

Advance JAVA



ज्ञानगंगा घोघरी

Yashwantrao Chavan Maharashtra Open University
Dnyangangotri, Near Gangapur Dam, Nashik 422 222

Unit No. and Name	Details
Unit 1: JDBC	JDBC Architecture, Overview of Drivers, DBC Driver Manager, Steps for accessing database using JDBC API, Statements Prepared, Statement Callable, Statement Scrollable and Updatable ResultSet, ResultSet MetaData and Database MetaData, Working with Rowset Interface.
Unit 2: Servlet	Introduction To Java Servlets, Servlet API, Servlet Life- Cycle, Working With Apache Tomcat, Generic Servlets Http Servlet, HttpSession, Session Binding/Tracking, Inter-Servlet Communication.
Unit 3: JSP	JSP SYNTAX, Page Directive, Include Directive, Data Declaration and Method Definition, Scriptlets Implicit Objects, Custom Tags, Session Tracking in JSP, Page Context, Exception
Unit 4: Hibernate	Why Hibernate?, Understanding ORM, Objects and Persistence, Hibernate Architecture, Mapping Documents, Hibernate Database Connection, Creating Persistent Classes, Mapping Collection of Objects, Persistent Object Life Cycle, Hibernate with Servlets, HQL: Hibernate Query Language.
Unit 5: Spring Core	Introduction to Spring Framework, Inversion of Control and Dependency Injection, IOC Container, Bean Creation, Construction Injection, Setter Injection,
Unit 6: Spring MVC	Spring Web MVC, MVC Architecture, Front Controller and DispatcherServlet.
Unit 7: Java Mail	Introduction to Java API, Using Java Mail API to send mail using Java Codes, Sending Text Mail, Sending HTML Mail, Sending Mail with Attachments.
Unit 8: Java with JSON	JSON Syntax, DataTypes, Objects, Arrays in JSON, JSON Library in Java, Encoding a JSON Object in Java, Decoding a JSON Object in Java, Publishing a Service using JSON in JSP.



Yashwantrao
Chavan
Maharashtra
Open University

CMP514
Advance
Java

Advance JAVA

Yashwantrao Chavan Maharashtra Open University

Dnyangangotri, Near Gangapur Dam

Nashik-422222

Yashwantrao Chavan Maharashtra Open University

Vice-Chancellor: Prof. E. Vayunandan

SCHOOL OF COMPUTER SCIENCE

Dr. Pramod Khandare Director School of Computer Science Y.C.M.Open University Nashik	Shri. Madhav Palshikar Associate Professor School of Computer Science Y.C.M.Open University Nashik	Dr. P.V. Suresh Director School of Computer and Information Sciences I.G.N.O.U. New Delhi
Dr. Pundlik Ghodke General Manager R&D, Force Motors Ltd. Pune.	Dr. Sahebrao Bagal Principal, Sapkal Engineering College Nashik	Dr. Madhavi Dharankar Associate Professor Department of Educational Technology ,S.N.D.T. Women's University, Mumbai
Dr. Urmila Shrawankar Associate Professor, Department of Computer Science and Engineering G.H. Raison College of Engineering ,Hingana Road, Nagpur	Dr. Hemant Rajguru Associate Professor, Academic Service Division Y.C.M.Open University Nashik	Shri. Ram Thakar Assistant Professor School Of Continuing Education Y.C.M.Open University Nashik
Mrs. Chetna Kamalskar Assistant Professor School of Science and Technology Y.C.M.Open University, Nashik	Smt. Shubhangi Desle Assistant Professor Student Service Division Y.C.M.Open University Nashik	

Writer/s	Editor	Co-ordinator	Director
1. Mr. Tushar Kute Assistant Professor, Researcher, Computer Science, MITU Skillologics, Pune	Mr. Amit brahmecha Assistant Professor, SNJB's Late Sau. K.B. Jain College of Engineering, Nashik	Ms. Monali R. Borade Academic Co-ordinator School of Computer Science, Y.C.M. Open University, Nashik	Dr. Pramod Khandare Director School of Computer Science, Y.C.M. Open University, Nashik
2. Mrs. Shilpa Mistry Centre Head/HOD/Lecturer Atharva Institute Of Information Technology, Mumbai			

Production

SYLLABUS

Unit No. & Name	Details	Counseling Sessions	Weightage
Unit 1: JDBC	JDBC Architecture, Overview of Drivers, DBC Driver Manager, Steps for accessing database using JDBC API, Statements Prepared, Statement Callable, Statement Scrollable and Updatable ResultSet, ResultSetMetaData and DatabaseMetaData, Working with Rowset Interface.	4	10
Unit 2: Servlet	Introduction To Java Servlets, Servlet API, Servlet Life- Cycle, Working With Apache Tomcat, GenericServletsHttpServlet, HttpSession, Session Binding/Tracking, Inter-Servlet Communication.	4	10
Unit 3: JSP	JSP SYNTAX, Page Directive, Include Directive, Data Declaration and Method Definition, ScriptletsImplicit Objects, Custom Tags, Session Tracking in JSP, Page Context, Exception	3	10
Unit 4: Hibernate	Why Hibernate?, Understanding ORM, Objects and Persistence, Hibernate Architecture, Mapping Documents, Hibernate Database Connection, Creating Persistent Classes, Mapping Collection of Objects, Persistent Object Life Cycle, Hibernate with Servlets, HQL: Hibernate Query Language.	4	10
Unit 5: Spring Core	Introduction to Spring Framework, Inversion of Control and Dependency Injection, IOC Container, Bean Creation, Construction Injection, Setter Injection,	4	10
Unit 6: Spring MVC	Spring Web MVC, MVC Architecture, Front Controller and DispatcherServlet.	4	10
Unit 7: Java Mail	Introduction to Java API, Using Java Mail API to send mail using Java Codes, Sending Text Mail, Sending HTML Mail, Sending Mail with Attachments.	2	10
Unit 8: Java with JSON	JSON Syntax, DataTypes, Objects, Arrays in JSON, JSON Library in Java, Encoding a JSON Object in Java, Decoding a JSON Object in Java, Publishing a Service using JSON in JSP.	2	10
	Revision and Practice	3	
		30	80

Books and References:			
Sr. No.	Title	Author/s	Publisher
1.	Jdbc, Servlets, and Jsp Black Book (New Edition)	Kogent Solutions Inc. Santosh Kumar K.	Dreamtech Press
2.	Head First Servlets and JSP, 2nd Edition	Bert Bates, Bryan Basham, Kathy Sierra	O'Reilly
3	Just Hibernate	Madhusudhan Konda	O'Reilly
4	Getting Started with Spring Framework	J Sharma, Ashish Sarin	

UNIT 1–JDBC

Learning Objectives

- In this unit you will learn the fundamental aspects of JDBC and get a clear understanding of how it operates.
- After completing this chapter you will be able to create database connected Java applications.

1.1 INTRODUCTION

A **database** is group of related and rganized data. There are many different ways for organizing data to make it easy to access and manipulate. A **database management system (DBMS)** provides mechanisms for **storing, organizing, retrieving and modifying data** from any user. **Database management systemsallow for the access and storage of data without concern for the internal representation of data.**

Today's most popular database systems are **relational databases**. A language called **SQL**—is the international standard query language used almost universally with relational databases to perform **queries** and to manipulate data.

Some popular **relational database management systems (RDBMSs)** are **Microsoft SQL Server, Oracle,Sybase, MySQL,etc.**The JDK now comes with a pure-Java RDBMS called **Java DB**—Oracles's version of Apache Derby.

Java programs allows the programmers to interact with databases and manipulate their data using the **Java Database Connectivity (JDBC) API**.

A **JDBC driver** permits to connect to a database through Java applications in aparticular DBMS and allows you to manipulate that database using the JDBC API.

1.2 JDBC Introduction

The **JDBC API** is a Java API that can access data stored in a Relational Database or allows to access any kind of tabular data.

JDBC helps to write Java applications that manage these programming activities:

- Connect to a data source, like a database
- Send queries
- update or modify the database
- delete records
- Answer to your query by retrieving and processing the results received from the database

1.3 JDBC Components:

1.3.1 The JDBC API — The JDBC API provides programmatic access to relational data from the Javaprogramming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and transmit changes back to the underlying data source. The JDBC API is also able to communicate with various data sources in a distributed, heterogeneous environment. The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java

Enterprise Edition (Java2EE). The packages **java.sql** and **javax.sql** are sub package of **JDBC 4.0 API**. And both packages are encompassed in the JavaSE and Java EE platforms.

