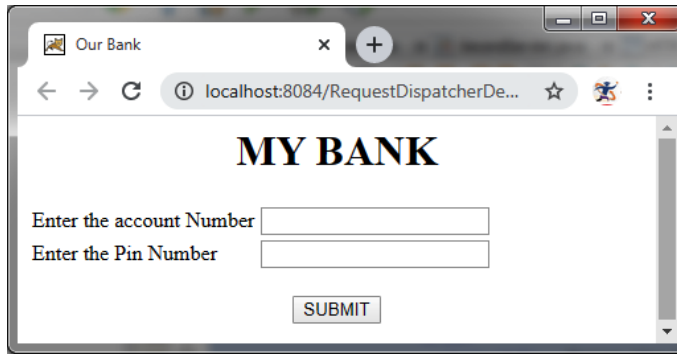


***Figure 6.1**: Output of Index.html (Client Interface)*

Code of FirstServlet

The first servlet is used to validate the account number and the PIN. If the values are valid, an attribute called account number is created, and the assigned account number is entered. Otherwise, an error message is displayed to the user. If the account number and PIN are valid, then it forwards the request to the second servlet that is invoked, as shown as follows:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FirstServlet extends HttpServlet {
    protected void doGet(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>Servlet FirstServlet</title>");
            out.println("</head>");
             out.println("<body>");
             String n=request.getParameter("accnum");
String p=request.getParameter("pinnum");
             if(p.equals("201306")&&n.
equals("AJ123456789"))
{
       RequestDispatcher rd=request.
getRequestDispatcher("servlet2");
        rd.forward(request, response);
 }
```

```
            else
{
 out.print("Sorry account Number or Password Error!");
        RequestDispatcher rd=request.
getRequestDispatcher("/index.html");
             rd.include(request, response);
}
           out.println("</body>");
           out.println("</html>");
         }catch(Exception e){System.out.
println("Exception");}
}}
```

The following is the Second servlet code:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class SecondServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
           throws ServletException, IOException {
        try  {
             response.setContentType("text/
html;charset=UTF-8");
             PrintWriter out = response.getWriter();
           out.println("<!DOCTYPE html>");
           out.println("<html>");
           out.println("<head>");
           out.println("<title>Servlet SecondServlet</title>");
           out.println("</head>");
           out.println("<body >");
           String n=request.getParameter("accnum");
           out.print("<font color=BLUE,size=20>Welcome
"+n+"<br><center>");
```

```
            out.print(" You can deposit/ withdraw the
amount ");
            out.println("</body>");
            out.println("</html>");
        }catch(Exception e){System.out.println("Exception
occured"+e);}
    }}
```

Output

The following is the output if the Account Number and PIN are valid, as shown in *Figure 6.2*:

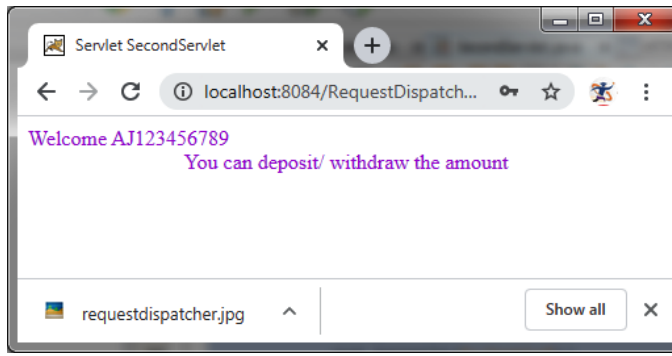


***Figure 6.2**: Output if the Account Number and PIN are valid*

The following is the output if the Account Number and PIN are invalid, as shown in *Figure 6.3*:

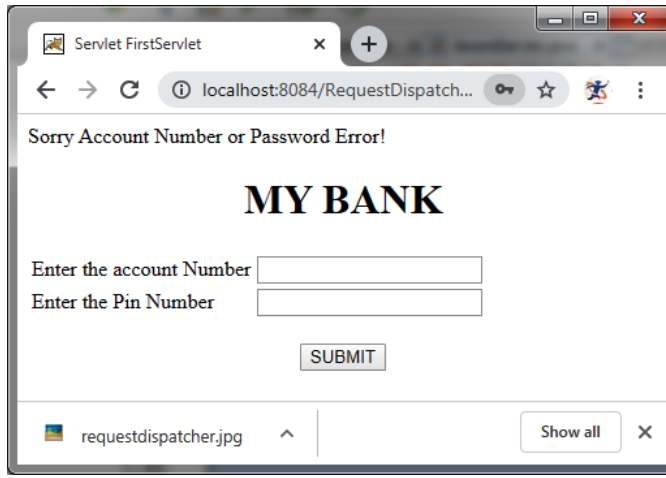


***Figure 6.3**: Output if the Account Number and PIN are invalid*

SendRedirect

In the **`include()`** and **`forward()`** method of the **`RequestDispatcher`** class, the servlet is not making a new request. It only includes the request in the same servlet or forwards it into another servlet.

The **`client (browser)`** wants to create a new request to get to the resource by using the **`sendRedirect()`** method; the user can see the new URL. It takes both relative and absolute paths.

NOTE:

sendRedirect() works on the response object while the request dispatch works on the request object. The sendRedirect is a method of the HttpServletResponse interface.

Syntax:

```
public void sendRedirect(String URL)throws IOException;
```

Example

In this example, we are redirecting the request to the **`instituteoflearning`** server, as shown as follows:

SendRedirectExample:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class SendRedirectExample extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
          throws ServletException, 0
Exception {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            response.sendRedirect("http://
```

```
instituteoflearning.in/");
        }
        finally {
            out.close();
        }
    }}
```

Difference between forward() and sendRedirect() method

Table 6.1 shows the difference between forward() and sendRedirect() methods:

forward()	sendRedirect()
It is a method of the `RequestDispatcher` interface.	It is a method of the `HttpServletResponse` interface.
Works on server-side within the server.	Works on the client-side both outside and within the server.
Sends the same request and response to another servlet.	Sends a new request to another servlet.

***Table 6.1**: Difference between forward() and sendRedirect() method*

Conclusion

In this chapter, we learned that the RequestDispatcher interface could be used to call one servlet from the other.

The `forward()` and the `include()` methods can invoke one servlet from the other. The data common to the servlets can be accessed by using the ServletContext interface.

Servlet chaining is a very simple process in which we give the output of one servlet as an input for another servlet. sendRedirect is a method of the HttpServletResponse interface, which creates a new request and sends it to another servlet.

In the next chapter, we will study Java Server Pages (JSP). JSP technology has facilitated the segregation of the work profiles of a web designer and a web developer. A web designer can design and formulate the layout for the web page by using HTML. On the other hand, a web developer, working independently, can use the Java code and other JSP-specific tags to code for the business logic.

Questions

1. Which function must be used to add a cookie to the response that is sent by the server to the client?
2. What is a Request Dispatcher?
3. What is sendRedirect? Differentiate between forward() and sendRedirect() method.
4. What is the process of servlet chaining?
5. What is the process to get the object of Request Dispatcher?

Chapter 7
Java Server Pages (JSP)

Introduction

With the advent of the Internet, the monolithic application architecture changed to a multi-tiered client-server architecture. The need for server-side scripting gradually began to dominate the aspects of web programming. Microsoft introduced Active Server Pages (ASP) to capture the market demand for server-side scripting. Working on similar lines, Sun Microsystems (now taken over by Oracle) released Java Server Pages (JSP) to add the server-side programming functionality. It can be considered an advancement of the Servlet, as it adds more functionality than Servlet, like custom tags and expression language (these will be discussed in subsequent chapters). We can write the HTML tags along with the JSP tags. But in Servlet, we embed the HTML tag with the help of the out.println() method of ServletResponse interface. JSP is platform-independent, whereas ASP is not platform-independent.

Structure

In this chapter, we will cover the following topics:

- Need and features of JSP
- Difference between Servlet and JSP
- The life cycle of JSP: Example of JSP
- Directory Structure of JSP: Structure of Web.xml

Objectives

After studying this chapter, you will understand the life cycle methods of JSP. After compilation, JSP is converted into Servlet and incorporates all functionalities of Servlet. You will understand the differences between JSP and Servlet and the advantages of JSP.

Need for JSP

A typical web application consists of the presentation logic representing the static content used to design the web page structure in terms of the page layout, color, and text. The business logic or the dynamic content involves the application of business intelligence and diagnostics in terms of financial and business calculations. When developing a web application, time is often lost in situations where the developer is required to code for the static content.

The JSP technology has facilitated the segregation of the work profiles of a web designer and a web developer. A web designer can design and formulate the layout for the web page by using HTML. On the other hand, a web developer, working independently, can use the Java code and other JSP-specific tags to code for the business logic. The simultaneous construction of static and dynamic content facilitates the development of quality applications with increased productivity.

A JSP page, after compilation, generates a servlet, and therefore, incorporates all the servlet functionalities. Servlets and JSP, thus, share common features such as the following:

- They are platform-independent.
- They create database-driven web applications.
- Server-side programming capabilities.
- JSP needs no recompilation.
- In JSP, visual display and logic are separated.

Difference between Servlet and JSP

The following are the differences between Servlets and JSP:

1. Servlets tie-up files (an HTML file for the static content and a Java file for the dynamic content) to handle the static presentation and dynamic business logic independently. Due to this, a change made to any file requires recompilation of the Servlet. On the other hand, JSP allows Java to be embedded directly into an HTML page by using special tags. The HTML content and the Java content can also be placed in separate files. Any change made to the HTML content is automatically compiled and loaded onto the server.
2. Servlet Programming involves extensive coding. Therefore, any change made to the code requires identification of the static code content (for the designer) and dynamic code content (for the developer) to facilitate the incorporation of the changes. On the other hand, a JSP page, by the separate placement of the static and dynamic content, facilitates both the web developers and the web designers to work independently.

Advantages of JSP

The following are the advantages of JSP:

- Extension to Servlet: JSP is an extension to the Servlet. It supports both scripting and dynamic content and allows the developers to create custom tag libraries to satisfy the application-specific needs.
- Compilation: JSP is always compiled before the server processes it. There is no need to recompile and redeploy. If the JSP page is modified, we don't need to recompile and redeploy the project. Whereas the Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.
- JSP is platform-independent.
- There is no need to redeploy JSP.

- The length of the JSP code is less than Servlet.

The JSP request-response cycle

The JSP files are stored on the webserver with an extension of JSP. When the client/browser requests for a particular JSP page, the server, in turn, sends a request to the JSP engine. *Figure 7.1* represents the process of the flow of events that occur after a client request:

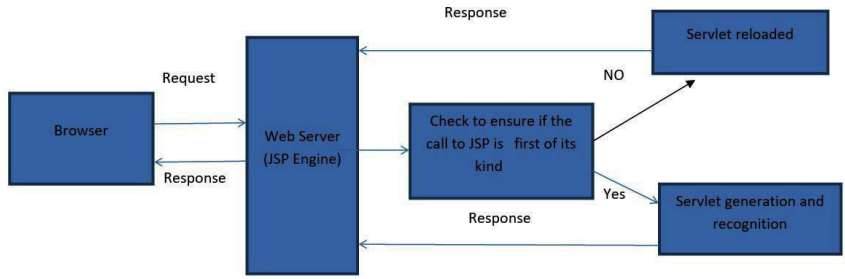


***Figure 7.1**: The flow of events after client request*

The request-response cycle essentially comprises of two phases, namely the translation phase and the request-processing phase. The translation phase is implemented by the JSP engine and involves the generation of a servlet. Internally, these result in the creation of a class file for the JSP page that implements the servlet interface. During the request-processing phase, the response is generated according to the request specifications. The Servlet then sends back a response corresponding to the request received. After the Servlet is loaded for the first time, it remains active and processes all the subsequent requests with responses, saving the time that would otherwise be lost in reloading a servlet at each request.

Lifecycle of JSP

The JSP lifecycle is a process in which the JSP translates into servlets, and then the servlet lifecycle plays its role.

The JSP page lifecycle completes the following steps:

1. **Translation of JSP Page**: The JSP container translates the JSP page into Servlet.
2. **Compilation**: Like any other Java class, the generated Servlet is compiled into byte code, and is ready to be loaded and

executed. The generated servlet has the method **`jspInit()`**, **`_jspService()`**, and **`jspDestroy()`**.

3. **Classloading**: Once JSP is compiled as a servlet class, its lifecycle is similar to Servlet and it gets loaded into memory.
4. **Instantiation**: The object of the generated Servlet is created.
5. **Initialization**: The web container invokes the **`jspInit()`** method. The method is called once. After initialization, the **`ServletConfig`** and **`ServletContext`** objects become accessible to the JSP class. Allow the Servlet to initialize the instance variables when it's loaded Request processing; the container invokes the **`_jspService()`** method.
6. **Destroy**: The container invokes the **`jspDestroy()`** method. The generated Servlet is unloaded from memory and the cleanup instance variable from memory when it's shut down.

NOTE: jspInit(), _jspService(), and jspDestroy() are the lifecycle methods of JSP.

The JSP page lifecycle is illustrated in *Figure 7.2*:

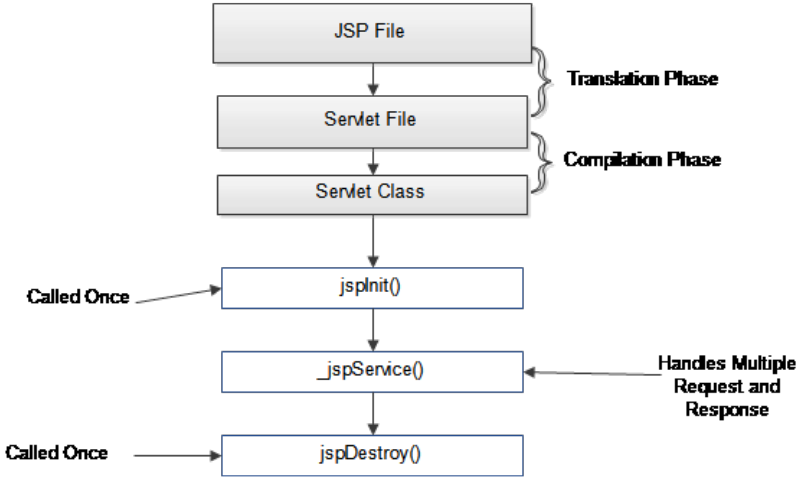


***Figure 7.2**: Lifecycle of JSP Page*

Structure of a JSP Page

A JSP page consists of the regular HTML tags representing the static content and the code enclosed within the special tags representing

the dynamic content. These tags begin with a "<%" and end with a "%>" and contain the scripting and directive elements. The scripting elements consist of java code snippets, while the directives define the specifications for the entire JSP page. The comment line entries that provide additional information about the various sections of the code are enclosed within the "<%--" and "--%>" tags. For example, the following JSP code displays the server time on the browser. It contains the HTML content and the JSP content placed separately within the respective tags.

The following is an example:

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ page language="java" %>
<!DOCTYPE html>
<html>
<head>
<title>JSP Example</title>
</head>
<body>
<h1>This is the code within the JSP tags to display the
server time</h1>
<%--IThis is the JSP content that displays the server time
by using the Date class of the java.utilpackagel--%>
<% java.util.Date now=new java.util.Date(); %>
<H2><%= now %></H2>
</body>
</html>
```

The following is the output of **index.jsp**, as shown in *Figure 7.3*:

***Figure 7.3**: Output of index.jsp*

The directory structure of the JSP Page

The JSP page is placed in the root folder. It should not be placed in the WEB-INF directory as we won't be able to access it directly from the client, as shown in *Figure 7.4*:

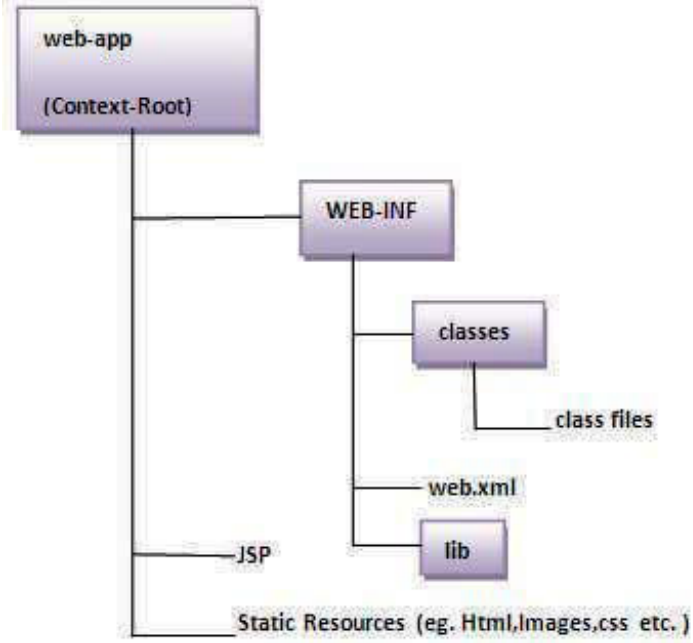


***Figure 7.4**: Directory Structure of the JSP page*

Conclusion

JSP stands for Java Server Pages. It is a server-side technology used for creating dynamic web applications. The features of JSP are segregation of the presentation and business logic.