1.3.2 JDBC Driver Manager —The JDBC DriverManager class defines objects which can connect Javaapplications to a JDBC driver. DriverManager has usually been the backbone of the JDBC architecture. It is quite small and simple.

1.3.3 JDBC Test Suite —It is used to ensure that JDBC drivers will run yourprogram. These tests do exercise many of the important features in the JDBC API, but these tests are not comprehensive or exhaustive.

1.3.4 JDBC-ODBC Bridge —The Java Software bridge provides JDBC access via ODBC drivers. remember that ODBC binary code has to be load into each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

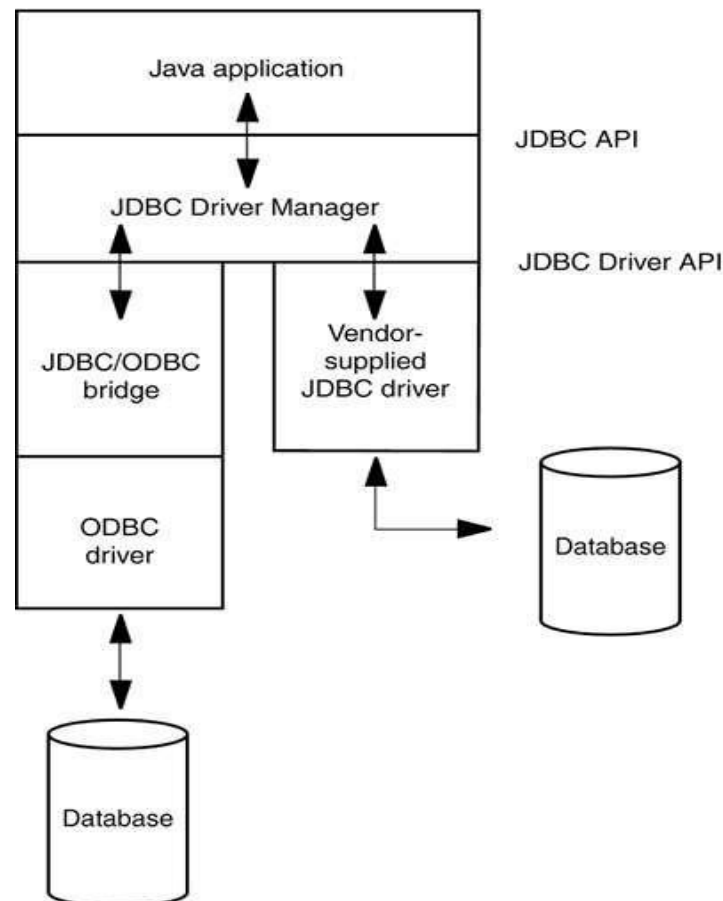


Figure 1.1.JDBC-to-database communication path

1.3.5 JDBC Driver Types

JDBC drivers are divided into the different types as mentioned below:

- **A type 1 driver** these drivers relies on an ODBC driver to communicate with the database and translates JDBC to ODBC. One of this type of driver included by sun is the JDBC/ODBC bridge, with the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver.

- A **type 2 driver** is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code in addition to a Java library.
- A **type 3 driver** is a pure Java client library which translates the requests into a database-specific protocol these drivers uses a database-independent protocol to communicate database requests to a server component; this can simplify deployment since the database-dependent code is located only on the server.
- A **type 4 driver** is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Now days maximum number of database vendors offers either a type 3 or type 4 driver with their database. As well as most of third party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

JDBC is used to for the following:

Programmers can write applications in the Java programming language to access any database, using standard SQL statements or even specialized extensions of SQL while still following Java language conventions.

Database vendors and database tool vendors optimize their drivers for their specific products, to maintain the quality of drivers.

1.4 1.4 Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (figure below). In this model, clients has inbuilt JDBC driver is deployed on it.

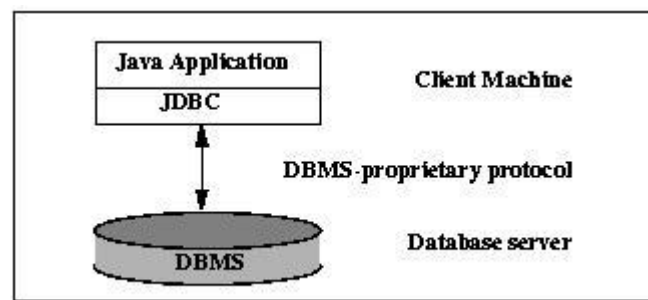


Figure 1.2. Two-tier Architecture for Data Access.

Now the world is adopting a "three-tier model" or even more advanced "n-tier models and is moving away from traditional client/server model." In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a various benefits and advantages as compared to traditional client/server model. It is responsible to separate visual presentation from the business logic in the middle tier and the raw data in the database. Which makes imaginable to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client), RMI (when you use an application), or another mechanism. JDBC manages the

communication between the middle tier and the back-end database. Figure below shows the basic three tier architecture.

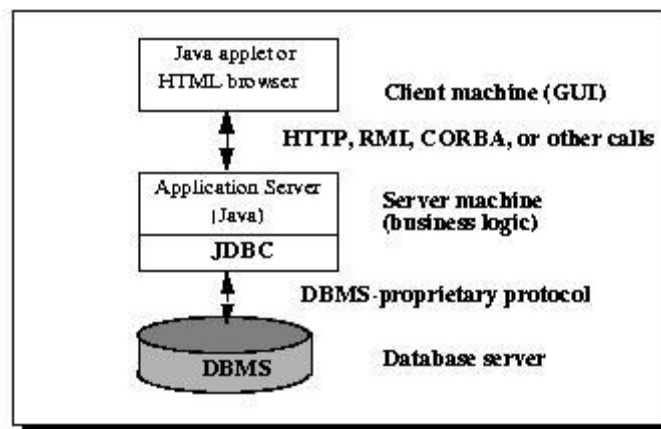


Figure1.3: Three-tier Architecture for Data Access

1.5 Database connections in Java using JDBC

Basic steps to use a database in Java are:

- 1.Establish a connection**
- 2.Create JDBC Statements**
- 3.Execute SQL Statements**
- 4.GET ResultSet**
- 5.Close connections**

1.5.1 Establishing a connection - An object is able to manage the connection between the Java program and the database when this object implements the Connection interface. The object created using Connection interface enables programs to manipulate databases by creating SQL statements. The connection can be initialized in the program by calling the static method `getConnection` of class `DriverManager` from `java.sql` package, which allows to connect to the database specified by its URL.

Method `getConnection` takes three arguments—a String that specifies the database URL, a String that specifies the username and a String that specifies the password.

e.g.

```

import java.sql.*;

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection conn = null;

conn = DriverManager.getConnection("jdbc:odbc:mydata","","");

// conn now has a live connection to the database
  
```

1.5.2.Create JDBC Statements

To execute the different types of SQL queries different interfaces are offered by JDBC API.

They are:

1) Statement – The Statement is first interface used to execute normal SQL queries. The statement can execute the predefined queries only, that is at runtime queries cannot be executed or cannot be passed as a parameter to the Sql statement. This interface can be used only if you need to execute a particular SQL query only once. Most of the times it is used with DDL statements like CREATE,ALTER,DROP etc.

e.g.

```
//Creating The Statement Object
```

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

```
//Executing The Statement
```

```
stmt.executeUpdate("CREATE TABLE STUDENT(ID NUMBER NOT NULL, NAME  
VARCHAR)");
```

2) PreparedStatement–To execute dynamic or parameterized SQL queries the Prepared Statement is used. This interface allows to pass the parameters to SQL query at runtime. It is better to use PreparedStatement if you are executing a particular SQL query multiple times.

e.g

```
//Creating PreparedStatement object
```

```
PreparedStatement pstmt = conn.prepareStatement("update STUDENT set NAME = ? where ID = ?");
```

```
//Setting values to place holders using setter methods of PreparedStatement object
```

```
pstmt.setString(1, "MyName"); //Assigns "MyName" to first place holder
```

```
pstmt.setInt(2, 111); //Assigns "111" to second place holder
```

```
//Executing PreparedStatement
```

```
pstmt.executeUpdate();
```

3) CallableStatement– CallableStatement is used to execute stored procedures. CallableStatement allows to pass three different types of parameters to stored procedures. The first parameter is IN which allows to pass the values to stored procedure, The second is OUT – Which hold the result returned by stored procedure and IN OUT – acts as both IN and OUT parameter.

e.g

```
//Creating CallableStatement object
```

```
CallableStatement cstmt = conn.prepareCall("{ call anyProcedure(?, ?, ?)}");
```

```
Use cstmt.setter() methods to pass IN parameters
```

```
//Use cstmt.registerOutParameter() method to register OUT parameters
```

```
//Executing the CallableStatement
```

```
cstmt.execute();
```

```
//Use cstmt.getter() methods to retrieve the result returned by the stored procedure
```

Table 1.1: Difference of SQL queries

Statement	PreparedStatement	CallableStatement
It is used to execute normal SQL queries.	It is used to execute parameterized or dynamic SQL queries.	It is used to call the stored procedures.
It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

Table 1.2: Mapping types JDBC – Java

JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	
BINARY	byte[]
VARBINARY	
LONGVARBINARY	
CHAR	String
VARCHAR	
LONGVARCHAR	

JDBC Type	Java Type
NUMERIC	BigDecimal
DECIMAL	
DATE	java.sql.Date
TIME	java.sql.Timestamp
TIMESTAMP	
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

*SQL3 data type supported in JDBC 2.0

1.5.3 Retrieving and Modifying Values from Result Sets

A **ResultSet** object is usually generated by executing a statement that queries the database which is a table of data representing a database result set. Any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet can create a ResultSet object.

Cursor Allows to access the data in a ResultSet object. Remember and note that this cursor is different than the database cursor. This cursor act as a pointer that can points to single row of data in the ResultSet. Cursor is set to the before the first row position initially. To moves the cursor to the next row the method **ResultSet.next()** is used. If the cursor is positioned after the last row, then this method returns false. To iterate through all the data in the ResultSet, the ResultSet.next method can be repeatedly calls using a while loop.

1.5.3.1 ResultSet Interface

For retrieving and manipulating the results of executed queries the ResultSet interface provides methods, and it's objects can have different functionality and characteristics such as type, concurrency, and cursor holdability.

ResultSet Types

The level of its functionality is determined in two areas by the type of a ResultSet object: the first is how simultaneous variations made to the primary data source are reflected by the ResultSet object and the another is the ways in which the cursor can be manipulated.

One of three different ResultSet type determines the sensitivity of a ResultSet object:

(a)TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows than can be the part of in the result set depend on how the results are generated by underlying database. In other words, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

(b)TYPE_SCROLL_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. The rows that satisfy the query at either the time the query is executed or as the rows are retrieved are contained in it.

(c)TYPE_SCROLL_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

You should remember and note that not all databases and JDBC drivers support all ResultSet types. if the specified ResultSet type is supported then the method DatabaseMetaData.supportsResultSet Type returns true otherwise it returns false.

1.5.3.2 ResultSet Concurrency

The level of update functionality supported is determined by concurrency of a ResultSet object

There are two concurrency levels:

CONCUR_READ_ONLY: The ResultSet interface cannot update the ResultSetobject .

CONCUR_UPDATABLE: The ResultSet interface can update the ResultSet object

The default ResultSet concurrency is CONCUR_READ_ONLY.

You should remember and note that concurrency is not supported by all JDBC drivers and databases. If the specified concurrency level is supported by the driver the method DatabaseMetaData.supportsResultSetConcurrency returns true and false otherwise.

1.5.3.3 Cursor Holdability

To ResultSet objects that have been created during the current transaction can be closed by calling the method Connection.commit. However, this may not be the desired behavior in some cases. When commit is called The ResultSet property holdability gives the application control over whether ResultSet objects are closed.

The following are some of the ResultSet constants that may be supplied to the Connection methods prepareStatement, createStatement, and prepareCall:

HOLD_CURSORS_OVER_COMMIT: ResultSet cursors are holdable they are not closed; when the method commit is called, they are held open. If your application uses mostly read-only ResultSet objects, then holdable cursors might be ideal.

CLOSE_CURSORS_AT_COMMIT: As soon as the commit method is called the ResultSet objects (cursors) are get closed. It can result in better performance for some applications when cursors are closed by calling this method.

The default cursor hold ability varies depending on your DBMS.

1.5.3.4 Retrieving Column Values from Rows

For retrieving column values from the current row the ResultSet interface declares getter methods such as, getBoolean and getLong. The values can be retrieved by using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. In order to achieve maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

```
try
{
    create Statement for querying database statement = connection.createStatement();
    query database
    resultSet = statement.executeQuery(
    "SELECT AuthorID, FirstName, LastName FROM authors" );
    // process query results
    ResultSetMetaData metaData = resultSet.getMetaData(); int numberOfColumns =
    metaData.getColumnCount(); System.out.println( "Authors Table of Books Database:\n" );

    for ( inti = 1; i<= numberOfColumns; i++ )
    System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
    System.out.println();
    while ( resultSet.next() )
    {
        int id = rs.getInt("AuthorID");
        String firstName = rs.getString("FirstName");
        String lastName = rs.getString("LastName");

        System.out.println(id+ "\t" + firstName+
        "\t" + lastName );
    } //end while
} // end try

catch ( SQLException sqlException )
{
    sqlException.printStackTrace();
} // end catch
```

1.6 CURSORS

Cursors allow you access the data in a **ResultSet object**, which points to one row in the ResultSet object. However, when a ResultSet object is first created, the cursor is positioned before the first row. There are other methods available to move the cursor:

next: Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.

previous: Moves the cursor backward one row. Returns true if the cursor is positioned before last row and false if the cursor is positioned before the first row.

first: This method is used to move the cursor to the first row in the ResultSet object. If the cursor is pointing to the first row of ResultSet then it returns true and will return false if the ResultSet object does not contain any rows.

last: This function will be used to migrate the cursor to the last row in the ResultSet object. If the cursor is pointing to the last row of ResultSet then it returns true and will return false if the ResultSet object does not contain any rows.

beforeFirst: This method will locate the cursor at the start of the ResultSet object, before the first row. This method has no effect, if the ResultSet object does not contain any rows.

afterLast: This function is used to shift or migrate the cursor from current position to at the end of the ResultSet object, after the last row. This method has no effect, if the ResultSet object does not contain any rows. **relative(int rows):** Moves the cursor relative to its current position.

absolute(int row): Positions the cursor on the row specified by the parameter row.

The ResultSet has default sensitivity which is `TYPE_FORWARD_ONLY`, which does not allow it to scroll; you cannot call any of these methods that move the cursor, except next, if your ResultSet cannot be scrolled.

1.6.1 Updating Rows in ResultSet Objects

The default ResultSet object cannot be modified, and it can only move its cursor in forward direction only. But ResultSet objects can be created and that can be scrolled, but note that the cursor can move backwards or move to an absolute position and updated.

```
try {  
  
    establish connection to database connection = DriverManager.getConnection( DATABASE_URL,  
    "root", "" );  
  
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
  
    ResultSet uprs= statement.executeQuery("SELECT * FROM authors");  
  
    while (uprs.next()) {  
        uprs.updateString( "LastName","Sharma");  
  
        uprs.updateRow();  
    }  
} catch ( SQLException sqlException )  
{  
    sqlException.printStackTrace();  
} // end catch
```

A ResultSet object which is created by The field **ResultSet.TYPE_SCROLL_SENSITIVE**, the cursor of this object can move both backward and forward relative to the current position and to an

absolute position. The field **ResultSet.CONCUR_UPDATABLE** creates a ResultSet object that can be updated. There are some more other fields in ResultSet that can be used to modify the behavior of ResultSet objects.

To update the specified column in a ResultSet the method **ResultSet.updateString** is used. Let us consider with above mentioned example, LastName is modified with the specified float value in the row where the cursor is positioned. ResultSet has different version of update methods and with different parameters that allows to update column values of various data types. However, none of these update methods modifies the database; you must call the method **ResultSet.updateRow** to update the database.