The changes made to the code consecutively are automatically updated and recompiled once the Servlet is reloaded into the server.

In JSP, we can use many tags, such as action tags, Java Standard Tag Library (JSTL), custom tags, and so on, that reduce the code difference between Servlet and JSP.

During the lifecycle of JSP, the JSP page is converted into Servlet, the Object of Servlet is created, and jspInit(), _jspService(), and jspDestroy() are called to fulfil the request.

A JSP page is placed with a static resource, i.e., in the root folder.

JSP builds various elements and objects that allow the web designer to write dynamic web applications with the help of the embedded JSP elements and objects.

In the next chapter, we will study tags to be embedded in the JSP page.

Questions

1. What is JSP, and why do we need it?
2. What are the JSP lifecycle phases?
3. What are the JSP lifecycle methods?
4. Which of the JSP lifecycle methods can be overridden?
5. What are the advantages of using JSP?
6. What are the advantages of JSP over pure Servlets?
7. What are the advantages of JSP over static HTML?

Chapter 8
Comment Tag and Scripting Element

JSP builds various elements and objects that allow the web designer to write dynamic web applications with the help of embedded JSP elements and objects. Scriptlet tag helps the programmers to write the Java codes in a web application. The code automatically moves to the _jspservice() method while converting JSP to Servlet.

Structure

In this chapter, we will cover the following topics:

- Comment tag
- Scripting Elements
- Implicit Objects

Objectives

After studying this chapter, you will know how to embed a Java code in the web application and learn how the embedded Java code is automatically converted into a servlet.

JSP Elements

The JSP elements help the web developers to write the Java code within the tag. The code within the tag automatically moves to the **`_jspService()`** method while converting JSP into the servlet.

It consists of the following tags of elements.

Comment Tag

The comment tag provides additional information about the various sections of the code that are enclosed within the **`<%--`** and **`--%>`** tags. Comments are used when the programmer wants to hide some text or statements from the web container, as shown as follows:

Syntax: **`<% -- JSP Comments %>`**

Scripting Elements

The Scriptlet tag helps the programmers to write the Java code in a web application. The code automatically moves to the **`_jspservice()`** method while converting JSP to Servlet. The **`_jspservice`** method is invoked for each request. There are three types of scripting elements, which are as follows:

- Scriptlet tag
- Expression tag
- Declaration tag

Scriptlet tag

A JSP scriptlet tag consists of valid code snippets that are enclosed within the **`<%`** and **`%>`** JSP tags. This tag allows the developer to write the Java code within the scriptlet tag. This code moves to the **`_jspService()`** method. When the user requests, the JSP service method is invoked, and the content which is written inside the scriptlet tag gets executed.

Syntax: **`%// java code%>`**.

Example

The following code snippet contains the code to create a request session object and retrieve the variable values using the **`getParameter()`**

method. The following code snippet accepts a name from the user and forwards the input parameters to a JSP page:

index.html:

```
<html>
<head>
<title>JSP Elements</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
</head>
<body>
<h3> Scriptlet Example</h3>
<form action="test.jsp">
Your name  :
<input type = "text" name="user_name" />
<input type = "submit" value="submit">
</form>
</body>
</html>
```

Test.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<h1>Welcome to JSP Page</h1>
<%
            String name=(String)request.getParameter("user_name");
            out.println(name);
        %>
</body></html>
```

The following is the output of `index.html`, as shown in *Figure 8.1*:

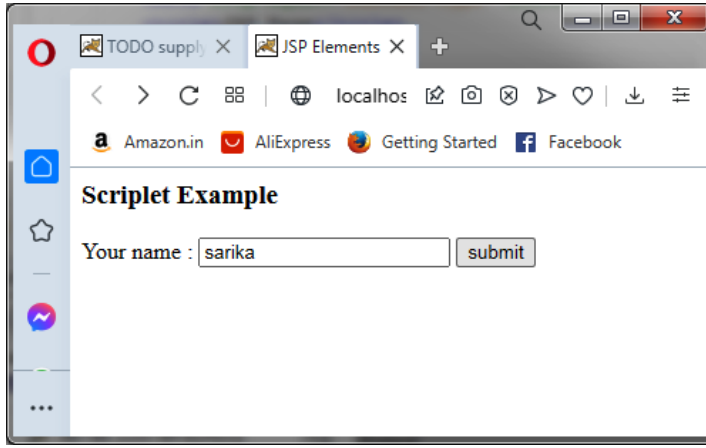


***Figure 8.1**: Output of index.html*

After clicking on **`submit`**, we get the following output, as shown in *Figure 8.2*:

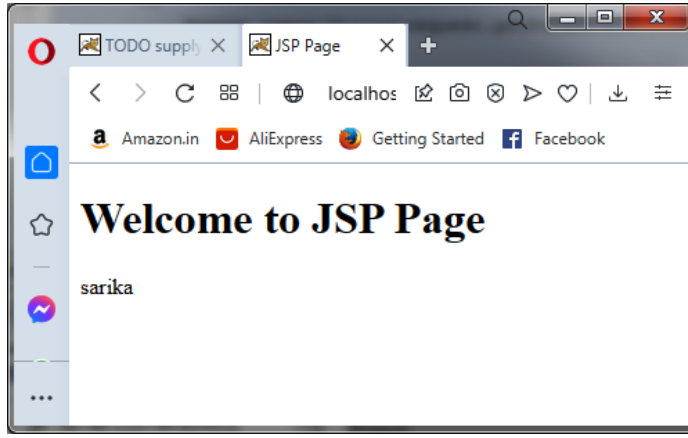


***Figure 8.2**: Output of test.jsp page*

Expression Tag

It is used to insert the values directly into the output. You need not write the **`out.println()`** statement to write the data.

Syntax:`<%= msg%>`

The following is an example:

```
html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>

<%= "value to be printed" %>
</body>
</html>
```

The following is the output:

```
Value to be printed
```

Declaration tag

It is used to declare the fields and methods. The code is written with the help of this tag placed outside the service method and gets memory one time, not at every request. The scriptlet tag can declare only variables, and all the variables are local variables to the service method, whereas, in the declaration tag, both the variables and methods can be declared outside the service method.

Syntax:

```
<%! Variable or method declaration %>
```

NOTE: The JSP scriptlet tag can only declare the variables, and not the methods, whereas the JSP declaration tag can declare both the variables and the methods.

Example:

In the following example, one variable data and method square is declared using a declaration tag:

```
%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<html>
<body>

<%! int data=120;
int square(int n) {return n*n;}
%>
<%= "Value of the variable is:"+data %><br>
<%= "square of   4 is:"+square(4) %>
</body>
</html>
</body>
</html>
```

The following is the output:

```
Value of the variable is:120
square of 4 is:16
```

Implicit Objects

The objects in JSP can be created implicitly by using the directives, explicitly by using the standard actions, or directly by declaring them within the scriptlets. JSP implicit objects are certain predefined variables that can be included in the JSP expressions and scriptlets. The implicit objects of JSP are implemented from the servlet classes and interfaces.

There are nine implicit objects created by the web container and are available in JSP; they are as follows:

1. **`application`**: The application object defines a web application. Usually, it is the application in the current web context or instance of **`ServletContext`**. The instance of **`ServletContext`** is created only once by the web container when the project is deployed on the server. It is used to set or remove the attribute from the application scope.

The application object can be used to get the initialization parameter from the configuration file (**`web.xml`**). All the JSP pages can use this initialization parameter.

Example: The password is stored in context-param and can be fetched with the help of the application object. Refer to the following example code:

```
<!DOCTYPE html>
<html>
<head>
<title>Implicit Object</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
</head>
<body>
<form method="post" action="submit.jsp" >
<input type="text" name="uname">
<input type="submit" value="Submit"><br/>
</form>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
<%
    out.print("Welcome "+request.
getParameter("uname"));
    String pwd=application.getInitParameter("pwd");
    out.print("Your Password is="+pwd);
  %>
</body></html>
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
<servlet>
<servlet-name>NewServlet</servlet-name>
<jsp-file>/submit.jsp</jsp-file>
</servlet>
<servlet-mapping>
<servlet-name>NewServlet</servlet-name>
<url-pattern>/submit</url-pattern>
</servlet-mapping>
<context-param>
<param-name>pwd</param-name>
<param-value>SAILBOAT</param-value>
</context-param>
<session-config>
<session-timeout>
            30
</session-timeout>
</session-config>
</web-app>
```

The following is the **output** for index.html, as shown in *Figure 8.3*:

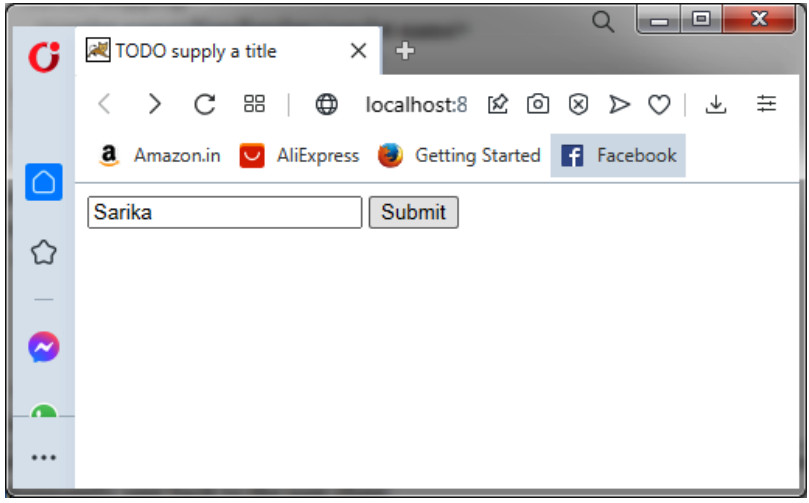


***Figure 8.3**: Output of the preceding program (index.html)*

The following is the **output** of `submit.jsp`, as shown in *Figure 8.4*:

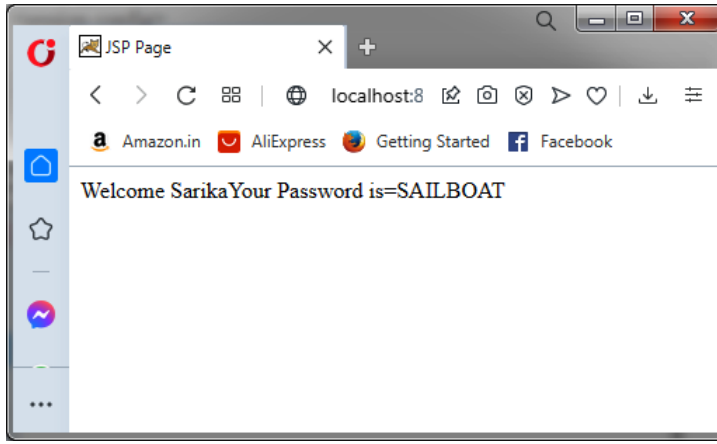


***Figure 8.4**: Output of the preceding program (submit.jsp)*

2. **`out`**: This represents a reference to a JspWriter used to write to the output stream and subsequently sent back to the user client.

   The following is an example:

   ```
   <html>
   <body>
   <% out.print("Today is" + java.util.Calender.
   getInstance(),getTime());%>
   </body>
   ```

   The following is the output:

   ```
   Today is: Sun Nov 14:19:08 IST 2021
   ```

3. **`page`**: This represents the current instance of the JSP page that, in turn, is used to refer to the current instance of the generated servlet. It is written as follows:

   ```
   Object page =this;
   ```

   For using this **`Object`**, it must be cast to the servlet type. Refer to the following example:

   ```
   <% (HttpServlet)page.log("message"); %>
   ```

   Since, it is of type **`Object`**, it is less used because you can use this object directly in JSP.

Refer to the following example:

```
<% this.log("message"); %>
```

4. **request**: This represents a request object of **HttpServletRequest**. It is used to retrieve the data submitted along with a request. An example of the request object is **request.getParameter()**, which is shown as follows:

```
index.jsp
<html>
  <body>
    This is my JSP page. <br>
    <form method="post" action="next1.jsp" >
            First Name: <input type="text"
name="uname">
             Last Name: <input type="text"
name="lname">
             <input type="submit"
value="Submit"><br/>
      </form>
      </body>
</html>
```

The following is the output of the preceding program, as shown in *Figure 8.5:*

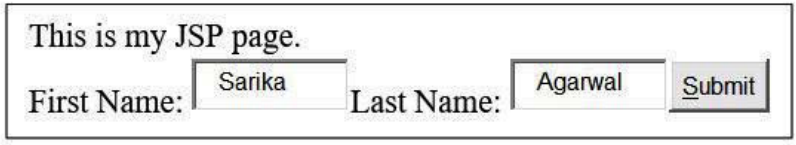


Figure 8.5: *Output of the index.jsp*

After clicking on the **Submit** button, the following code (**next1.jsp**) will run:

next1.jsp

```
<html>
<body>
<%
String str1=request.getParameter("uname");
String str2=request.getParameter("lname");
```

```
out.println("First name is "+str1);
out.println("<br>Last Name is"+str2);
 %>
</body>
</html>
```

The following is the output:

```
First name is sarika
Last  Name  is Agarwal
```

Some methods that can be used with the **`HttpServletRequest`** interface are listed as follows:

- **`getCookies()`**: It is used to pass the cookie information along with the request. Cookies are textual data that are sent from the server to a browser. The data thus sent is retrievable in subsequent transactions.
- **`getSession()`**: It is used to get the current session associated with the request object. If a session is not associated with the request, it creates a new session.
- **`getMethod()`**: It is used to return a String specifying the presence of HTTP get, post, or put methods to the request.
- **`getQueryString()`**: It is used to return the query passed along with the **`request(GET)`**.

5. **`response`**: This represents a response object of **`HttpServletResponse`** that is used to write an HTML output onto the browser using methods such as **`response.getWriter()`**.

   Some methods that can be used with the **`HttpServletResponse`** interface are listed as follows:

   - **`addCookie()`**: It is used to add a cookie along with the response. The maximum number of cookies accepted by the browser is 20.
   - **`sendRedirect()`**: It is used to specify a new URL to the browser. The invocation of the method terminates the prior response and redirects the browser contents to a new URL.

6. **session**: This represents a session object of HttpSession that is used to store the object between client requests.

   Example: The following codes use the request implicit object to access the values of the user input; set the parameter in session and access the same:

   index.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
</head>
<body>
<form method="post" action="next.jsp" >
             First Name: <input type="text"
name="uname">
             Last Name: <input type="text"
name="lname">
<input type="submit" value="Submit"><br/>
</form>
</body>
</html>
```