1.6.2 Inserting Rows in ResultSet Objects

Inserting new rows with the ResultSet interface is not supported by all JDBC drivers, a **SQLFeatureNotSupportedException** exception is thrown, if any one attempt to insert a new row and in case JDBC driver database used are not supporting to this feature.

```
try {  
  
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
  
    ResultSetuprs = statement.executeQuery(  
    "SELECT * FROM authors");  
  
    uprs.moveToInsertRow();  
  
    uprs.updateInt("AuthorID",9);  
    uprs.updateString("FirstName","Subash");  
    uprs.updateString("LastName","Pakhrin");  
    uprs.insertRow();  
    uprs.beforeFirst();  
}  
  
catch ( SQLExceptionsqlException )  
{  
    sqlException.printStackTrace();  
} // end catch
```

In the example mentioned above we can understand the calling of **Connection.createStatement** method with two arguments, **ResultSet.TYPE_SCROLL_SENSITIVE** and **ResultSet.CONCUR_UPDATABLE**. The first parameters allow the cursor of the ResultSet object to be moved both forward and backward. The second parameter which is **ResultSet.CONCUR_UPDATABLE** allows to insert rows into a ResultSet object; and it specifies that it can be updatable.

The same conditions for using strings in getter methods also apply to update methods.

The **ResultSet.moveToInsertRow** method is used to migrate the cursor to the insert row. The insert row is associated with an updatable result set and is treated as a special row. It is essentially a buffer where a new row can be constructed by calling the update methods prior to inserting the row into the result set. Let us consider and understand it with the example mentioned above in which the method **ResultSet.updateString** is used to update the insert row's **FirstName** column to Subash. To insert the contents of the insert row into the ResultSet object the method **ResultSet.insertRow** is used and which can also insert it into the database.

Once the new row is inserted using the `ResultSet.insertRow` method, then the cursor has to be moved to a row other than the insert row. Let us consider and understand it with the example mentioned above in which the method `ResultSet.beforeFirst` is used to move it to before the first row in the result set. If another part of your application uses the same result set and the cursor is still pointing to the insert row then unexpected results can occur.

1.6.3 Using Statement Objects for Batch Updates

`Statement`, `PreparedStatement` and `CallableStatement` objects have a list of commands that is associated with them. The statements for updating, inserting, or deleting a row may be contained in this list; as well as it may also contain DDL statements such as `CREATE TABLE` and `DROP TABLE`. But it cannot contain a `SELECT` statement that can produce a `ResultSet` object. So here we can conclude that, only statements that produce an update count can be contained in the list.

The list, which is associated with a `Statement` object at its creation, is initially empty. The method `addBatch` can be used to add SQL commands to this list and empty it with the method `clearBatch`. The method `executeBatch` can be called to send them all to the database to be executed as a unit, or batch, once you have finished adding statements to the list.

```
try {
    connection = DriverManager.getConnection(
        DATABASE_URL, "root", "" );
    connection.setAutoCommit(false);
    statement = connection.createStatement();

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('15','Hari','Shrestha')");

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('16','Ram','Acharya')");

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('17','Shyam','Gautam')");

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('18','Govinda','Paudel')");

    int [] updateCounts = statement.executeBatch();
    connection.commit();

}
catch (BatchUpdateException
    n b) { b.printStackTrace();
}
catch (SQLException
    n ex) {
    ex.printStackTrace(
    );
}
```


The following line disables auto-commit mode for the Connection object conn so that the **transaction** will not be automatically committed or rolled back when the method executeBatch is called.

connection.setAutoCommit(false);

One should always disable auto-commit mode before beginning a batch update, which will help and allow for correct error handling.

The method **Statement.addBatch** is used to add a command to the list of commands associated with the Statement object statement. Let us understand and observe this in example mentioned above in which, these commands are all INSERT INTO statements. In which each command is adding a row consisting of three column values.

Let us understand and observe this in example mentioned below in which the mentioned sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

int [] updateCounts = statement.executeBatch();

The **executeBatch** method is used by Statement to send the batch of insertions, **not the method executeUpdate, which sends only one command and returns a single update count**. The commands in the DBMS are executed in the order in which they were added to the list of commands, so it will first add the row of values for "Hari", and then one by one add the row for various record let us consider for "Amit", then "Rama", and finally "Govinda". The DBMS will return an update count for each command in the order in which it was executed if all four commands execute successfully. The number of rows affected by each command are stored in array indicated updateCounts.

If all four of the commands in the batch are executed successfully, updateCount will contain four values, all of which are 1 because an insertion affects one row. When the commands added previously were sent to the database and when statement called the method executeBatch, then the list of commands associated with statement will get empty. The method clearBatch can be called any time to explicitly empty this list at any time.

The Connection.commit method makes the batch of updates to the "authors" table permanent. As the auto-commit mode for this connection was disabled, this method needs to be called explicitly.

To enable auto-commit mode for the current Connection object you have to call the command as mentioned below.

connection.setAutoCommit(true);

This command will make every statement automatically be committed once it is executed, and it no longer needs to invoke the method commit.

1.7 TRANSACTION PROCESSING

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. Let us consider with an example, there are several factors determine if the transaction is successful when you transfer money electronically between bank accounts. In this case we expect the amount you wish to transfer from source account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the

money in the source account. The way to be sure that either both actions occur or neither action occurs is to use a **transaction**. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.

The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode**.

1.7.1 Disabling Auto-Commit Mode

When a connection is created, it is in auto-commit mode. It indicates that after it is executed each individual SQL statement is treated as a transaction and is automatically committed.

By disabling the auto-commit mode one can enable to allow two or more statements to be **grouped into a transaction**.

```
conn.setAutoCommit(false);
```

1.7.2 Committing Transactions

No SQL statements are committed after the auto-commit mode is disabled, until you call the **method commit explicitly**. All statements executed after the previous call to the method commit are included in the **current transaction and committed together as a unit**. `conn.commit();`

1.7.3 Rollback

If all the update statements are grouped together to a transaction, then the transaction either succeeds in its entirety and it can be committed, or it fails somewhere in the middle. In that case, you can carry out a **rollback** and the database automatically undo the effect of all updates that occurred since the last committed transaction.

You turn off autocommit mode with the command

```
conn.setAutoCommit(false);
```

Now you create a statement object in the normal way:

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

Call executeUpdate any number of times:

```
stat.executeUpdate(command1);
```

```
stat.executeUpdate(command2);
```

```
stat.executeUpdate(command3);
```

...

When all commands have been executed, call the commit method:

```
conn.commit();
```

However, if an error occurred, call

```
conn.rollback();
```

Then, all commands until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a `SQLException`.

1.8 ROWSET INTERFACE

A JDBC RowSet object holds tabular data efficiently which will allow it more flexible and easier to use than a result set. The RowSet interface configures the database connection and prepares query statements automatically. To specify the properties needed to establish a connection it provides several **set methods**. For example the database URL, user name and password of the database and create a Statement such as a query. The several **get methods are also provided by RowSet** that return these properties.

1.8.1 Connected and Disconnected RowSets

The RowSet objects are basically of two types which are **connected and disconnected**. To connect to the database once a **connected RowSet** object is used and it remains connected while the object is in use. A **disconnected RowSet** object connects to the database and instantly it closes the connections when, it executes a query to retrieve the data from the database. The data in a disconnected RowSet

may be modified by the Program while it's disconnected. After a disconnected RowSet reestablishes the connection with the database, Modified data can be updated in the database.

Two sub-interfaces of RowSet—**JdbcRowSet** and **CachedRowSet** are contained in a package `Package javax.sql.rowset`. a connected RowSet, acts as a wrapper around a ResultSet object and allows you to scroll through and update the rows in the ResultSet. A ResultSet object set to nonscrollable by default and is read only and it must explicitly set the result set type constant to `TYPE_SCROLL_INSENSITIVE`. To `CONCUR_UPDATABLE` to make a ResultSet object scrollable and updatable set the result set concurrency constant.

A JdbcRowSet object is scrollable and updatable by default. RowSet, caches the data of a ResultSet in memory and disconnects from the database. Like JdbcRowSet, a CachedRowSet object is scrollable and updatable by default. A Cached-RowSet object is also serializable, so it can be passed between Java applications through a network, such as the Internet. However, CachedRowSet has a limitation—the amount of data that can be stored in memory is limited. Package `javax.sql.rowset` contains three other sub-interfaces of RowSet: **WebRowSet**, **JoinRowSet** and **FilteredRowSet**.

JdbcRowSet

Navigating JdbcRowSet Objects

```
JdbcRowSet jdbcRs = new JdbcRowSetImpl();  
jdbcRs.absolute(4);  
jdbcRs.previous();
```

Updating Column Values

```
jdbcRs.absolute(3);  
jdbcRs.updateString("lastName", "Sharma");  
jdbcRs.updateRow();
```

Inserting Rows

```
jdbcRs.moveToInsertRow();  
jdbcRs.updateInt("Author_ID", 10);  
jdbcRs.updateString("FirstName", "Navin");  
jdbcRs.updateString("LastName", "Sharma");  
jdbcRs.insertRow();
```

Deleting Rows

```
jdbcRs.last();  
jdbcRs.deleteRow();
```

Important Questions

1. What is RowSet Interface?
 2. Explain the types of RowSet Interface.
 3. Write a short note on Transaction processing.
 4. Write a short note JDBC API.
 5. List and explain various JDBC Components.
 6. Explain the various types of JDBC Driver.
 7. Write a short note on Cursors.
 8. Explain the different interfaces offered by JDBC API to execute SQL Queries.
 9. Explain different architectures of JDBC to Access the data.
-

UNIT 2 SERVLETS

Learning Objectives

After going through this unit you will be able to explain.

- What is a Servlet?
- The various Servlet API
- What is the Life-cycle of a Servlet?
- How to run a servlet in Apache Tomcat Server?
- Types of servlets. Generic and HTTP
- What is a session?
- How to bind a Session?
- Session tracking.
- How servlets of a web application communicate with each other?

2.1 What is a Java Servlet?

Servlets are Java programs that can be executed dynamically from a Web Server. They can be executed only on server and hence they are called as a Server-side technology. A Servlet is an intermediating layer between an HTTP request of a client and the Web server.

For example, a Servlet might be responsible for taking data in an HTML order-entry form and applying the business logic used to update a company's order database

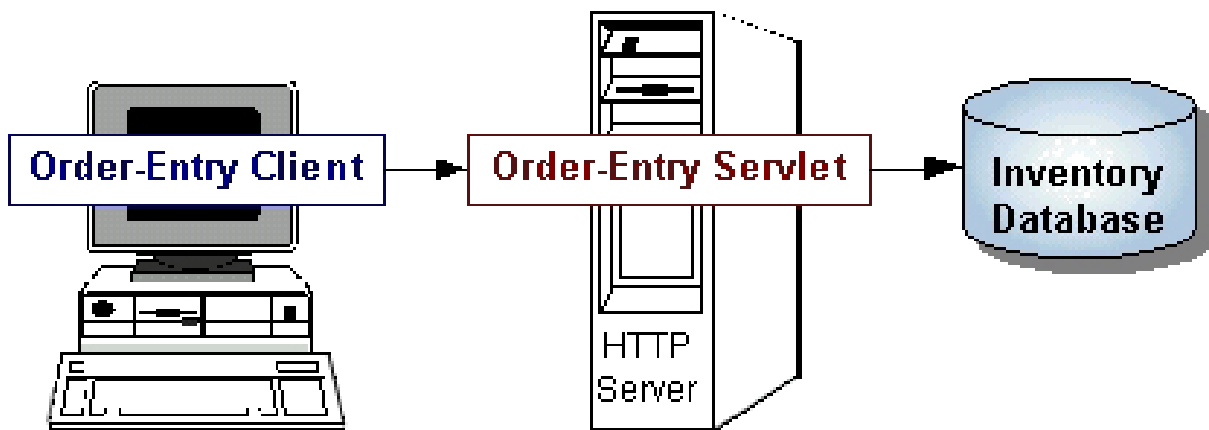


Figure 2.1 Working of Java Servlets

2.2 Servlet API

The Servlet API is made up of two packages. These packages contain the classes and interfaces required to build servlets. They are **javax.servlet** and **javax.servlet.http**. These packages are not part of the Java core packages. Instead, they are standard extensions. Hence, these packages are not

contained in the Java Software Development Kit. You must download Tomcat or Glass Fish server to obtain their functionality.

2.2.1 The javax.servlet Package

The **javax.servlet package** having a number of classes and interfaces that operate the servlets by establishing the framework.

The following table lists the **core interfaces** that are provided in this package. The most important is **Servlet**. All servlets you create must implement this interface or extend to a class that implements this interface.

The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	It is used to Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	It is Used to read data from a client request.
ServletResponse	Used to write data to a client response.
SingleThreadModel	It Describe thatthat the servlet is thread safe.

The following table summarizes the **core classes** that are provided in the javax.servlet package.

Class	Description
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

2.2.2 The Servlet Interface

All servlets must implement the **Servlet interface**. This interface declares the various methods such as **init()**, **service()**, and **destroy()** methods that are called by the server during the life cycle of a servlet. The methods defined by Servletare shown below:

Table 2.1 Methods of servlet interface

Method Summary	
void	<u>destroy()</u> Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
<u>ServletConfig</u>	<u>getServletConfig()</u> This method is used to returns a <u>ServletConfig</u> object, This methods accepts parameters for initialization and startup for this servlet.

Java.lang.String	<u>getServletInfo()</u> This method can be used to get the various information about the servlet, such as author, version, and copyright.
Void	<u>init(ServletConfig config)</u> Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
Void	<u>service(ServletRequest req, ServletResponse res)</u> This method is used to allow the servlet to respond to a request and Called by the servlet container.

2.2.3 The ServletRequest Interface

The ServletRequest interface is implemented by the server. It permits a servlet to acquire information about a client request. Several of its methods are summarized in Table below.

Table 2.2 Methods of ServletRequest interface

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value of the attribute named <i>attr</i> .
String getCharacterEncoding()	Returns the character encoding of the request.
int getContentLength()	Returns the size of the request. The value -1 is returned if the size is unavailable.
String getContentType()	Returns the type of the request. A null value is returned if the type cannot be determined.
ServletInputStream getInputStream() throws IOException	Returns a ServletInputStream that can be used to read binary data from the request. An IllegalStateException is thrown if getReader() has already been invoked for this request.
String getParameter(String <i>pname</i>)	Returns the value of the parameter named <i>pname</i> .
Enumeration getParameterNames()	Returns an enumeration of the parameter names for this request.
String[] getParameterValues(String <i>name</i>)	Returns an array containing values associated with the parameter specified by <i>name</i> .
String getProtocol()	Returns a description of the protocol.
BufferedReader getReader() throws IOException	Returns a buffered reader that can be used to read text from the request. An IllegalStateException is thrown if getInputStream() has already been invoked for this request.
String getRemoteAddr()	Returns the string equivalent of the client IP address.
String getRemoteHost()	Returns the string equivalent of the client host name.
String getScheme()	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName()	Returns the name of the server.
int getServerPort()	Returns the port number.

2.2.4 The ServletResponse Interface

The ServletResponse interface is implemented by the server. It enables a servlet to formulate a response for a client. Several of its methods are summarized in Table below.

Table 2.3 Methods of ServletResponseinterface

Method	Description
String getCharacterEncoding()	Returns the character encoding for the response.
ServletOutputStream getOutputStream() throws IOException	Returns a ServletOutputStream that can be used to write binary data to the response. An IllegalStateException is thrown if getWriter() has already been invoked for this request.
PrintWriter getWriter() throws IOException	Returns a PrintWriter that can be used to write character data to the response. An IllegalStateException is thrown if getOutputStream() has already been invoked for this request.
void setContentLength(int size)	Sets the content length for the response to <i>size</i> .
void setContentType(String type)	Sets the content type for the response to <i>type</i> .

2.3 Servlet Life-Cycle

The life-cycle of a servlet is the entire process or steps from its creation till its destruction. The entire process is maintained by the Servlet Container.

A servlet's life cycle is managed via the `init()`, `service()` and `destroy()` methods.

The diagram shown below describe the detailed life cycle of Servlet.