   Bottom of Form:

   After clicking on the **Submit** button, the following code will run:

```
<!DOCTYPE html>
<html>
<%
      String str1=request.getParameter("uname");
String str2=request.getParameter("lname");
          HttpSession sess=request.getSession(true) ;
        sess.setAttribute("FName", str1);
      sess.setAttribute("LName", str2);
  %>
```

```
<body>
<%        out.println("First name is "+session.
getAttribute("FName"));
            out.println("<br>Last Name is"+session.
getAttribute("LName"));


            %>
</body></html>
```

The following is the output of `index.html`, as shown in *Figure 8.6*:

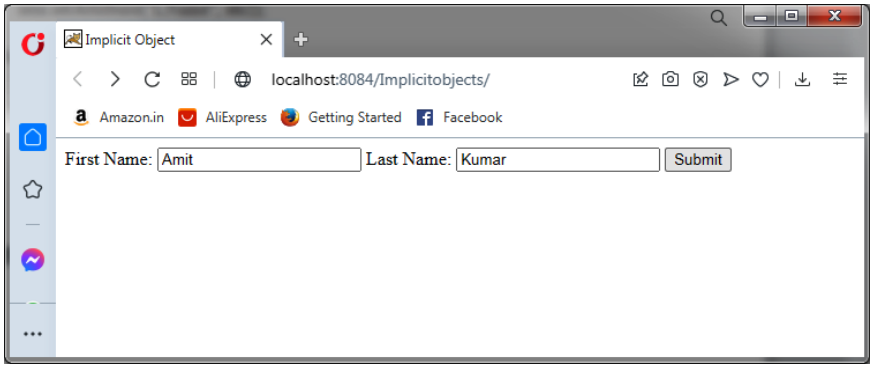


***Figure 8.6**: Output of the preceding program (index.html)*

The following is the output of `next.jsp`, as shown in *Figure 8.7*:

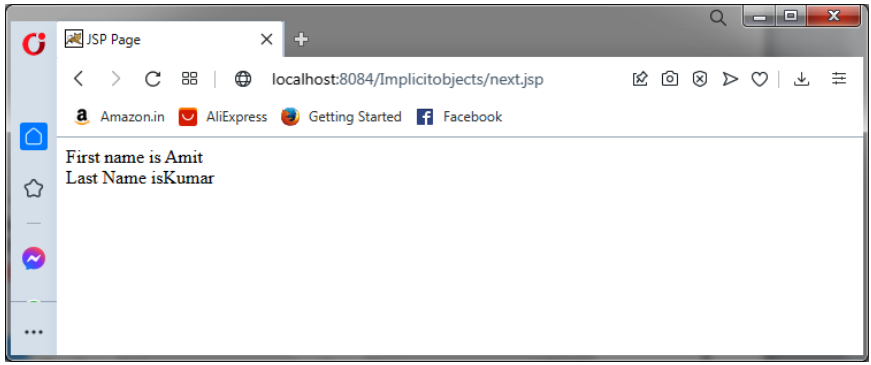


***Figure 8.7**: Output of next.jsp (Retrieval from Session Object)*

7. **Config**: It is an implicit object of the **ServletConfig** class. The web container creates this object. It is used to set or get the initialization parameter for a JSP page.
8. **exception**: It is an implicit object of the **java.lang.Throwable** class. It is used to print the **exception**.
9. **pageContext**: It is an implicit object of the **PageContext** class. It is used to set, get, and remove the attribute from the page, request, session and application scope. By default, JSP has a Page scope.

Conclusion

In this chapter, we learned that a JSP page consists of HTML and JSP tags. The JSP tags include comments, scriptlets, expressions, and actions tags.

The JSP comment tag provides additional information about the various sections of the code and are enclosed within the **<%--** and the **--%>** tags.

Scriptlet tag consists of valid code snippets placed within the **<%** and the **%>** tags.

JSP expressions are used to insert values into the output directly. There are nine implicit objects in JSP.

In the next chapter, we will learn how to provide global information about a particular JSP page and how to give information to a Web Container while converting a JSP page to a corresponding servlet.

Questions

1. What are JSP elements?
2. What is the role of scriptlet tag?
3. What is the difference between a JSP scriptlet tag and a JSP declaration tag?
4. How do we use comments within a JSP page?
5. How many implicit objects are there in JSP?

6. What is the use of implicit objects in JSP?

Select the Correct Option

1. **Which of the scripting of JSP does not put content into the service method of the converted servlet?**

   A. Scriptlets

   B. Declarations

   C. Expressions

   D. None of the above

Answer: C

2. **Which of the following tags is used to execute the Java source code in JSP?**

   A. Scriptlets

   B. Declarations

   C. Expressions

   D. None of the above

Answer: A

3. **Which of the following is the correct order of phases in the JSP life cycle?**

   A. Compilation, Initialization, Execution, Cleanup

   B. Cleanup, Compilation, Initialization, Execution

   C. Initialization, Cleanup, Compilation, Execution

   D. Initialization, Compilation, Cleanup, Execution

Answer: A

4. **J2EE includes which of the following enterprise-specific APIs?**

   A. Java Message Service (JMS)

   B. Enterprise JavaBeans (EJB)

   C. JavaServer Pages (JSP)

D. All of the above

Answer : All of the above

5. A JSP page is transformed into_____.

A. Java Class

B. Java Servlet

C. Java Bean

Answer : Java Servlet

6. JSP pages provide a means to create a dynamic web page using HTML and Java programming language.

A. True

B. False

Answer: True

Chapter 9
JSP Directives

A directive element in a JSP page provides global information about a particular page. It gives information to a Web Container while converting a JSP page into a corresponding servlet.

Structure

In this chapter, we will cover the following topics:

- Page Directives
- Exception Handling via error page
- Include Directive
- Taglib Directive

Objective

In this chapter, we will learn about the JSP Directives that are applied on a JSP Page. JSP Directives are used to define the variables and methods on a page.

JSP Directives are used to specify general information about a particular page.

Types Of Directives

The following are the types of directives:

- The page
- include
- taglib

JSP directives

JSP directives are used to give messages to a web container about how to handle the client requests while converting a JSP page into servlet code.

The syntax for adding a directive in a JSP page is as follows:

```
<%@ directive type (attribute= “attribute value”) %>
```

The page directive

The page directive defines the attributes that notify the servlet engine about the general settings of a JSP page. The page directive applies to the entire JSP page. The page directive has many attributes separated by commas as the key-value pairs. The attributes that can be specified for a page directive are listed in *Table 9.1*:

S.No	Attribute	Used to Specify
1	`contentType=”MIME type”`	This is the MIME type (Multipurpose Internal Mail Extension type) of the response. The default value for the attribute is text/HTML.
2	`extends “packagename. class”`	This is the name of the parent class that the generated servlet will extend from.
3	`errorPage= “url”`	This is the URL of the error page that will be used to handle exceptions.

4	**isErrorPage= "False"**	If a particular JSP page can be used as an error page for another JSP page. The default value for this attribute is `false`.
5	**import= "package list"**	These are the names of the packages available for the particular JSP page.
6	**Language= "scripting language."**	This is the scripting language to be used when compiling the JSP page. The language currently available is Java.
7	**Session= "true"**	It is used to specify the availability of the session data for the particular JSP page. The default value for this attribute is `true`.
8	**Info**	It sets the information of the JSP page, which can be retrieved by using the `getServletInfo()` method of the Servlet interface.
9	**Buffer**	It sets the buffer size in kilobytes to handle the output generated by the JSP page. The default size of the buffer is 8KB.
10	**isELIgnored**	We can ignore the **Expression Language** (**EL**) in JSP by the ELIgnored attribute. By default, its value is `false`, that is, Expression Language is enabled by default.
11	**isThreadSafe**	It serializes the JSP request.
12	**Autoflush**	It specifies whether the buffered output should be flushed automatically when the buffer is filled or whether an exception should be raised to indicate the buffer overflow.
13	**PageEncoding**	The page encoding is the character encoding in which the file is encoded.

***Table 9.1**: Implicit Objects*

Implicit Objects

contentType

The **contentType** attribute defines the MIME type of the response. MIME is a standard that specifies how the messages need to be formatted when exchanged between the different systems. A MIME message can consume different types of data like text, images, audio, and video.

The default value is "text/html.

The syntax is as follows:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

extends

The **extends** attribute extends the parent class in the JSP. The name of the parent class that the generated servlet will extend from must be specified.

The syntax is as follows:

```
<%@ page extends = "package.ClassName" %>
```

errorPage

It is used to handle exceptions. The **errorPage** attribute is used to define the error page; if the exception occurs on the current page, it will be redirected to the error page.

The syntax is as follows:

```
<%@ page errorPage= “next.jsp” %>
```

isErrorPage

The **isErrorPage** attribute is used to declare that the current page is the error page.

The syntax is as follows:

```
<%@ page isErrorPage=true  %>
```

Example: In this example, **`index.jsp`** has the Exception as '**`s`**' variable is pointed to the null type. The exception is handled in **`error.jsp`** page.

Refer to the following example code:

`Index.jsp`:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<h1> Welcome to the is error page example %>
<%@ page errorPage="error.jsp" %>
<%
        String s=null;
char j=s.charAt(0);
out.println(j);
        %>
</body>
</html>
```

`error.jsp`:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<h1> Welcome to the  error page  </h1><br>
```

```
<%@ page isErrorPage="true" %>
<%= exception %>

</body>
</html>
```

The following is the output, as shown in *Figure 9.1:*

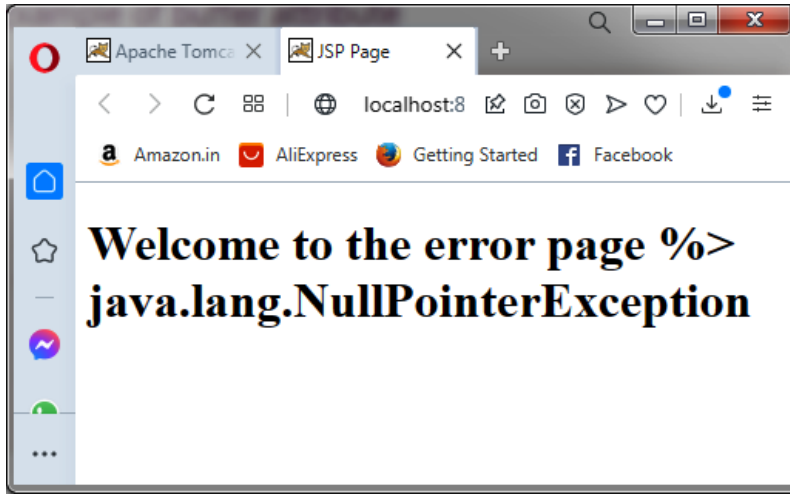


***Figure 9.1**: Output of the preceding program*

import= "package list"

This attribute imports interface and class.

The syntax is as follows:

```
<%@ page import= "java.util.Date" %>
```

Example: This example **`import java.util.Date`** package and display the current date:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
    <title>JSP Page</title>
```

```
   </head>
   <body>
      <h1> import attribute</h1>
      <%@ page import="java.util.Date" %>
      Today is: <%= new Date() %>
   </body>
</html>
```

The following is the output, as shown in *Figure 9.2:*

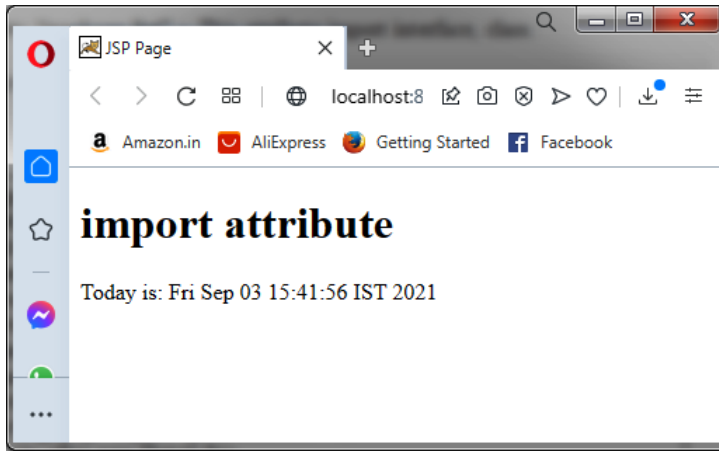


***Figure 9.2**: Output of the preceding program*

language= "scripting language."

It specifies the scripting language to be used when compiling the JSP page. The language currently available is Java.

The syntax is as follows:

```
<%@ page language= "java" %>
```

Session=true/false

It is used to specify the availability of the session data for the particular JSP page. It indicates whether the JSP page uses the HTTP sessions or not. The default value for this attribute is **`true`**. If the value is **`true`**, the JSP page has access to a built-in session object. We can use the methods like **`session.getCreationTime()`** or **`session.getLastAccessTime()`**.

The syntax is as follows:

```
<%@ page session= “true” %>
```

The following is the example:

```
  <%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    session="false"%>
```

info= “servlet information”

It specifies the servlet description. It defines the servlet information in the form of a string and can be accessed with the **`getServletInfo()`** method.

The syntax is as follows:

```
<%@ page info="servlet information" %>
```

The following is the example:

```
<%@ page info="page directive info explanation”%>
```

Buffer

Client response should be stored. By default, we have an 8 KB buffer size to store the same. If we want to change the size, we can use this attribute.

The syntax is as follows:

```
<%@ page buffer="value" %>
```

The following is an example:

```
  <%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    buffer="16KB"%>
```

isELIgnored= “true/false”

This attribute is used to disable the **Expression Language** (**EL**) in JSP by the **`isELIgnored`** attribute introduced in JSP 2.0. It is enabled by default. It means that its value is set to **`false`** by default.

The syntax is as follows:

```
<%@ page is ELIgnored= “true” %>”
```

NOTE: Expression Language will be ignored in this syntax.

isThreadSafe

Both JSP and Servlets are multithreaded by default, that is, the value **`isThreadSafe`** is **`true`**. If you want to serialize the client request, you have to make it **`false`**. Web container will not accept any further client requests until the previous serialized client request is fulfilled. In such a case, the web container generates a servlet that implements the **`SingleThreadModel`** interface.

The syntax is as follows:

```
<%@ page isThreadSafe="false" %>
```

autoFlush

This attribute is used to specify whether the buffered output should be flushed automatically or not. By default, the value is **`true`**, that is, it flushes automatically when the buffer is full. If the value is set to **`false`**, the buffer will not be flushed automatically, and if it is full, it will throw an exception.

The syntax is as follows:

```
<% @ page autoFlush="true/false" %>
```

NOTE: If the user sets autoflush = "none", the output will not be buffered.

The include directive

The **`include`** directive is used to specify the names of the files (in the form of their relative URLs) to be inserted during the compilation of the JSP page. The contents of the files so included become part of the JSP page. The include directive can also be used to insert a part of the code that is common to multiple pages, to avoid using a bean for each of the code instances separately. The included file can be an HTML file, a text file, or a code written in the Java programming language.

For example, the code line to include a header **`file(Instituteoflearning.html)`** for a JSP page containing the name and logo of the institute can be written as follows:

```
<%@ include="Instituteoflearning.html" %>
```

The following are the advantages:

- Reusability of code, useful for including copyright information, scripting language files, or anything you might want to reuse in the other applications.
- Inserts the contents of another file in the main JSP file, where the directive is located.

The following is an example:

```
html>
<body>
  <%@ include file=”instituteoflearning.html" %>
  <h6> Today Date and Time  is </h6>
<%= java.util.Calendar.getInstance().getTime() %>
  </body>
</html>
```

The Taglib directive

The JSP **taglib** directive is used to define a tag library that defines many tags. A tag library consists of a collection of functionality-related user-defined XML tags called **custom tags**. Custom tags are described in further chapters. We use the **Tag Library Descriptor** (**TLD**) file to define the tags.

The syntax is as follows:

```
<%@ taglib uri="urioftheta glibrary"
prefix="prefixoftaglibrary" %>
```

Conclusion

The JSP directives are used to specify the general information about a particular page. Page directive defines the attributes that apply to an entire JSP page such as extend to import class, isThreadSafe to define if the JSP is thread-safe, error page andisErrorPage to handle the exception, and session to check if the session object is available or not.

The include directive inserts the contents of another file in the main JSP file, where the directives are located. The Taglib directive is used to define a tag library that defines many tags.

In the next chapter, we will learn about JSP actions such as useBean, getProperty, setProperty, and forward to perform tasks such as inserting files, reusing beans, forwarding a user to another page, and instantiating objects. We will also learn about a custom tag library that provides a mechanism by which the programmer can encapsulate recurring code or tasks and reuse them in multiple applications.

Questions

1. **JSP handles the runtime errors using _____ attribute in page directive.**

Answer: By errorPage and isErrorPage attributes

2. **What are the major attributes of page directives?**

Answer: buffer, ContentType, autoFlush, errorPage, is ErrorPage, extends, isThreadSafe, language, Session, import, info, isElIgnored

3. **The JSP Page is extensible.**

   A. True

   B. False

Answer: True

4. **What is the page directive used for?**

   A. To instruct the translator about assigning values within the JSP page it is contained within.

   B. To instruct the translator about the characteristics of the JSP page it is contained within.

   C. To instruct the translator about the files being used by this JSP page.

   D. To instruct the translator about the modules being used by this JSP page.

Answer: A

5. Where do we place a page directive within a JSP page?

A. At the start

B. In the middle

C. At the end

D. Anywhere

Answer : D

6. How many page directives are allowed within a JSP page?

A. 1

B. 2

C. Multiple

Answer: C

7. How many page directive attributes are there?

Answer: 3

8. Using the page directive, what type of files can we include in a JSP page?

A. HTML

B. JSP

C. Neither

D. Both

Answer: D

9. What are pages included using the page directive commonly referred to as?

A. Components

B. Fragments

C. Modules

D. Parcels

Answer: A

Chapter 10
JSP Action Element and Custom Tags

Introduction

JSP actions are used to perform tasks such as inserting files, reusing beans, forwarding a user to another page, and instantiating objects. The custom tags provide a mechanism that the programmer can use to encapsulate the complex recurring code or tasks.

Structure

In this chapter, we will cover the following topics:

- JSP Action Elements
- Custom Tags
- Model view controller (MVC) Architecture

Objectives

After studying this chapter, you will learn about the use of action tags, and also learn how to make your own tags, use them in your

application, and see how the Expression language simplifies the accessibility of the data stored in the Java Bean component and other objects like request, session, and application. In this chapter, you will also learn about the MVC architecture that separates the business logic from presentation logic.

JSP Action Tags

JSP provides standard tags used to insert, remove, and reuse beans, forward the users to another page and instantiate the objects.

The syntax of the JSP action tag is as follows:

```
<jsp:action_name attribute="attribute_value"/>
```

The various JSP actions are described in *Table 10.1*, as follows:

S.No	JSP action	Usage	Attribute	Description
1	`<jsp:use-Bean>`	Used to find and load an existing bean	`id`	Uniquely identifies the instances of the bean
			`class`	Specifies the class from which the bean objects are to be implemented
			`scope`	Specifies the bean's life in terms of a page, session, or application
			`beanName`	Specifies a referential name for the bean.
2	`<jsp:get-Property>`	Used to retrieve the property to the specified bean and direct it as the output. The retrieved value is converted to a string value before sending it as an output.	`name`	Specifies a name for the bean
			`property`	Specifies the property from which the values are to be retrieved

3	**<jsp:set-Property>**	Used to set the property of the specified bean; to set the value of the bean property, either an explicit value is specified, or the value is obtained from a request parameter. Corresponding to the specified request property value, the set method in the bean is called with the matching value.	**name**	Specifies the name of the bean
			property	Specifies the property for which the values are to be set. If set to "*", it specifies that the set methods for all the specified values should be called.
			value	Specifies an explicit value for the bean property
			param	Specifies the name of the request parameter to be used to set the value of the bean property
4	**<jsp:forward>**	Used to forward a request to a different page	**page**	Specifies the relative URL of the target page
5	**<jsp:include>**	Used to insert a file into a particular JSP page. The file inclusion takes place at the time of the request of the JSP page.	**page**	Specifies the relative URL of the page to be included
			flush	Specify if the buffer is to be flushed. A mandatory Boolean attribute must be included when declaring the included action.

6	`<jsp: param>`	Used as a sub-attribute with `jsp:include` and `jsp:forward` to pass additional request parameters	`name`	Specifies the name of the reference parameter
			`value`	Specifies the value for the reference parameter

***Table 10.1**: The various JSP Action tags*

There are many JSP action tags, and each of them has its uses and characteristics.

jsp:useBean action tag

It is used to create a reference to specify the inclusion of a predefined bean component in the JSP page. If the bean object of the Bean class is already created, it doesn't create the bean depending on the scope. But if an object of the bean is not created, it instantiates the bean.

Syntax:

```
<jsp:useBean id= "instanceName" scope= "page | request |
session | application"   class= "packageName.className"
type= "packageName.className"  beanName="packageName.
className | <%= expression >" >  </jsp:useBean>
```

Example: **`<jsp:useBean id=”savAcc” scope= “application” class= “savAccount” />`**

The preceding code line specifies the inclusion of a bean referred to by the name **`savAcc`** of the class **`savAccount`**. The class is the Java class that defines the bean. A JavaBean class should follow the following conventions:

- It should have a constructor with no argument.
- It should be serializable.
- It should provide methods to set and get the values of the properties, known as the getter and setter methods.

Once instantiated, a bean can be reused using the same scope and ID.

The following are the attributes of the **`useBean`** tag:

- **id**: This attribute uniquely identifies the instance of the bean.
- **Class**: This attribute specifies the class from which the bean objects are to be implemented.
- **Scope**: This attribute specifies the life of the object and takes the following values, as described in *Table 10.2*:

Scope	Life of the object
page	The availability of the object corresponds to the response of the particular page to the current year's request. The object is created with the initiation of a user request and destroyed on its completion.
session	The availability of the object corresponds to the existence of a particular session.
application	The availability of the object exists throughout the application.
request	The availability of the object corresponds to the existence of the `HttpServletRequest` object.

***Table 10.2**: Value of scope attributes*

- **beanName**: This attribute specifies a referential name for the bean.
- **type**: This attribute provides the bean a data type if the bean already exists in the scope. It is mainly used with the class or **beanName** attributes. If you use it without class or **beanName**, no bean is instantiated.

  Example: In the following example, a bean has a method square which is fetched with the help of the **useBean** tag:

  MyBean class:

```
package IOL;
public class MyBean
{
public int square(int n)
{return n*n;}
}
```

index.jsp:

```
<html>
 <body>
    <jsp:useBean id="obj" class="IOL.MyBean"/>
    <%
       int m=obj.square(5);
       out.print("squareof 5 is "+m);
       %>
</body>
</html>
```

The following is the output, as shown in *Figure 10.1:*

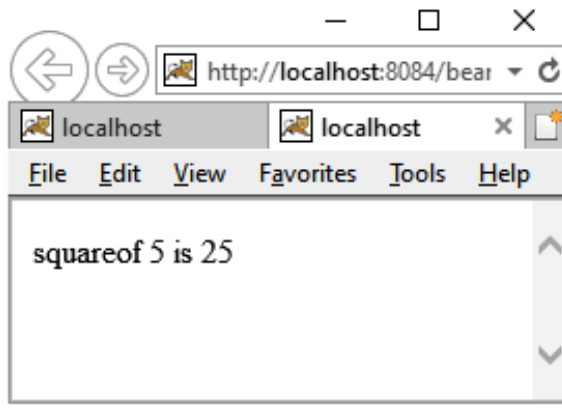


***Figure 10.1**: Output of index.jsp*

jsp:setProperty and jsp:getProperty action tags

These tags are used to set and get the property value in the bean class. It is a reusable component that represents the data.

The syntax of the `setProperty` action tag is as follows:

```
<jsp:setProperty name="instanceOfBean" property= "*"
|  property="propertyName" param="parameterName"  |
property="propertyName" value="{ string | <%= expression
%>
/>
```

Example:

```
<jsp:setProperty name=””savAcc” property=”*” />
```

The preceding code line sets all the values of the upcoming request in the bean class.

The syntax of the `jsp:getProperty` action tag is as follows:

```
<jsp:getProperty name="instanceOfBean" property=
propertyName/>
```

Example:

```
<jsp:setProperty name =”savAcc”  name=”accountNo” />
```

The **`jsp:getProperty`** action tag returns the value of the property **`accountNo`**.

The following are the attributes of the **`setProperty`** tag:

- **`name`**: Specifies a name for the bean.
- **`property`**: Specifies the property for which values are to be set. If set to “*”, it specifies that the set methods for all the specified values should be called.
- **`Value`**: Specifies an explicit value for the bean property.
- **`Param`**: Specifies the name of the request parameter to be used to set the value of the bean property.

The following are the attributes of the **`getProperty`** tag:

- **`Name`**: Specifies the name of the bean.
- **`Property`**: Specifies the property that needs to be retrieved.

Example: In the following example, we set and get the user name and password from the bean class:

`index.jsp`:

```
<html>
<body>
    <form action="next.jsp" method="post">
        User Name:<input type="text" name="userName"><br>
        Password:<input type="password"
name="password"><br>
     <input type="submit" />
```

```
     </form>
</body>
</html>
```

Bean class:

```
package IOL;
public class NewClass implements java.io.Serializable
{
    public String userName, password;

    public String getUserName()
       {
        return userName;
       }

    public void setUserName(String userName)
       {
this.userName = userName;
       }

    public String getPassword()
       {
        return password;
       }
    public void setPassword(String password)
       {
this.password = password;
        }
}
```

next.jsp:

```
<html>
<body>
   <jsp:useBean id="myBean" class="IOL.NewClass"></
   jsp:useBean>
   <jsp:setProperty property="*" name="myBean"/>
     Record:<br>
```

```
 User name=<jsp:getProperty property="userName"
 name="myBean"/><br>
 password=<jsp:getProperty property="password"
 name="myBean"/><br>
<br>
</body>
</html>
```

The following is the output of `index.jsp`, as shown in *Figure 10.2:*

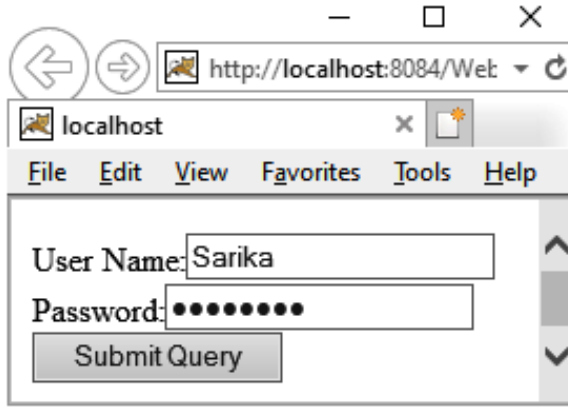


***Figure 10.2**: Output of index.jsp*

The following is the output of `next.jsp`, as shown in *Figure 10.3:*

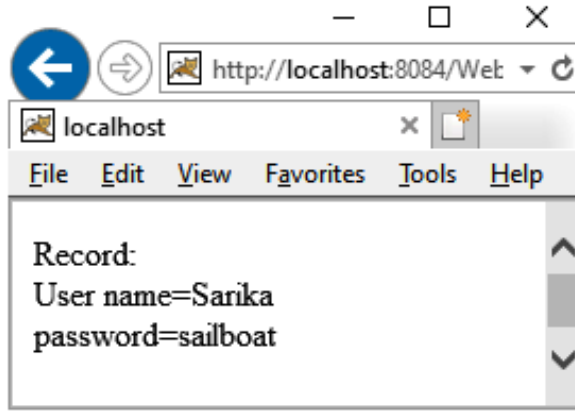


***Figure 10.3**: Output of next.jsp*

Jsp:forward action tag

This tag is used to forward the request to another page.

Syntax:

```
<jsp:forward page="relativeURL | <%= expression %>">
<jsp:param name="parametername" value="parametervalue |
<%=expression%>" /></jsp:forward>
```

page is the attribute of the forward tag.

Example:

Index.html:

```
<html>
<body>
<form action="Next.jsp" method="post">
        User Name:<input type="text"
name="userName"><br>
        Password:<input type="password"
name="password"><br>
<input type="submit" />
</form>
</body>
</html>
```

Next.jsp:

```
<html>
   <body>
       <jsp:forward page="next2.jsp">
       <jsp:param name="date" value=" 17 January 2022"/>
       </jsp:forward>
   </body>
</html>
```

Next2.jsp:

```
<html>
    <body>
<h1> Record <br>
```

```
            User Name=<%= request.getParameter("userName")
%><br>
            password = <%= request.getParameter("password")
%><br>
            Date=<%= request.getParameter("date") %></h1>
      </body>
</html>
```

The following is the output of `index.html`, as shown in *Figure 10.4:*

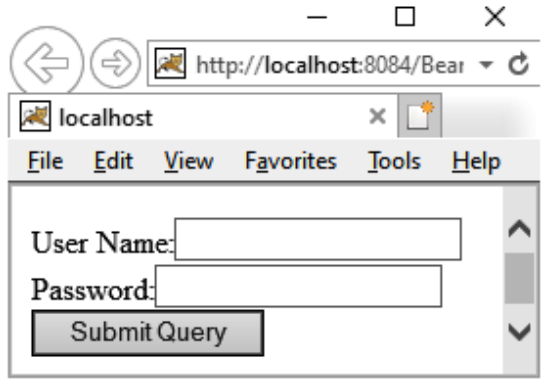


***Figure 10.4**: Output of index.html*

The following is the output of `next2.jsp`, as shown in *Figure 10.5:*

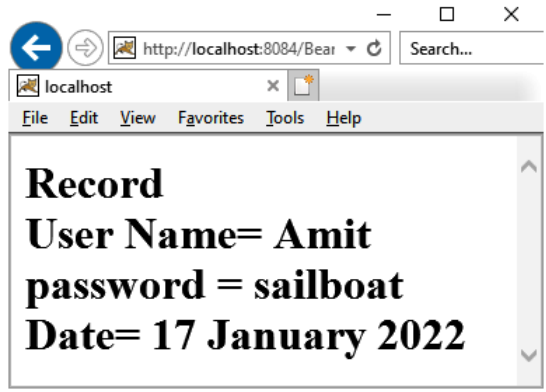


***Figure 10.5**: Output of Next2.jsp*

jsp:include action tag

It is used to insert the content of the file into a particular JSP file. The file may be JSP, HTML, or servlet. The file inclusion takes place at the time of the request of the JSP page.

The following are the attributes:

- **page**: It specifies the relative URL of the page to be included.
- **flush**: It specifies if the buffer is to be flushed. A mandatory Boolean attribute has to be included when declaring the include action.

Syntax:

```
<jsp:include page="relativeURL | <%= expression %>">
<jsp:param name="parametername" value="parametervalue |
<%=expression%>" />  </jsp:include>
```

Example:

printdate.jsp

```
<% out.print("Today is:"+java.util.Calendar.getInstance().
getTime()); %>
```

Index.jsp