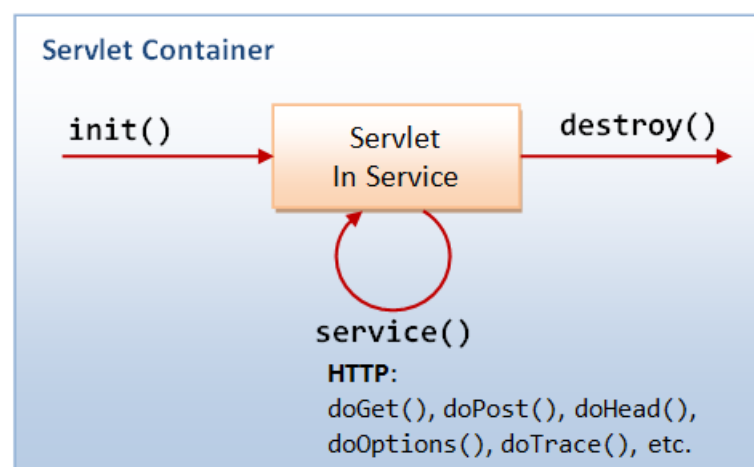


Figure 2.2 Servlet Life Cycle

The steps of the life cycle of the servlet are:

1. The server loads the Servlet class and initializes one instance of it
2. Servlet instance is created. The Servlet instance in a separate thread handles each client request.
3. `init()` method is invoked.
4. `service()` method is invoked.
5. `destroy ()` method is invoked.

Step 1-3: Loading and Initialization

For loading and instantiating the servlet, Servlet container such as Tomcat or Glassfish are responsible. It may load and instantiate servlets when it is started, or delay until it determines that the servlet is needed to service a request (usually at the first request to the servlet).

The servlet container is responsible to invoke the `init(ServletConfig)` method of the servlet, which is providing a `ServletConfig` object as an argument. `init()` runs only once. It is generally used to initialize costly resource and to read persistent configuration data.

The servlet access *initialization parameters* for this particular servlet with the permission of this `ServletConfig` object.

These parameters are defined in the *web application deployment descriptor file* (i.e., “web.xml”), under the servlet's name, as follows:

```
<servlet>
<servlet-name>ServletName</servlet-name>
<servlet-class>ServletClassFile</servlet-class>
<init-param>
<param-name>initParam1</param-name>
<param-value>initParam1 Value</param-value>
</init-param>
<init-param>
<param-name>initParam2</param-name>
<param-value>initParam2 Value</param-value>
</init-param>
</servlet>
```

These methods are defined by **the ServletConfig interface** to retrieve the initialization parameters for this servlet.

```
String getInitParameter(String name)
java.util.EnumerationgetInitParameterNames()
```

`HTTPServlet` and `GenericServlet` both classes implement the interface `ServletConfig`. That's why, the `getInitParameterNames()` and `getInitParameter()` methods can be invoked directly within `init()` or `service()`.

The `ServletConfig` grant the servlet access to a `ServletContext` object that offers information about this web context (aka web application).

Step 4: In Service

The servlet container invokes its `service()` method once a servlet is initialized to handle client requests. For each request this method is called only once. To the same servlet by running `service()` on different threads the servlet container handle concurrent request unless and until `SingleThreadModel` interface is declared.

For HttpServlet, service() dispatches doGet(), doPost(), doHead(), doOptions(), doTrace(), etc, to handle HTTP GET, POST, HEAD, OPTIONS, TRACE, etc, request respectively.

The service() method of an HttpServlet takes two arguments, an HttpServletRequest object and an HttpServletResponse object that corresponds to the HTTP request and response messages respectively.

Step 5: End of Service

When it is decided by servlet container that a servlet should be detached from the container such as in case of shutting down the container or time-out, which is implementation-dependent, it calls the destroy() method to release any resource it is using and save any persistent state. It must allow all service() threads to complete or time-out before the servlet container calls the destroy().

2.4 Working with Apache Tomcat HTTP Server

Apache Tomcat is a Java-capable HTTP server, which can execute special Java web programs known as "Java Servlets" and "Java Server Pages (JSP)". Tomcat is an open-source project, under the "Apache Software Foundation". The mother site for Tomcat is <http://tomcat.apache.org>. Tomcat was originally written by James Duncan Davidson (then working in Sun) in 1998, based on an earlier Sun's server called Java Web Server (JWS). Sun subsequently made Tomcat open-source and gave it to Apache.

Tomcat Server is a container for Java Servlets. To run a Servlet the Apache Tomcat web server has to be downloaded from <http://tomcat.apache.org>.

2.4.1 Steps to run a Servlet in Apache Tomcat

1. Download Apache tomcat and save folder in C drive.
2. Right click on Computer from properties select advance tab and select environment variable.
3. Select new button and set variable CATALINA_HOME to C:\mywebproject\tomcat
4. Select new button and set variable JAVA_HOME to C:\Program Files\Java\jdk1.7.0_04
5. Click on OK and close property dialog.
6. Write servlet program in notepad and save it in the following path
C:\ mywebproject\tomcat \webapps\examples\WEB-INF\classes
7. Goto command prompt change directory to the above path
8. Set CLASSPATH=C:\Program Files\Java\jdk1.7.0_04\bin;C:\mywebproject\tomcat \bin;
C:\ mywebproject\tomcat\lib\servlet-api.jar;
9. set PATH=C:\Program Files\Java\jdk1.7.0_04\bin
10. Compile the servlet program by running javac.

e.g:

```
C:\ mywebproject\tomcat \webapps\examples\WEB-INF\classes>javac Login.java
```

11. Make changes to web.config file which is in the following path of Tomcat

```
C:\ mywebproject\tomcat \webapps\examples\WEB-INF
```

12. e.g

```
<servlet>
```

```

<servlet-name>Login</servlet-name>
<servlet-class>Login</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Login</servlet-name>
<url-pattern>/servlets/servlet/Login</url-pattern>
</servlet-mapping>

```

13. Start Tomcat server by selecting startup.exe from the following path
C:\mywebproject\ tomcat\bin

14. Start Internet Explorer and in the URL type
http://localhost:9999/

15. To view your servlet in the URL type
http://localhost:9999/examples/servlets/servlet/Login

2.5 GenericServlets

Generic Servlets make writing a Servlet easier. Simple versions of the init and destroy are provided, it also provides simple version ServletConfig interface methods. A log method, from the ServletContext interface is also provided. The written servlet should override only the service abstract method. The following diagram shows the hierarchy of the Servlet interface and classes.

Hierarchy of Servlet ,GenericServlet , HttpServlet

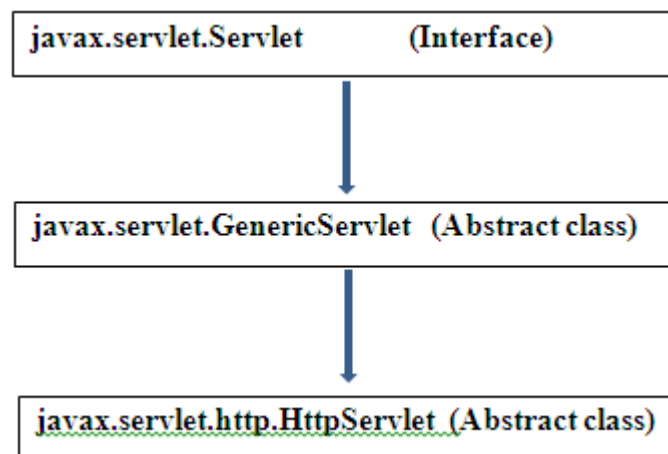


Figure 2.3 The hierarchy of the Servlet interface and classes

We can write a Servlet in three ways.

1. By implementing the Servlet interface.
2. By extending GenericServlet abstract class for protocol independent Servlet.
3. By extending HttpServlet abstract class.

Writing a Generic Servlets

As discussed a Generic servlet is a protocol-independent servlet that should always override the `service()` method to handle the client request. The `service()` method accepts two arguments, *ServletRequest object*, and *ServletResponse object*. The request object tells the servlet about the request made by the client while the response object is used to return a response back to the client. `GenericServlet` is an *abstract class* and it has only one abstract method, which is `service()`. In the following example we see how to create and invoke a Generic servlet.

We have to create 3 files :

1. HTML file

First we create an HTML file that will call the servlet once we click on the link on the web page.

```
<html>

<title>Generic Servlet Demo</title>
</head>
<body>
<a href="welcome">Click here for call to Generic Servlet</a>
</body>
</html>
```

2. Java Class file

Second we create a Java file `MyGeneric` by extending `GenericServlet` class. When creating a `GenericServlet`, you must override the `service()` method

```
import java.io.*;
import javax.servlet.*;

public class MyGeneric extends GenericServlet{
    public void service(ServletRequest req,ServletResponse res)
        throws IOException,ServletException
    {
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.print("<html>");
        out.print("<body>");
        out.print("<h1>Generic Servlet Example</h1>");
        out.print("Welcome to Advance Java Servlets programming");
        out.print("</body>");
        out.print("</html>");
    }
}
```

3. Change web.xml file

We make changes to the `web.xml` file to map to the Servlet with the specific URL.

web.xml

```
<web-app>
<servlet>
<servlet-name>MyGeneric</servlet-name>
<servlet-class>MyGeneric</servlet-class>
</servlet>
<servlet-mapping>
```

```
<servlet-name>MyGeneric</servlet-name>
<url-pattern>/1 MyGeneric</url-pattern>
</servlet-mapping>
</web-app>
```

2.6 HttpServlets

As shown in Figure 2.2 the HttpServlet class extends the GenericServlet class. It is an http protocol based servlet which implements the Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

To write the HttpServlet we have to override either the doGet or doPost method. The doGet method is overridden if the Servlet supports the HTTP Get request and doPost for HTTP POST request.

In the following example we see how to create and invoke an HttpServlet.

We have to create 3 files :

1. HTML file

First we create an HTML file that will call the servlet once we click on the link on the web page.

```
<html>
<head>

<title>Http Servlet Demo</title>
</head>
<body>
<a href="welcome">Click here for calling Http Servlet</a>
</body>
</html>
```

2. Java Class file

Second we create a Java file MyHttpServlet by extending HttpServlet class. When creating a HttpServlet, we will override the doGet() method

```
import java.io.*;
import javax.servlet.*;

public class MyHttpServlet extends HttpServlet{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.print("<html>");
        out.print("<body>");
        out.print("<h1>Generic Servlet Example</h1>");
        out.print("Welcome to Advance Java Servlets programming");
        out.print("</body>");
        out.print("</html>");
    }
}
```

3. Change web.xml file

We make changes to the web.xml file to map to the Servlet with the specific URL.

web.xml

```
<web-app>
<servlet>
<servlet-name>MyHttpServlet</servlet-name>
<servlet-class>MyHttpServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>MyHttpServlet</servlet-name>
<url-pattern>/1 MyHttpServlet</url-pattern>
</servlet-mapping>
</web-app>
```

2.7 HttpSession

2.7.1 Session in Servlets

The **interactive time** between client and server on a single connection is known as a session or the **period of time** between connection establishment and connection closing between client and server is known as a session. A Session begins when the client **logs in** to a Web site and ends when the user **logs out**.

A connection is **well maintained** by the Servlet container while the client and server are conversing back and onwards in a session.

2.7.2 HttpSession

The HttpSession object is used for session management. A session contains information specific to a particular user across the whole application. When a user enters into a website (or an online application) for the first time HttpSession is obtained via request.getSession(), the user is given a unique ID to identify his session. This unique ID can be stored into a cookie or in a request parameter. The HttpSession stays alive until it has not been used for more than the timeout value specified in tag in deployment descriptor file(web.xml). The default timeout value is 30 minutes, this is used if you don't specify the value in tag. This means that when the user doesn't visit web application time specified, the session is destroyed by servlet container. The subsequent request will not be served from this session anymore; the servlet container will create a new session.

HttpSession methods

public void setAttribute(String name, Object value): This method binds the object with a name and stores the name/value pair as an attribute of the HttpSession object. If an attribute already exists, then this method replaces the existing attributes.

public Object getAttribute(String name): This method returns the String object specified in the parameter, from the session object. If no object is found for the specified attribute, then the getAttribute() method returns null.

public Enumeration getAttributeNames(): This method returns an Enumeration that contains the name of all the objects that are bound as attributes to the session object.

public void removeAttribute(String name): This method removes the given attribute from session.

setMaxInactiveInterval(int interval): This method sets the session inactivity time in seconds. This is the time in seconds that specifies how long a sessions remains active since last request received from client.

2.8 Session Binding/Tracking

Session Tracking or Session Management

A session includes a **lot of interactions**, where data will be exchanged, between client and server, on a single connection. Once the server accepts the client connection, the client and server talk together and keep with them lot of data exchanged that includes commits, questions, answers etc. Preserving the data of a session so that the data can be reused later is known as session tracking or session management.

A session can temporarily store information related to the activities of the user while logged in. A servlet should be capable to store temporary information pertaining to the activities of the user in a session.

Session tracking is required many a times in Web communication, especially in e-commerce or online shopping (shopping cart). In online shopping, a client logs into an e-commerce web site and clicks many times the items he would like to buy. In between he may ask the server the details of a product, its price and any schemes available etc. The server accordingly responses and make a note of the items the buyer orders. Preserving the item names and quantity all over the session is a must to make a final bill before the client transfers money online. All this requires session tracking.

In the following session example 3 files are created – index.html, MyServlet1.java,

MyServlet2.java and changes are made to the web.xml file for servlet mapping

Session Example

index.html

```
<form action="login">
  User Name:<input type="text" name="userNm"/><br/>
  Password:<input type="password" name="userPsw"/><br/>
  <input type="submit" value="Submit"/>
</form>
```

MyServlet1.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet1 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            String name = request.getParameter("userNm");
            String password = request.getParameter("userPsw");
            out.print("Hello "+name);
            out.print("Your Password is: "+password);
            HttpSession session=request.getSession();
            session.setAttribute("uname",name);
            session.setAttribute("upass",password);
            out.print("<a href='welcome'>view details</a>");
            out.close();
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
}
```

```

    }
}
MyServlet2.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response){
    try{
response.setContentType("text/html");
PrintWriter out = response.getWriter();
HttpSession session=request.getSession(false);
    String myName=(String)session.getAttribute("uname");
    String myPass=(String)session.getAttribute("upass");
out.print("Name: "+myName+" Pass: "+myPass);
out.close();
    }catch(Exception ex){
System.out.println(ex);
    }
    }
}

```

```

web.xml
<web-app>
<servlet>
<servlet-name>Servlet1</servlet-name>
<servlet-class>MyServlet1</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Servlet1</servlet-name>
<url-pattern>/login</url-pattern>
</servlet-mapping>
<servlet>
<servlet-name>Servlet2</servlet-name>
<servlet-class>MyServlet2</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Servlet2</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>

```

2.9 Inter Servlet Communication

The communication between the servlets of a web application is known as Inter Servlet Communication. The uses of Inter Servlet communication are direct servlet manipulation, reusing servlets and servlet collaboration. There are various ways through which servlets communicate with each other; we are going to discuss Request Dispatcher method and Redirect methods.

Inter-servlet communication using Request Dispatcher

The RequestDispatcher object can **forward** a client's request to a resource or **include** the resource itself in the response back to the client. The resource can be another servlet, an HTML file, or a JSP file, etc.

There are two ways to delegate a request:- 1) Forward and 2) Include

Request Dispatcher:

An object of the `javax.servlet.RequestDispatcher` interface that allows inter-servlet communication. Object is used to include or forward the content of another servlet.

To get `RequestDispatcher` Object:

```
//for relative path  
ServletContext.getRequestDispatcher(String resource)
```

```
//for context path  
ServletRequest.getRequestDispatcher(String resource)
```

`RequestDispatcher` interface provides two methods:

```
public void forward(ServletRequest req, ServletResponse res) throws  
ServletException, java.io.IOException;
```

Forwards a request from a servlet to another resource on the server.