```
<html>
      <body>
            <h2>this is index page</h2>
            <jsp:include page="printdate.jsp" />
            <h2>end section of index page</h2>
      </body>
</html>
```

Example:

In the following example, the user enters the account number and password on a JSP page. If the account number and password exist in the database, then the user is validated, otherwise an error message will be generated, as shown as follows:

Index.jsp

```
<html>
  <body>
```

```
<form method="post" action="usebean.jsp">
   <table border="0" cellspacing="1" cellpadding="5">
   <tr>
        <td width="100"> </td>
        <td align="right">
        <h1>
        <font color="red">
         Welcome to Famous bank</font>
        </h1>
        </td>
   </tr>
   <tr>
    <td width="100" align="right"><b>
      <font color="blue">User ID</font>
        </b>
      </td>
      <td align="left">
      <input type="text" name="userId" size="30">
      </td>
   </tr>
   <tr>
      <td width="100" align="right"><b>
           <font color="blue">Password</font>
           </b>
      </td>
      <td align="left">
        <input type="password" name="pwd" size="30">
      </td>
        </tr>
        <tr>
           <td width="100"> </td>
           <td align="right"></td>
           </tr>
           <tr>
           <td width="100" >  </td>
```

```
                <td align="left">
                <input type="submit" value="submit" />
                </td>
                <tr>
  </table>
  </form>
  </body>
  </html>
```

Usebean.jsp

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<jsp:useBean id="BA" scope="application" class="IOL.
MyBank"/>
<jsp:setProperty name="BA" property="*"/>

<%
            String userid=BA.getUserId();
           String pwd=BA.getPwd();
boolean valid=BA.validate();
            if(valid == true)
out.println("The user is valid");
            else
out.println("The user is not valid");
  %>
</body>
</html>
```

MyBank.java

```
package IOL;
import java.sql.*;
```

```
public class MyBank
{
    private String userId,pwd;
    public Connection con;
    public String getUserId()
   {
        return userId;
    }
    public void setUserId(String userId)
      {
this.userId = userId;
      }
    public String getPwd()
{
        return pwd;
      }
    public void setPwd(String pwd)
      {
        this.pwd = pwd;
       }
    public MyBank()throws
ClassNotFoundException,SQLException
{
Class.forName("com.mysql.cj.jdbc.Driver");
con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/MyBank","root","sailboat");
    }
  public boolean validate()
   {
String ppwd="";
boolean valid=false;
       try{
userId=getUserId();
pwd=getPwd();
             String  strQuery="select * from login where
```

```
userid='"+userId+"'";
            Statement stat=con.createStatement();
ResultSet result=stat.executeQuery(strQuery);
System.out.println(result.next());
            while(result.next())
             {
ppwd=result.getString(1);
System.out.println(ppwd);
            }
ppwd.trim();
pwd.trim();
             if(ppwd.equals(pwd))
                   valid=true;
       }catch(Exception e){}
            return valid;
   }
}
```

The following is the output of `index.jsp`, as shown in *Figure 10.6:*

***Figure 10.6**: Output of index.jsp*

The following is the output of `usebean.jsp`, as shown in *Figure 10.7:*

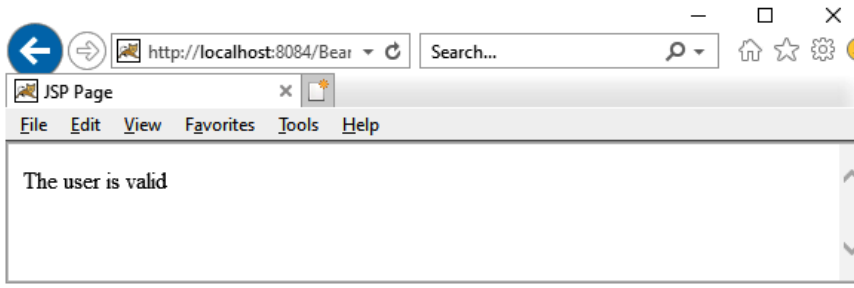


***Figure 10.7**: Output of usebean.jsp*

JSP custom tags

Custom Tags are user-defined action tags and can encapsulate both presentation and business logic.

Custom Tag Library

The contribution of JSP towards segregation of the presentation and business logic has functionally detached the designers from the intricacies of the programming constructs. However, a major part of the work involves writing lengthy and complex business-intrinsic code. In many cases, the structure of the code is such that some of its sections are repetitive and require rewriting.

The custom tags of the tag library provide a mechanism that the programmer can use to encapsulate the complex recurring code or tasks. Once encapsulated, these codes can then be reused in a simpler form. A tag library consists of a collection of functionally related, user-defined XML tags called custom tags.

Just as the JSP action element **`useBean`** facilitates the reuse of the existing beans, the tags in the custom library can be reused to decrease the cycle time for the development of the application and improve productivity. JSP 1.0 does not support tag libraries. However, JSP 1.1 supports the incorporation of the user-created custom tags in a JSP file. The structure of the custom tags in JSP is similar to the XML tags. They are user-defined and explicitly render information about the type of data. Therefore, before we delve into the details about tag libraries, let us appreciate the need for XML and the concept of tags in XML.

Need of XML

Extensible Markup Language or (**XML**) is used in Web applications to create custom or user-defined tags. An XML tag is no different from an HTML tag as it allows the interactions between the user and the browser. What then was the need for a new markup language considering the acceptance and support of HTML? Consider a simple HTML code for displaying the data of an account holder in a bank. The code for displaying this data in HTML would be as follows:

```
<HTML><BODY>
<TABLE align= “center”>
<TR>
<TD>Ms Anne Brown</TD>
</TR>
<TR>
<TD>9, Sunley House, Gunthorpe Street</TD>
</TR>
<TR>
<TD>London E1-7RW</TD>
</TR>
</TABLE>
</BODY></HTML>
```

The Presentation-Centric HTML Output of the code is as follows:

```
Ms Anne Brown
9, Sunley House, Gunthorpe Street
London E1-7RW
```

As displayed in the output of the HTML code is presentation-centric and appears as any textual data. However, there is no differentiation in the data presented for the account holder's first name and last name. In other words, the information about the type of data is lost by using the predefined HTML tags.

On the contrary, consider the same example written in XML using the custom tags (user-defined), as follows:

XML

```
<CUSTOMERDETAILS>
<NAME>
<TITLE>Ms</TITLE>
<FIRSTNAME>Anne</FIRSTNAME>
<LASTNAME>Brown</LASTNAME>
</NAME>
<ADDRESS>
<APTNAME>9, Sunley House</APTNAME>
<STREETNAME>Gunthorpe Street</STREETNAME>
<COUNTRY>London</COUNTRY>
<ZIP>E1 – 7RW</ZIP>
</ADDRESS>
</CUSTOMERDETAILS>
```

The rewriting of the preceding code in XML is as follows:

```
-<xml>
-<CUSTOMERDETAILS>
     -<NAME>
          <TITLE>Ms</TITLE>
          <FIRSTNAME>Anne</FIRSTNAME>
          <LASTNAME>Brown</LASTNAME>
     </NAME>
     -<ADDRESS>
          <APTNAME>9, Sunley House</APTNAME>
          <STREETNAME>Gunthorpe Street</STREETNAME>
          <COUNTRY>London</COUNTRY>
          <ZIP>E1 – 7RW</ZIP>
     </ADDRESS>
</CUSTOMERDETAILS>
```

Needless to say, in the output of the XML code, the definition of the account holder's details is presented in a format with emphasis on the type of data. This is achieved by the use of specific custom tags such as Firstname, Lastname, street, and zip. The use of these

tags makes it easier to differentiate the data. For example, the text enclosed within the <COUNTRY>AND</COUNTRY> tags define the country to which the account holder belongs. Although the presentation of this data is similar to that written in HTML, the code of XML is more data-centric.

The advantages of using XML are as follows:

- **Easy coding**: Being similar to HTML, XML is easy to code, except for the inclusion of custom tags.
- **Easy data interchange**: Translating an XML document is easy due to explicit custom tags. Therefore, the data can be exchanged at different levels without decoding or interpreting its structure.
- **Easy business communication**: The data can be exchanged between organizations without the need to understand the intricacies of the counterpart's business, system organization, or structuring. However, there needs to be a common understanding of the tags used in the data.

The detailed rules and specifications followed in the XML code are written in the **Document Type Definition** (**DTD**). The DTD defines the tags, tag structures, attributes, and values that are used in a particular document.

Custom Tags

JSP 1.1 supports the creation of custom tags that enable the segregation of business complexities to form the content presentation. The structure of a custom tag in JSP, like those in XML and HTML, contains the start tags, end tags, and a body. The structure of a custom tag can thus be represented as follows:

Syntax:

```
<start tag>
     Body
</end tag>
```

Depending upon the presence or absence of the tag body, the custom tags can be categorized as body tags or empty tags. In addition to this, the custom tags can also be nested or include attributes. A nested tag contains tags of various levels. A tag with an attribute uses the attribute parameters to customize the tag behavior.

The attributes of a custom tag are listed as follows:

- **`name`**: It is used to specify the name used within the tag.
- **`required`**: It is used to specify the tag requirement. The values accepted by this attribute are true, false, yes, or no.

Advantages of using Custom Tags

Scriptlets and custom tags encapsulate the Java code snippets, and are hence, functionally quite similar. However, by providing better packaging functionalities, the custom tags productively contribute towards segregating the work profiles of the Web designers and developers. The advantages of using custom tags are as follows:

- **Reduction of scriptlets in the code**: The attributes of a custom tag can be used to accept the parameters. Therefore, the inclusion of the declarations (to define the variables) and scriptlets (to set the properties of the Java components) can be avoided or reduced.
- **Reusability**: Contrary to scriptlets that are non-reusable Java code snippets, the custom tags can be reused. This enables saving the time spent in the development and deployment of codes.

Components of a Tag Library

A JSP file use custom tags that consists of the following:

- **Tag handler**: Tag Handler is used to define custom tags. The tag handler uses different methods and objects to define the behavior of the tag
- **Tag Library Descriptor (TLD) file**: The TLD file is an XML file that contains a descriptive list of the custom tags.

The JSP file contains a tag along with the HTML code for the presentation content. Considering this structure, it is evident that the developer is responsible for coding the tag handler and the TLD file.

The web designer codes for the static display content of the web page and the custom tag is then added to the JSP file. Thus, the custom tag feature of JSP enhances the productivity of the quality web applications by separating the work profiles of the web designer and developer.

The following are the steps to use tags in a JSP file:

- Create a tag handler containing the Tag and BodyTag interfaces. The tag handler file should define the tasks to be performed by the tags.
- Map the tags and the tag handler file by using the TLD file. The TLD file should define the inputs to the tag handler.

The JSP file should include the **`taglib`** directive specifying the use of tags and the definition of the tag.

For creating any custom tag, we need to complete the following steps:

- Create the Tag handler class and perform an action at the start or the end of the tag.
- Create the **Tag Library Descriptor** (**TLD**) file and define the tags.
- Create the JSP file that uses the Custom tag defined in the TLD file.

The structure of Tag Handler

The Tag Handler is used to define the working of the custom tags. The class file derives its methods from the **`javax.servlet.tagext`** package and implements the **`TagSupport`** or the **`BodyTag`** Support interfaces. The **`javax.servlet.jsp.tagext`** package contains classes and interfaces for the JSP custom tag API. The **`JspTag`** is the root interface in the Custom Tag hierarchy. *Figure 10.8* shows the hierarchy of the JSP tag; the JSP Tag interface is a marker interface, therefore, it does not have any method:

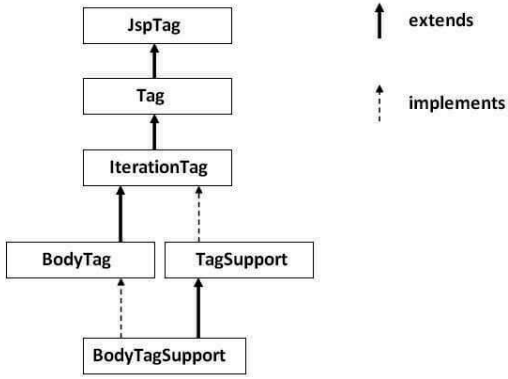


***Figure 10.8**: Hierarchy of JSP tag*

The **`TagSupport`** interface is implemented for tags with an empty body, and the **`BodyTagSupport`** interfaces are implemented for tags that use a body. The tag handler definition can also include classes from other packages, such as **`javax.servlet.jsp`**, and java.io. It is, therefore, essential to add the corresponding import statements for the tag implementations.

The structure of the tag handler can be categorized as that for a basic tag, a tag with attributes, and a tag with a body. Therefore, the methods to be implemented in the tag handler will depend upon the structure of the tag, as listed in *Table 10.3*:

Structure of the Tag Handler	Methods to be implemented
Simple Tag with no body and no attributes	`doStartTag()`, `doEndTag()`, and release
Tag with attributes	`doStartTag()`, `doEndTag()`, and the respective `setAttribute()` and `getAttribute()` methods for each of the tags defined.

***Table 10.3**: Method to be implemented in the Tag Handler class*

The functionality of the tag library is defined using the methods from the abstract class tag. Some methods of the tag class and their return type are listed in *Table 10.4*, *Table 10.5*, and *Table 10.6*:

S.No	Method	Description
1	**`public int doStartTag()`**	Initializes the tag handler and establishes connectivity with a database, if required.
2	**`public int doEndTag()`**	Performs post-tag tasks such as writing the output and closing the database connection.
3.	**`public void release()`**	Removes the instance of the tag handler.
4.	**`public int doAfterBody()`**	It is invoked after the completion of the evaluation of the body tag.
5.	**`public int doBeforeTag()`**	It is invoked before the evaluation of the Body Tag.

6	`Public void setPageContext (PageContext pc)`	It sets the given PageContext object.
7	`public Tag getParent()`	It sets the parent of the tag handler.

***Table 10.4**: Methods of the Tag Interface*

S.No	Field	Description
1	`public static int SKIP_BODY`	Used in empty tags to direct the JSP engine to skip the body of the tag and subsequently invoke the next method, `doEndTag()`.
2	`public static int EVAL_BODY_INCLUDE`	Used to direct the JSP engine to process the body content of a Tag. This method is used only if the interface is implemented as `javax.servlet.tagext.Tag`.
3	`public static int EVAL_BODY_TAG`	Used to direct the JSP engine to process the body of a tag. This method is used only if the interface implemented is `javax.servlet.tagext.BodyTagSupport`.

***Table 10.5**: Return type of doStartTag() method*

S.No	Field	Description
1	`SKIP_PAGE`	Used to specify skipping or omission of the evaluation of the rest of the JSP page.
2	`EVAL_PAGE`	Used to specify the evaluation of the rest of the JSP page.

***Table 10.6**: Return type of doEndTag() method*

In addition to the preceding methods, the tag handler also includes the following classes and methods:

- The **`JspWriter()`** method has to be explicitly mentioned in the tag to write the output to a JSP page.

- The **`getAttribute()`** and **`setAttribute()`** methods are used to retrieve the variable values from scriptlets. After processing the variable, its value is then set by using the **`setAttribute()`** method. When using the **`getAttribute()`** and **`setAttribute()`** methods, the details about the scripting variables also need to be specified by using the **`TagExtraInfo`** class.

- The **`TagExtraInfo`** class consists of methods such as **`getVariableInfo()`** to return the information about the scripting values retrieved from scriptlets, **`TagExtraInfo()`**, that is the default constructor for this class, and the **`setTagInfo()`** and **`getTagInfo()`** methods to set and get the **`TagInfo`** object for this class.

The execution cycle of a JSP file containing the Custom Tags is shown in *Figure 10.9*:

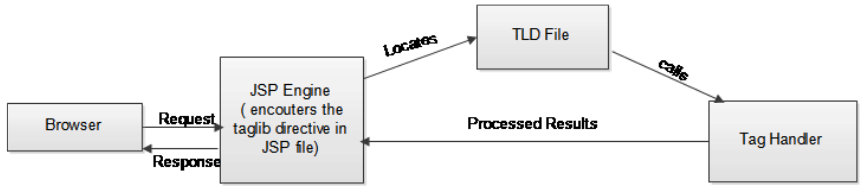


***Figure 10.9**: Execution cycle of JSP file having Custom Tag*

The sequence of the execution of a JSP file containing the custom tags is listed as follows:

1. When the JSP engine identifies the **`taglib`** directive in the JSP page, it recognizes the presence of a custom tag associated with the JSP file. The **Uniform Resource Identifier** (**URI**) and the prefix for the tag act as the referential data for specifying the unique URI and the name for the tag.

2. The specified tag handler is initialized.

3. The **`get()`** and **`set()`** methods for each tag is executed.

4. The **`doStartTag()`** method is invoked and used to perform tasks such as opening a connection to a database.

5. The tag body is evaluated next, but is skipped if the **`SKIP_BODY`** field constant is specified.

6. The tag's output is stored in a special **`PrintWriter`** called the **`JSPWriter`**. The **`pageContext.getOut()`** method is used to make the contents available to the subsequent methods. The output is not forwarded to the client at this stage.

7. Next, the **`doAfter()`** method is invoked to process the content generated after the evaluation of the tag body. The **`SKIP_BODY`** and the **`EVAl_BODY_TAG`** field constant can be returned to estimate the exact stage of the life cycle.

8. The **`doEndTag()`** method is invoked next. All connections created earlier are closed, and the output is directed to the browser.

The structure of the TLD file

The TLD file is an XML file that contains the tag library description. It contains the list and description of all the custom tags in the library that are used as a reference to validate the existence of the respective tags. The components of the TLD file can be broadly classified into two groups. The first group placed within the **`taglib`** tag contains the elements that are a part of the tag element, written as **`<tag>`**.

The elements of the TLD file at the **`taglib`** level are listed in *Table 10.7*:

S.No	Component	Description
1	**`<tlib-version>`**	The version of the tag library such as `<tlib-version>1.0</tlib-version>`.
2	**`<jsp-version>`**	The version of JSP that the tag library depends on, such as **`<jsp-version>1.2</jsp-version>`**.
3	**`<short-name>`**	The name for the tag library.
4	**`<uri>`**	The universal resource identifier, an optional component that is a unique ID for the tag library.
5	**`<info>`**	The detailed information about the tag library.

Table 10.7: *Elements of the TLD file at the Taglib level*

The elements of the TLD file at the tag level are listed in *Table 10.8*:

S.No	Component	Description
1	**<name>**	Defines a name for the tag.
2	**<tagclass>**	Specifies the tag handler class. The format for this specification is `<tagclass>package.classname</tagclass>`.
3	**<info>**	Provides additional information about the tag and its functionality.
4	**<attribute>**	Specifies the attribute name and requirement specification for the tag.
5	**<bodycontent>**	Contains the definition of the body for the tag. Specify empty if the tag is empty, specify JSP if the body content is in JSP, and specify tag dependent if the tag itself controls any part of the body content. The default value for the body content is JSP.

***Table 10.8**: Elements of the TLD file at the tag level*

The steps to create the TLD file for a tag named first in the example tag library are listed as follows:

1. In the Notepad, include the following definitions for the document type and its definition (DTD) as a first line of the TLD file:

```
<!DOCTYPE taglib    PUBLIC "//Sun Microsystems,
Inc.//DTD JSP Tag Library 1.2//EN"  "http://java.
sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
```

2. Add the **tlibversion**, **jspversion**, and **uri**, **short-name** along with their relevant information within the **<taglib>** and **</taglib>** tags, as follows:

```
<taglib>
    <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>simple</short-name>
  <uri>http://tomcat.apache.org/example-taglib</uri>
```

3. Add the tag definitions separating each element within the **<tag>** and **</tag>** tags, as follows:

```
tag>
<name>today</name>
<tag-class>IOL.TagHandler</tag-class>
</tag>
```

4. Add the end tag for **taglib**, as follows:

```
</taglib>
```

The structure of the JSP File

A JSP page using custom tags from the tag library specifies the tag usage with the **taglib** directive. The following code snippet declares a tag library usage:

```
<%@ taglib uri="WEB-INF/example.tld" prefix="example" %>
```

The two attributes for the **taglib** directive are **uri** and **prefix** which are used to specify the unique identifier and a reference name for the particular tag library. When the JSP engine encounters insertion of the **taglib** directive, it uses **uri** to locate the descriptor file for the particular library. To specify the inclusion of a new tag named first for the tag library example, the tag would be written as follows:

```
<example:today>
```

Example:

Tag Handler

```
package IOL;
import java.util.Calendar;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
public class TagHandlerBank extends TagSupport {
    private String Name=null;
       public String getBankName()
          {
          return Name;
          }
```

```
       public void setBankName(String Name)
          {
          this.Name = Name;
           }
   public int doStartTag() throws JspException
{
     JspWriter out=pageContext.getOut();//returns the
instance of JspWriter
       try{
        out.print(Calendar.getInstance().
getTime());/*printing date and time using JspWriter*/
out.print("Your Bank Name is "+Name);
       }catch(Exception e){System.out.println(e);}
    return SKIP_BODY;//will not evaluate the body content
of the tag
}
}
```

TLD File:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/
ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://java.sun.com/
xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
jsptaglibrary_2_1.xsd">
<tlib-version>1.0</tlib-version>
<jsp-version>1.0</jsp:version>
<uri>/WEB-INF/tlds/MYTLD</uri>
<tag>
<name>bank</name>
<tag-class>IOL.TagHandlerBank</tag-class>
<body-content>empty</body-content>
<attribute>
<name>Name</name>
<required>true</required>
<type>java.lang.String</type></attribute></tag>
</taglib>
```

JSP File

```
<html>
<body>
<%@ tagliburi="WEB-INF/tlds/MYTLD.tld" prefix="n" %>
<n:bank Name="My Bank"></n:bank>
</body>
</html>
```

Expression Language(EL)

The **Expression Language** (**EL**) simplifies the accessibility of the data stored in the Java Bean component and other objects like request, session, application, and so on. There are many implicit objects, operators, and reserve words in EL. It is the newly added feature in the JSP technology version 2.0.

Syntax:

```
${ expression }
```

Example: In the following example, **`El.jsp`** uses the **`param`** object to get the request **Parameter**:

`index.jsp`:

```
<html>
<body>
<form action="El.jsp">
Enter Name :<input type="text" name="name" /><br/><br/>
<input type="submit" value="submit"/>
</form>
</body>
</html>
```

`el.jsp`:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
      Welcome, ${ param.name }
</html>
```

The following is the output of `index.html`, as shown in *Figure 10.10:*

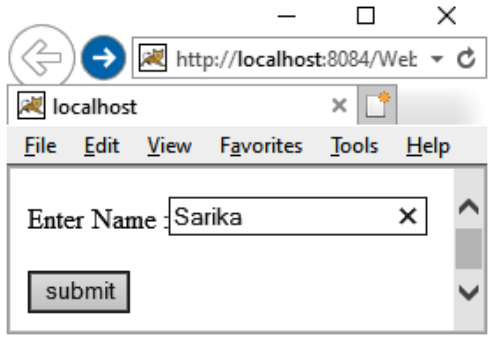


***Figure 10.10**: Output of index.html*

The following is the output of `el.jsp`, as shown in *Figure 10.11:*

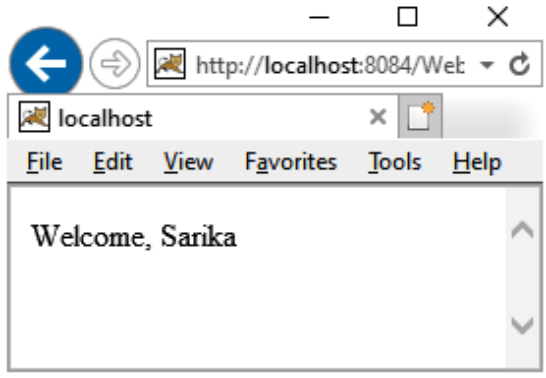


***Figure 10.11**: Output of El.jsp*

There are many implicit objects used by the Expression language. Some implicit objects are listed in *Table 10.9*:

S.No	Implicit Objects	Usage
1	`pageScope`	It maps the given attribute name with the value set in the page scope.
2	`requestScope`	It maps the given attribute name with the value set in the request scope.
3	`sessionScope`	It maps the given attribute name with the value set in the session scope.
4	`applicationScope`	It maps the given attribute name with the value set in the application scope.

S.No	Implicit Objects	Usage
5	**param**	It maps the request parameter to the single value.
6	**paramValues**	It maps the request parameter to an array of values.
7	**header**	It maps the request header name to the single value.
8	**headerValues**	It maps the request header name to an array of values.
9	**cookie**	It maps the given cookie name to the cookie value.
10	**initParam**	It maps the initialization parameter.
11	**pageContext**	It provides access to many objects request, session, and so on.

***Table 10.9**: Implicit Objects*

The following is an example of Session and Cookie:

index.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
<%
session.setAttribute("user","Sarika");
Cookie ck=new Cookie("pwd","sailboat");
response.addCookie(ck);
%>
<a href="process.jsp">visit</a>
</html>
```

Process.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
      Name= ${ sessionScope.user }
       Your Password = ${cookie.pwd.value}
</body>
</html>
```

The following is the output of `index.jsp`, as shown in *Figure 10.12:*

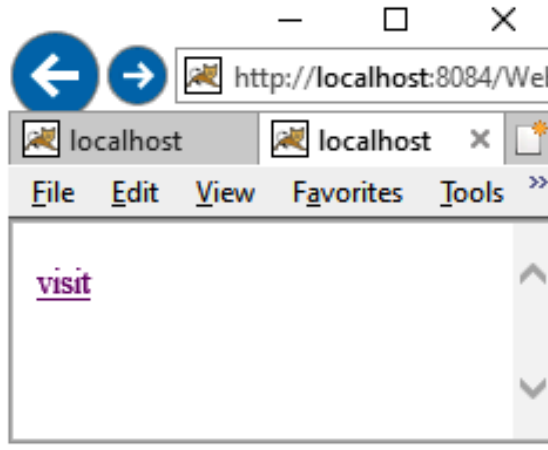


***Figure 10.12**: Output of index.jsp*

The following is the output of `process.jsp`, as shown in *Figure 10.13:*

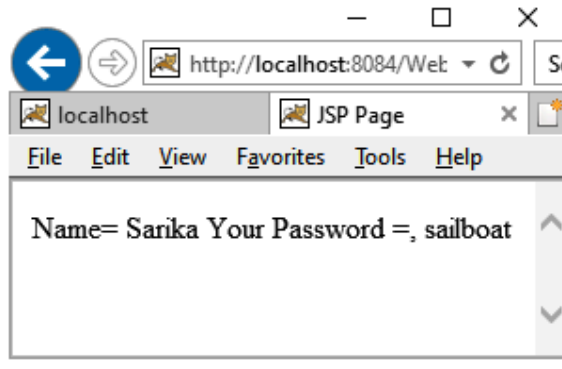


***Figure 10.13**: Output of Process.jsp*

Model View Controller (MVC) Architecture in JSP

MVC is a software design pattern that separates presentation logic from business logic and data. It has three interconnected elements, which are as follows:

- Model
- View
- Controller

The Model defines the business logic of the application. It is independent of the user interface. The Controller manages the flow of the application. The Controller fetches the data from the Model and gives it to the View. The View defines the presentation layer of the application. The Model consists of simple Java classes, the Controller consists of servlets, and the View consists of the JSP pages.

The following are the characteristics of MVC Pattern:

- It separates the presentation layer from the business layer.
- The Controller invokes the Model and sends the data to View.
- The Model is not even aware that it is used by some web application or a desktop application.
- Navigation control is centralized.
- It is easy to maintain the large application.

Refer to *Figure 10.14* that illustrates the MVC architecture:

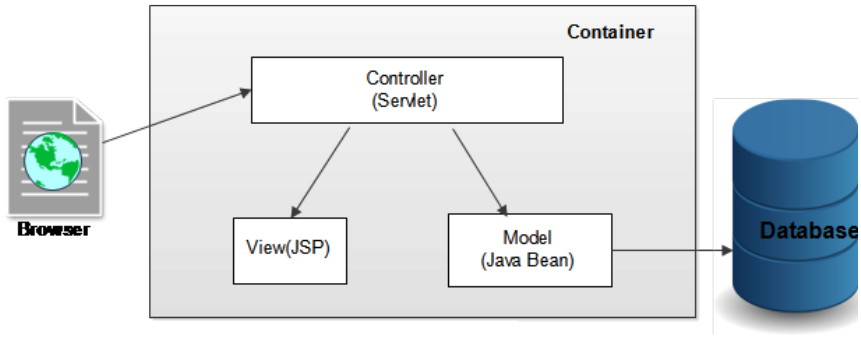


***Figure 10.14**: MVC Architecture*

MVC Example in JSP

In the preceding figure, we use a servlet as a controller, JSP as a view component, and Java Bean class as a model.

In this example, we have created the following five pages:

- **`index.html`**: A page that gets the input from the user.
- **`Controller.java`**: A servlet that acts as a controller.
- **`login-success.jsp`** and **`login-error.jsp`** files: Acts as view components.
- **`web.xml`**: File for mapping the servlet.

Example:

`index.html`:

```
<html>
    <body>
            <form action="Controller" method="post">
               Name:<input type="text" name="name"><br>
               Password:<input type="password"
               name="password"><br>
               <input type="submit" value="login">
            </form>
        </body>
</html>
```

`Controller.java`:

```
package IOL;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Controller extends HttpServlet
 {
```

```
    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException
 {
      response.setContentType("text/html");
      PrintWriter out=response.getWriter();
      String name=request.getParameter("name");
      String password=request.getParameter("password");
      LoginBean bean=new LoginBean();
      bean.setName(name);
      bean.setPassword(password);
      request.setAttribute("bean",bean);
      boolean status=bean.validate();
       if(status)
       {
          RequestDispatcher rd=request.
          getRequestDispatcher("logini-success.jsp");
          rd.forward(request, response);
        }
        else
        {
          RequestDispatcher rd=request.
          getRequestDispatcher("login-error.jsp");
          rd.forward(request, response);
        }
     }
    @Override
    public String getServletInfo() {  return "MVC"; }
}
```

Login Bean.jsp:

```
package IOL;
public class LoginBean
{
  private String name,password;
```

```
  public String getName()
{
    return name;
}
public void setName(String name)
 {
    this.name = name;
}
public String getPassword()
{
    return password;
}
public void setPassword(String password)
 {
this.password = password;
}
public boolean validate()
{
    if(password.equals("admin"))
{
        return true;
       }
    else
{
        return false;
    }
}
}
```

Logini-success.jsp:

```
%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
          <meta http-equiv="Content-Type" content="text/
```

```
            html; charset=UTF-8">
            <title>Successfully Login</title>
        </head>
        <body>
            <%@page import="IOL.LoginBean"%>
            <p>You are successfully logged in!</p>
            <%
                LoginBean bean=(LoginBean)request.
                getAttribute("bean");
                out.print("Welcome, "+bean.getName());
        %>
    </body>
</html>
```

Login-error.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
            <title>Error Page</title>
        </head>
        <body>
            <p>Sorry! username or password error</p>
            <%@ include file="index.html" %>
</body>
</html>
```

Web.xml:

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

```
        <servlet-name>Controller</servlet-name>
        <servlet-class>IOL.Controller</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>Controller</servlet-name>
        <url-pattern>/Controller</url-pattern>
</servlet-mapping>
<session-config>
        <session-timeout>
                30
        </session-timeout>
</session-config>
</web-app>
```

The following is the output of `index.html`, as shown in *Figure 10.15*:

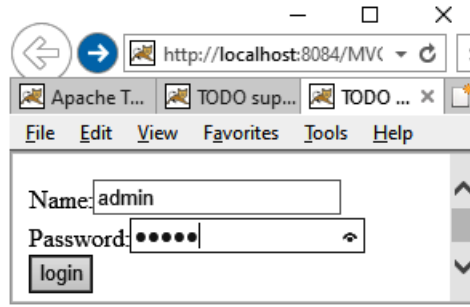


***Figure 10.15**: Output of index.html*

The following is the output of `logini-success.jsp`, as shown in *Figure 10.16*:

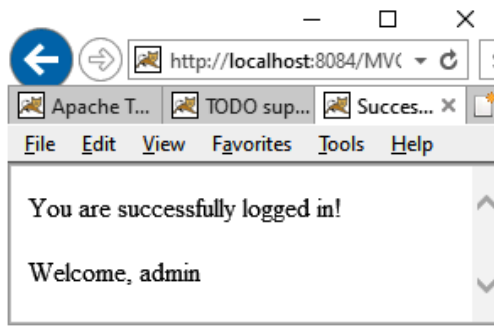


***Figure 10.16**: Output of logini-success.jsp*

The following is the output of `login-error.jsp`, as shown in *Figure 10.17*:

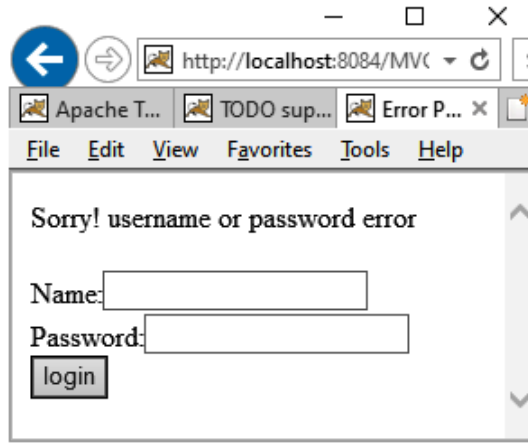


***Figure 10.17**: Output of login-error.jsp*

Conclusion

In this chapter, we learned that JSP actions such as **`useBean`**, **`getProperty`**, **`setProperty`**, **`setProperty`**, and forward are used to perform tasks such as inserting files, reusing beans, forwarding a user to another page, and instantiating objects. The custom tag library provides a mechanism by using which the programmer can encapsulate the recurring code or tasks and reuse them in multiple applications. Custom tags like XML and HTML tags contain the start and end tags and a body. Custom tags can be categorized as simple tags, body tags, tags with attributes, or nested tags. Attributes contain parameters that can be used to customize the behavior of the custom tags.

The tag handler and TLD files are created to implement the functionality of the custom tags in a JSP file. The tag handler is used to define the custom tags and is derived from the **`javax.servlet.tagext`** package that implements the tag or the **`TagBody`** interfaces. The tag handler consists of the **`doStartTag()`**, **`doEndTag()`**, and **`release()`** methods.

The **tag library definition** (**TLD**) file is an XML file that contains the tag library description and consists of the following tags:

- taglib
- tlibversion
- jspversion
- info
- name
- tagclass
- bodycontent

The errors in JSP can be categorized as either translation time errors or request time errors. The error page is a separate file used to trap the runtime errors of a JSP page. A tag library can be deployed either by separate placements of files or by packaging the files. The package files created to contain the TLD file and the class files are jar and war files. Expression Language simplifies the accessibility of the data stored in the Java Bean component and other objects like request, session, application, and so on. MVC architecture separates business logic from presentation logic.

In the next chapter, you will learn Structs Framework with a simple example.

Questions

1. **Which JSP action tag is used to include the content of another resource, be it JSP, HTML, or servlet?**
   a. Jsp:include
   b. Jsp:forward
   c. Jsp:page
   d. Jsp: param

Ans: a

2. What is the requirement of a tag library?
3. What is a JSP Expression?
4. How do you include static files on a JSP page?
5. How is Taglib Directive used in JSP?
6. What are custom tags, how do you use them, and how do you create them in JSP?.

7. What is the difference between custom JSP tags and JavaBeans?
8. What is EL?
9. How does EL search for an attribute?

Chapter 11
Introduction to Struts

Introduction

Struts is a framework based on an MVC pattern to develop web applications. The pattern is the way you can architect your application, whereas Framework provides the foundation classes and libraries. The Framework helps in the rapid development of common classes. It leverages industry best practices. Craig McClanahan built the Struts Framework and submitted it to the Apache Foundation in May 2000, and Struts 1.0 was released in June 2001. Struts 2.5.22 is the most recent stable release, which was released on November 19, 2019. There is some difference between struts1 and struts2.

Table 11.1 provides the differences between Struts1 and Struts2:

No.	Struts1	Struts2
1	Action class is not POJO. You need to inherit the abstract class.	The action class is POJO. You don't need to inherit any class or implement any interface.
2	Front controller is `ActionServlet`.	Front Controller is `StrutsPrepareAndExecuteFilter`.

No.	Struts1	Struts2
3	It uses the concept of the `RequestProcessor` class while processing requests.	It uses the concept of **Interceptors** while processing the request.
4	It has only JSP for the view component.	For the view component, it has JSP, Freemarker, Velocity, and so on.
5	The configuration file name can be [anyname].xml, and it can be placed inside the WEB-INF directory.	The configuration file must be struts.xml and placed inside the classes directory.
6	Action and Model are separate.	Action and Model are combined within the action class.

***Table 11.1**: Difference between Struts1 and Struts2*

Structure

In this chapter, we will cover the following topics:

- Features of Struts
- Architecture of Struts
- Components of Struts
- Create an application

Objective

In this chapter, you will learn about the struts framework that simplifies the development and maintenance of web applications by providing predefined functionality. The features and components of Struts2 will also be learned. An application based on the framework will also be created to understand the Strust2 framework.

Features of Struts2

The following are the main features of Struts2:

- **Configurable MVC components**: In the struts2 framework, we provide all the components' (view components and

Action) information in the struts.xml file. We may simply alter any information in the XML file if we need to.

- **POJO-based actions**: The action class in struts2 is a **Plain Old Java Object** (**POJO**), which is a simple Java class. You are not required to implement any interfaces or inherit any classes in this environment.
- **AJAX support**: Struts2 has the support of Ajax technology. It's used to make asynchronous requests, which means it doesn't keep the user waiting. It just sends the needed field data to the server, not all of it. As a result, the performance is quick.
- **Integration support**: We can simply integrate the struts2 Application with framework like hibernate, spring, tiles, and so on.
- **Various result types**: We can use JSP, freemarker, velocity, etc technologies as a result in struts2. Velocity and freemarker is a templating engine based on Java. Both are a free and open-source web framework that's meant to be used as a view component in the MVC architecture.
- **Various tag supports**: To make the struts2 applications easier, Struts2 includes many sorts of tags, such as UI tags, Data tags, Control tags, and so on.
- **Theme and Template support**: Struts2 supports three different types of themes – XHTML, simple, and CSSXHTML. The default theme for struts2 is XHTML. For a consistent appearance and feel, themes and templates can be employed.

Components of Struts2

The Model-View-Controller pattern in Struts2 is implemented with the following five core components:

- **Actions**: . Action class is responsible for maintaining the state and request processing. Action class is a Plain Java class which has a getter, setter, and an execute method. It represents the MODEL part of the MVC pattern
- **Controller**: In struts2, a controller is responsible for identifying the incoming requests, selecting the appropriate action classes for the processing, and finally searching the

JSP or HTML files to display the response. Controller class (`org.apache.struts2.dispatcher.FilterDispatcher`) is provided by the struts framework.

- **Interceptors**: Interceptors are the helper component for the container that contains the cross-cutting logic and can be extended in the following two cases:
    - Before the execution of Action
    - After the execution of Action

  Therefore, an interceptor is an object invoked at the preprocessing and postprocessing of a request. In Struts 2, Interceptor is used to perform operations such as validation, exception handling, internationalization, displaying intermediate results, and so on.
- **Value Stack**: A Value Stack is simply a stack that contains the application-specific objects such as action objects and other model objects. Value stack fetches the object through the struts tag with the help of an expression called OGNL (Object Graph Navigation language). A value stack is a storage area where the data is stored for processing the client requests.
- **Results/Result types**: In struts2, the result component is responsible for identifying the appropriate JSP or HTML to display the data to the client. We can also customize this component according to the requirement.
- **View technologies**. It is an HTML or JSP file that is used to display the information to the user.

Architecture of Struts2

Struts2 Architecture is different from the traditional MVC Pattern. Here, Action is a Model, and the controller class is maintained by the struts2 Framework. In Struts 2, Servlet Filter (Filter Dispatcher) looks at the request, and then as per the mapping of the URL, the request is forwarded to the appropriate Action Class, as shown in *Figure 11.1*:

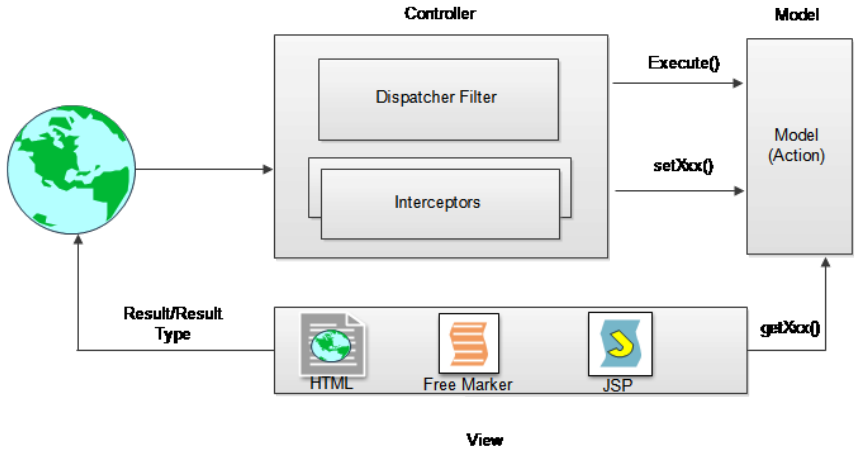


***Figure 11.1**: Architecture of Strut2*

In *Figure 11.1*, **Browser** sends a request. The request passes to the **Controller.** The Controller is implemented with a Struts2 dispatch servlet filter and the interceptors. The Controller finds the appropriate action class, sends the request, sets the parameters, and calls the **`execute()`** method. The configured interceptor functionalities apply, such as validation, file upload, and so on. Finally, the view prepares the result and returns it to the user.

Creating a Struts Application

The following are the steps to create a Struts Application:

1. Create the **`index.html`**.
2. Create the action class **`Login.java`**.
3. Create the view **`Welcome.jsp`** and **`Invalid.jsp`**.
4. Add the Struts configuration in **`struts.xml`**.
5. Add the filter in **`web.xml`**.
6. Build the WAR file and run the application.

Refer to the following code example:

`index.html`

```
<html>
      <head>
```

```
            <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
            <title>JSP Page</title>
      </head>
      <body>
            <form action="login.action">
                  User Name<input type="text"
                  name="uname"/><br>
                  <input type="submit" value="submit"/>
            </form>
      </body>
</html>
```

Action class-Login.java:

```
package p1;
public class Login
{
      private String uname;
      public String execute()
      {
            if(uname.equals("Sarika"))
                  return("Success");
            else
                  return("Invalid");
    }

    public String getUname()
 {
            return uname;
      }

    public void setUname(String uname)
{
        this.uname = uname;
    }
}
```

View – Welcome.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
    <title>JSP Page</title>
    </head>
    <body>
       <h1>welcome</h1>
       <%@ taglib uri="/struts-tags" prefix="s" %>
       <h1><s:property value="uname"/><br/></h1>
    </body>
</html>
```

View--Invalid.jsp

```
    <%@page contentType="text/html" pageEncoding="UTF-8"%>
    <!DOCTYPE html>
    <html>
       <head>
          <meta http-equiv="Content-Type" content="text/
          html; charset=UTF-8">
          <title>JSP Page</title>
       </head>
       <body>
             <h1>invalid</h1>
       </body>
</html>
```

Struts.xml

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration
2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
       <package name="default" extends="struts-default">
```

```
        <action class="p1.Login" name="login">
            <result name="Success">/welcome.jsp</result>
            <result name="Invalid">/invalid.jsp</result>
         </action>
      </package>
</struts>
```

Web.xml:

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

OUTPUT:

The following is the output of `index.jsp`, as shown in *Figure 11.2*:

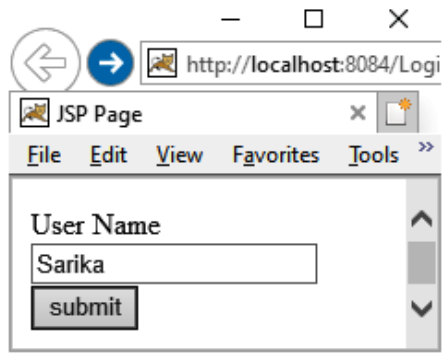


***Figure 11.2**: Output of index.jsp*

The following is the output of `welcome.jsp`, as shown in *Figure 11.3*:

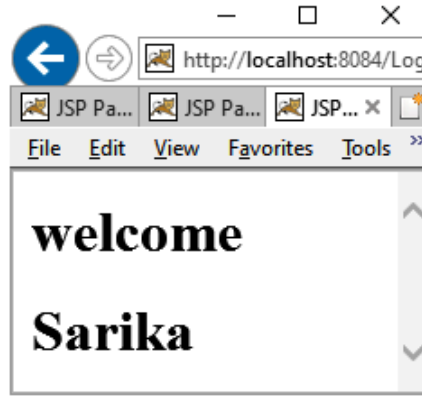


***Figure 11.3**: Output of Welcome.jsp*

The following is the output of `index.html`, as shown in *Figure 11.4*:

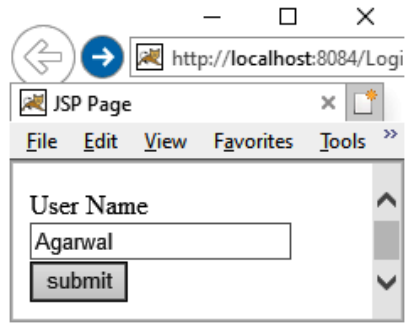


***Figure 11.4**: Output of index.html*

The following is the output of `invalid.jsp`, as shown in *Figure 11.5*:

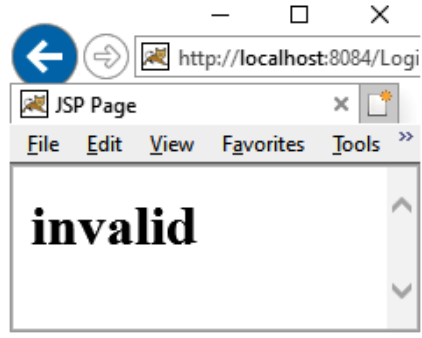


***Figure 11.5**: Output of Invalid.jsp*

Conclusion

The struts2 framework is used to develop an MVC-based web application. Struts is an open-source framework that extends the Java Servlet API and employs a Model-View-Controller (MVC) architecture. It enables you to create maintainable, extensible, and flexible web applications based on standard technologies, such as JSP pages, JavaBeans, resource bundles, and XML.

The components of struts are Controller, Action, ValueStack, View, and ResultType. Struts.xml is used to specify the configuration about the various components of the application, that is, the type of result, Interceptor, information about JSP, and so on.

Web.xml must have the entry of the filter class.

Questions

1. What is Struts?
2. What is the difference between struts1 and struts2?
3. What are the features of Struts?
4. Define Interceptors, ValueStack, and OGNL?
5. How does FrontController help in maintaining the MVC pattern in struts2?

Interview Questions

1. **What are server-side includes?**

   Ans: A server-side enables you to embed a Java Servlet in an HTML document. The file that contains a server-side include must be saved with a **`.shtml`** extension to inform the server that the file contains a server-side include.

2. **What are thread-safe servlets?**

   Ans: In a typical environment, the webserver creates only a single instance of a servlet. The **`service()`** method of the Servlet acts as a dispatcher of the requests. If the service method is processing a request and, at the same time, receiving a different request, the webserver can create a different thread and execute the **`service()`** method in the newly created thread. In the process of execution, the threads need to be in synchronization and, thus, make the servlets thread-safe. Servlets can make use of the Java API to implement thread synchronization.

3. **Can the address of the client that is sending a request be tracked in the Servlet?**

   Ans: Yes, the address of the remote client can be tracked in the Servlet by using the **`getRemoteAddr()`** and **`getRemoteHost()`** functions of **`HttpServletRequest`**.

4. **How can the client be intimated about the size of a file that is being sent by the server?**

   Ans: The content-length attribute of the HTTP header can be used to set the file size and sent to the length.

5. **Can I read cookies with a specific key?**

   Ans: No. All the cookies need to be retrieved, and then the key name and value should be checked individually.

6. **What are persistent cookies?**

   Ans: Few cookies can get deleted after a session is over. To make the cookies persistent, you can set an expiry time for the cookies. These types of cookies are known as persistent cookies.

7. **How do we get the absolute URL of a servlet at runtime?**

   Ans: You can use the **`getRequestURL()`** method of the **`HttpServletRequest`** interface to obtain the URL of the request, excluding the query string. The **`getQueryString()`** function of the **`HttpServletRequest`** interface can be used to obtain the query string that was part of the request. In addition, you can also use the **`getServerName()`** and the **`getServerPort()`** methods of the **`ServletRequest`** interface to obtain the name of the server which received the request and the port number through which the Servlet receives the request.

8. **What is the use of the `getScheme()` method?**

   Ans: The **`getScheme()`** method is used to obtain the scheme based on which the URL is constructed and sent to the Servlet. The scheme can be ftp, http, or https.

9. **What is the life cycle of a session?**

   Ans: There are three stages in the life of a session, which are as follows:

   a. **New**: Until the Session is being established with the client.

b. **Joins**: The Session will always be new if the client does not join a session.

c. **Destroyed**: The Session is invalidated or the session timeout period expires.

10. How is `PrintWriter` different from `ServletOutputStream`?

Ans: **`PrintWriter`** is a character-stream class, whereas **`ServletOutputStream`** is a byte-stream class. The **`PrintWriter`** class can be used to write only character-based information, whereas the **`ServletOutputStream`** class can be used to write primitive values and character-based information.

11. Can a JSP be called using a Servlet?

Ans: Yes, Servlet can call a JSP using the RequestDispatcher interface.

The following is an example:

```
RequestDispatcher reqdis=request.
getRequestDispatcher("log.jsp");
reqdis.forward(request,response);
```

12. What is load-on-startup in Servlet?

Ans: The load-on-startup element of Servlet in **`web.xml`** is used to load the Servlet at the time of deploying the project or the server to start. This saves time for the response of the first request.

13. Can you refresh the Servlet in client and server-side automatically?

Ans: Yes, there are a couple of primary ways that a servlet can be automatically refreshed. One way is to add a "Refresh" response header containing the number of seconds after which a refresh should occur.

The following is an example:

```
response.addHeader("Refresh", "5");
```

14. Can you send an Authentication error from a Servlet?

Ans: Yes, we can use the **`setStatus(statuscode)`** method of **`HttpServletResponse`** to send an authentication error. All we have to do is set an error code and a valid reason along with the error code.

The following is an example:

```
response.sendError(404, "Page not Found!!!" );
```

15. **Why doesn't a Servlet include `main()`? How does it work?**

    Ans: Servlets don't have a **`main()`** method, because servlets are executed using the web containers. When a client places a request for a servlet, the server hands the requests to the web container where the Servlet is deployed.

16. **What is the difference between Context Parameter and Context Attribute?**

    Ans: The main difference is that the Context Parameter is a value stored in the deployment descriptor, which is the web.xml and is loaded during the deployment process. On the other hand, Context Attribute is the value that is set dynamically and can be used throughout the application.

17. **What is servlet mapping?**

    Ans: Servlet mapping is the process of defining an association between a URL pattern and a servlet. The mapping is used to map the requests to Servlets.

18. **Which are the annotations that are used in Servlet?**

    Ans: The three important annotations used in the servlets are as follows:

    a. **`@WebServlet`**: For servlet class.

    b. **`@WebListener`**: For listener class.

    c. **`@WebFilter`**: For filter class.

19. **What is the difference between a Generic Servlet and an HTTP Servlet?**

    Ans: The similarity between Generic Servlet and HTTP Servlet is that both are **Abstract Classes**. However, the differences between them are as follows:

Generic Servlet	HTTP Servlet
Protocol Independent.	Protocol Specific.
Belongs to `javax.servlet` package.	Belongs to `javax.servlet.http` package.
Supports only `service()` method.	Supports `doGet()`, `doPost()`, `doHead()` methods.

20. How can you get the server information in a servlet?

Ans: We can retrieve the information of a server in a servlet. We can use the following code snippet to get the servlet information in a servlet through the servlet context object:

```
getServletContext().getServerInfo();
```

21. What is the JSP technology?

Ans: JSP technology or JavaServerPages technology is an extension of the Java programming technology. JSP includes a scripting language that is Java based. A JSP page upon compilation generates a servlet. Web applications that are developed by using JSP demonstrate the platform and web server independence.

22. How can you eliminate the task of reloading a JSP file?

Ans: A good approach to eliminate the task of having to reload a JSP file is to separately place the static content. Create a separate JSP file for the static content and use directives such as **include** to attach the contents of the file.

23. How does JSP add functionality to an application in the presence of the servlet technology?

Ans: The basic importance of using JSP is the auto-generation of servlets. This feature simplifies a part of the coding phase. The positive implementation of JSP is also brought out when the response output contains both template and business-specific data.

24. How do you correlate JSP and XML?

Ans: The HTML content representing the static content can also be coded by using XML. In addition to this, the JSP content or the scriptlets can also be represented within the XML tags `<jsp:scriplet>` and `</jsp:scriplet>`.

25. Is the Java platform API used with JSP?

Ans: Since the compilation of JSP results in the generation of a servlet, JSP needs JVM support that fulfils the standard followed by the Java Servlets.

26. How do you define a JSP tag library?

Ans: The JSP tag libraries contain user-defined custom tags that are used to write the XML-based codes containing

the JavaBean components. Tag libraries essentially aim at segregating the static and dynamic code content of a JSP file.

27. What are the advantages of using tag libraries?

Ans: The features of a tag library are as follows:

a. Tags created for the JSP pages are portable and can be easily used across projects. Once packaged as a jar or a war file, they can be reused.

b. Tag libraries are maintainable. This is because the dynamic code is placed in the tag handler and the JavaBean component. Any changes that need to be made to the code can thus be simply updated in the corresponding files.

The reusability factor also explains the shortening of the debugging time and thus reduces the time taken to deploy a JSP application that uses tag libraries.

28. How can you restrict the page errors displayed on a JSP page?

Ans: The errors can be stopped from getting displayed by setting up an **`ErrorPage`** attribute of the PAGE directory to the name of the error page in the JSP page, and then in the error JSP page, setting **`isErrorpage=" TRUE`**.

29. How can a thread safe JSP page be implemented?

Ans: It can be done by having them implemented by the **`SingleThreadModel`** Interface. Add **`<%@page isThreadSafe="false" %>`** directive in the JSP page.

30. How can the output of the JSP or servlet page be prevented from being cached by the browser?

Ans: Using appropriate HTTP header attributes can help prevent the dynamic content output by a JSP page from being cached by the browser.

31. How can you disable scripting?

Ans: Scripting can be easily disabled by setting the scripting-invalid element of the deployment descriptor as true. It is a sub-element of the property group. It can be false as well.

32. Is the JSP technology extensible?

Ans: Yes, JSP is easily extensible by the use and modification of tags, or custom actions, encapsulated in the tag libraries.

Index

A
action 175
Active Server Pages (ASP) 41
Applets
  versus Servlets 39
application data 72
autoFlush attribute 127

B
buffer
  about 126
  autoFlush attribute 127
  isELIgnored attribute 126
  isThreadSafe attribute 127

C
Callable Statement object 8
CGI script
  about 40
  disadvantages 40
comment tag 104
Common Gateway Interface (CGI) 40
contentType attribute 122
controller 175
cookie class
  constructor 77
cookies
  about 76
  advantages 76
  characteristics 76
  disadvantages 76
  example 78, 80

custom tag
about 128, 150
advantages 151
attributes 151
components 151
in JSP file 152
JSP file execution 155, 156
custom tag library 147

D

database
connecting to 7
connection, establishing 7
connection object 7
driver, loading 7
querying 7
query, processing 8
Database Management 2, 3
declaration tag 107
deployment descriptor
creating 47, 48
directives
types 120
Document Type Definition (DTD) 150
doGet() method 56

E

errorPage attribute
about 122
isErrorPage attribute 122-124
package list 124, 125
scripting language 125
servlet information 126
true/false 125
execute() method 10
executeQuery() method 10
executeUpdate() method 10
Expression Language (EL)
about 121, 160-163
implicit objects 161, 162
expression tag 106
extends attribute 122
Extensible Markup Language (XML)
about 148
advantages 150
need for 148-150

F

forward action tag
about 141
attribute 140
forward() method
versus sendRedirect() method 92

G

GenericServlet class 49
getAttribute() method 155
getConnection() method 7
GET method
about 42, 54
versus POST method 54
getProperty tag
attributes 137

H

hidden form fields
  about 68
  example 68-71
HTTP Request 52
HTTP Response 52
HttpServlet
  about 43, 49
  HTTP methods 53, 54
HTTP Servlet 52
HTTPServlet class
  need for 52
HttpServletRequest interface 56
  getCookies() 113
  getMethod() 113
  getQueryString() 113
  getSession() 113
HttpServletResponse interface
  addCookie() 113
  sendRedirect() 113
HttpSession 72
HttpSession interface
  methods 72, 73
  program 73, 75
  using 71
HttpSession object
  about 72
  types 72
HyperText Transfer Protocol (HTTP) 52

I

I18N-based application
  developing 27
  examples 27
implicit objects
  about 108
  application 108-110
  config 116
  contentType attribute 122
  exception 116
  extends attribute 122
  out 111
  page 111
  pageContext 116
  request 112
  response 113
  session 114, 115
include action tag
  about 146
  attribute 142
  example 142
include directive
  about 127
  advantages 128
init() method 56
Input Parameters
  passing 15, 17
interceptor 176
Internationalizing with Date and Time 28
Internationalizing with Numbers 30, 31

Inter-Servlet communication
client interface 87
first servlet code 88, 90
implementing, via problem statement 86
index.html 87
SendRedirect 91
tasklist 86
isELIgnored attribute 126
isErrorPage attribute 122-124
isThreadSafe attribute 127

J

Java Database Connectivity (JDBC) 4
Java Server Pages (JSP)
advantages 97
directory structure 101
features 96
lifecycle 98, 99
MVC architecture 164
need for 96
request-response cycle 98
structure 99, 100
versus Servlet 97
java.text.DateFormat
methods 28
javax.servlet.http.Cookie class
about 76
methods 77, 78
Javax.servlet package 42, 43
JDBC-API 4
JDBC classes
Callable Statement object 8
Prepared Statement object 8
Statement object 8
JDBC Driver Manager 5
JDBC drivers
about 4
JDBC Driver Manager 5
JDBC-ODBC bridge 5
JDBC-ODBC bridge driver 5
Native-API driver 5
network protocol driver 6
ODBC driver 5
thin driver 6
JDBC-ODBC bridge
about 5
advantages 5
disadvantages 5
JDBC-ODBC bridge driver 5
JDBC URL
about 7, 8
example 8
JSP action tag
about 132-134
jsp:forward action tag 140
jsp:getProperty action tags 136
jsp:include action tag 142
jsp:setProperty action tags 136
jsp:useBean action tag 134
syntax 132

JSP custom tags 147
JSP directives 120
JSP elements
  about 104
  comment tag 104
JSP file
  structure 158
jsp:forward action tag 140
jsp:getProperty action tags 136
jsp:include action tag 142
jsp:setProperty action tags 136
jsp:useBean action tag 134
jspWriter() method 154

L

Locale class
  about 24
  constructors 24
  methods 24, 25
Localization (L10N) 24

M

Model View Controller (MVC)
  about 164
  example, in JSP 165, 169, 170
MVC architecture
  in JSP 164
MVC pattern
  characteristics 164

N

Native-API driver
  about 5
  advantages 6
  disadvantages 6
network protocol driver
  about 6
  advantages 6
  disadvantages 6
NumberFormat
  methods 30

O

ODBC API 3, 4
ODBC driver 5
Open Database Connectivity (ODBC) 3

P

package list 124, 125
page directive
  about 120
  attributes 120, 121
Plain Old Java Object (POJO) 175
POST method
  about 42, 54
  versus GET method 54
Prepared Statement object
  about 8, 14
  using 14

Q

query string 42, 54

R

Records
  adding 18
  deleting 18, 19
  modifying 18

Relational Database Management Systems (RDBMS) 2
remote method invocation (RMI) 39
RequestDispatcher interface
about 84
methods 85
object method, obtaining 84, 85
resource bundle 27
Resource Bundle
about 25
constructor 26
methods 26
ResultSet object 10-13
result type 176

S

scripting elements
about 104
declaration tag 107
expression tag 106
scriptlet tag 104-106
scripting language 125
scriptlet tag 104-106
SendRedirect
about 91
example 91
sendRedirect() method
about 91
versus forward() method 92
server-side scripting technologies
versus Servlets 40
Servlet
about 38
characteristics 39
classes and interfaces 43
client request components 41
compiling 47
creating 45, 46
lifecycle 44
session techniques 64
versus Applets 39
versus Java Server Pages (JSP) 97
versus server-side scripting technologies 40
working 41
servlet chaining 85
ServletContext interface
methods 84
servlet information 126
Servlet interface
about 44
methods 45
ServletRequest interface
method 57, 60
session techniques
hidden form fields 68
HttpSession interface, using 71
in Servlet 64
URL Rewriting 64, 67, 68
session tracking 64

setAttribute() method 155
setProperty tag
  attributes 137
state data 72
Statement object
  about 8, 10
  using 9
Statement object, methods
  execute() method 10
  executeQuery() method 10
  executeUpdate() method 10
Struts2
  features 174, 175
Struts2 architecture 176, 177
Struts2 components
  about 175
  action 175
  controller 175
  interceptor 176
  result 176
  result type 176
  Value Stack 176
  view technologies 176
Struts application
  creating 177, 180, 181

T

TagExtraInfo() method 155
tag handler
  about 152
  structure 152, 153
tag handler, methods
  getAttribute() method 155
  jspWriter() method 154
  setAttribute() method 155
  TagExtraInfo() method 155
taglib directive 128
tag library
  methods 153
Tag Library Descriptor (TLD) file
  about 128
  creating 157, 158
  elements, at tag level 157
  elements, at taglib level 156
  structure 156
thin driver
  about 6
  advantages 6
  disadvantages 6

U

Uniform Resource Identifier (URI) 155
URL Rewriting
  about 64, 67, 68
  advantages 65
  disadvantages 65
  example 65
useBean tag
  attributes 135
  example 135, 136

V

Value Stack 176

view technologies 176

W

webserver 38

web.xml file

about 47

elements 48