```
public void include(ServletRequest req, ServletResponse res) throws  
ServletException, java.io.IOException;
```

Includes the content of a resource in the response.

Examples:

```
RequestDispatcher rd;  
rd = request.getRequestDispatcher("xyz.jsp?user=fred");  
rd.include(request, response);  
Or  
RequestDispatcher dispatcher = req.getRequestDispatcher("/index.html");  
dispatcher.forward(req, res);
```

Inter-servlet communication using Send Redirect

Send Redirect can be used to communicate between two servlet present in different servers, the output will be same as request dispatcher forward example but the url of the page will be changed to redirected page

Example:

```
response.sendRedirect  
("http://localhost:9999/Test/ServletTwo");
```

2.10 Important Questions

1. What is a servlet?
2. Can we use the constructor, instead of `Init()`, to initialize Servlet?
3. What is Servlet Context?
4. What Is A Servlet Filter? Explain life cycle.
5. What Is A War File? Describe the structure in brief.
6. What Is Genericservlet Class?
7. How Can The Session In Servlet Be Destroyed?
8. What Are The Mechanisms Used By A Servlet Container For Maintaining Session Information?

9. What Is The Procedure For Initializing A Servlet?
10. What Is The Web Container?
11. What Are The Uses Of Servletrequest?
12. What Are The Uses Of Servletresponse Interface?
13. How Http Servlet Handles Client Requests?
14. What Is Pre Initialization Of A Servlet?
15. How Do You Communicate Between The Servlets?
16. What Are The Differences Between A Session And A Cookie?
17. Why Should We Go For Inter Servlet Communication?
18. What's The Servlet Interface?
19. What Is The Difference Between Servletcontext And Servletconfig?
20. What is different between web server and application server?
21. What is the difference between GET and POST method?
22. What is MIME Type?
23. What is a web application and what is it's directory structure?
24. What are common tasks performed by Servlet Container?
25. What is ServletConfig object?
26. What is difference between GenericServlet and HttpServlet?
27. What is servlet attributes and their scope?
28. How can we invoke another servlet in a different application?
29. What are the phases of servlet life cycle?
30. What are life cycle methods of a servlet?

UNIT 3 JSP

Learning Objectives

After going through this unit you will be able to understand.

- What is JSP?
- The keywords and Syntax of JSP
- What are the different Page directives?
- Use of Include Directive.
- How to declare data in JSP?
- How methods are defined?
- What is a Scriptlet?
- Implicit Objects.
- What are Custom Tags?
- Session Tracking process.
- What is a Page Context object?
- JSP Exceptions.

3.0 INTRODUCTION

In the previous chapter, we learned how to generate dynamic Web pages with servlets.

In the Servlet examples most of the code generated output consisted of the HTML elements. Only a small part of the code deal with the business logic. Servlet writers have to be Java programmers. However, Web application developers and Web site designers, may not know Java. It is actually difficult for people who are not familiar with Java programming, to implement, maintain and create a Web application that consists of primarily of servlets. The solution to this problem is Java Server Pages (JSP) an extension of servlet technology that separates the presentation from the business logic.

3.1 Java Server Pages- JSP

Java Server Pages or JSP simplify dynamic Web content delivery. JSP allowse Web application programmers to reuse predefined components and interact with components using server-side scripting to create dynamic contents or dynamic web pages. JSP is having in built Custom-tag libraries that can be used to hide complex code for database access as well as other useful services for dynamic Web pages.

The JSP having various packages that consisting of the classes and interfaces that are specific to JavaServer Pages programming.

JavaServer Page (JSP) are just like Microsoft's *Active Server Pages* (ASP) that allows web developer to combine mix the features of *static* HTML with *dynamically generated* HTML – so that the *business logic* and the *presentation* are well separated.

The advantages of JSP are:

1. **Separation of static and dynamic contents:** JSP allows to separate out of *static* contents from *dynamic* contents. The dynamic contents are created using the programming logic and inserted into the *static template*. This significantly helps to simplified the creation and maintenance of web contents.
2. **Reuse of components and tag libraries:** The reusable components such as *JavaBean*, *Enterprise JavaBean* (EJB) and tag libraries provides the dynamic contents.
3. **Java's power and portability**

JSPs are Internally Compiled into Java Servlets

What can done in JSP can also be done using Java servlets. Servlets and JSPs are complementary technologies. The Servlet pages are just like as "*HTML inside Java*", which is

good combination for implementing business logic - as it is mainly written in Java. Whereas JSP is "**Java inside HTML**", which is better for creating presentation - as it mainly contains HTML. In a *Model-View-Control* (MVC) application, servlets involves complex programming logic and are used as a Controller (C). JSPs are mainly deals with presentation and can be used as View (V). The JavaBean or EJB are used to implement Model (M).

Apache Tomcat Server

JSPs are just like a servlets, which consist of server-side programs that run inside a HTTP server. To execute and to support JSP/servlet, a Java-capable HTTP server is required. Tomcat Server is an *open-source* software foundation and (@ <http://tomcat.apache.org>) is the official *reference implementation* (RI) for Java servlet and JSP, provided *free* by Apache @ <http://www.apache.org>).

3.2 JSP SYNTAX

There are **four key components** to JSP page -**directives, actions, scripting elements and tag libraries**. **Directives** are messages to the JSP container-the server component that executes JSPs-that enable the programmer to specify page settings, to include content from other resources and to specify custom tag libraries for use in a JSP. The functionalities can be encapsulated in predefined tags using **Actions** that programmers can embed in a JSP. Based on the information sent to the server as part of a particular client request Actions are performed. Actions are also able to create Java objects for use in JSP scriptlets. **Scripting elements** permit web developer to insert Java code that interacts with components in a JSP and possibly other Web application components to perform and complete request processing. **Scriptlets** contain code fragments that describe the action to be performed in response to a user request are popularly known as special kind of scripting element. **Taglibraries** enables programmers to create custom tags and is a part of the tag extension mechanism. Such tags enable Web page designers has o manipulate JSP content without prior Java knowledge.

In this way we can say that JavaServer Pages look like standard XHTML or XML documents. In actual, JSPs usually include XHTML or XML markup. This type of markup is called as **fixed-template text or fixed-template data**. Fixed-template data helps a programmer to decide whether to use a servlet or a JSP. Programmers incline to use JSPs when little or none of the content is generated dynamically with Java code or most of the content sent to the client is fixed-template data. Programmers typically use servlets when only a small portion of the content sent to the client is fixed-template data. In actual, some servlets do not produce content, but they perform a task on behalf of the client, and then invoke other servlets or JSPs to provide a response. In most of the cases JSP and servlet technologies offers interchangeability. Whereas JSPs normally execute as part of a Web server with servlets.

The JSP container translates the JSP into a Java servlet, as soon as a JSP-enabled server receives the first request for a JSP. It handles the current request and future requests to the JSP. Literal text in a JSP is get converted to string literals in the servlet that actually represent the translated JSP. If errors occur during compiling the new servlet, then it will cause translation-time errors. The JSP container places the Java statements that implement the JSP's response in method `_jspService` at translation time. The JSP container invokes method `_jspService` to process the request when the new servlet compiles properly. To assist in processing the request the JSP may respond directly or may invoke other Web application components. The errors are known as request-time errors that occur during request processing.

If we compare overall, the JSP life cycle and request-response mechanism are the same as those of a servlet. Similar to servlet methods `init` and `destroy`, JSPs can override methods **`jspInit`** and **`jspDestroy`**, which the JSP container invokes when initializing and terminating a JSP, respectively. JSP programmers can define these methods using JSP declarations--part of the JSP scripting mechanism.

A Simple JSP Example

JSP expression inserting the date and time into a Web page.

//test.jsp

```
<html>
<head>
  <meta http-equiv = "refresh" content = "60" />
  <title>A Simple JSP Example</title>
  <style type = "text/css">
    .big { font-family: helvetica, arial, sans-serif;
          font-weight: bold;
          font-size: 2em; }
  </style>
</head>
<body>
  <p class = "big">Simple JSP Example</p>
  <table style = "border: 6px outset;">
    <tr>
      <td style = "background-color: black;">
        <p class = "big" style = "color: cyan;">
          <!-- JSP expression to insert date/time -->
          <%= new java.util.Date() %>
        </p>
      </td>
    </tr>
  </table>
</body>
</html>
```

As you can see, most of test.jsp consists of XHTML markup. In cases like this, JSPs are easier to implement than servlets. In a servlet that performs the same task as this JSP, each line of XHTML markup typically is a separate Java statement that outputs the string representing the markup as part of the response to the client. Writing code to output markup can often lead to errors. That's why in such scenarios JSP is preferred than Servlets. The key line in the above program is the expression

<%= new java.util.Date() %>

JSP expressions are delimited by <%= and %>. The preceding expression creates a new instance of class Date (package java.util). By default, a Date object is initialized with the current date and time. When the client requests this JSP, the preceding expression inserts the String representation of the date and time in the response to the client. [Note: **Because the client of a JSP could be anywhere in the world, the JSP should return the date in the client locale's format.** However, the JSP executes on the server, so the server's locale determines the String representation of the Date.

We use the XHTML meta element in line 9 to set a refresh interval of 60 seconds for the document. This causes the browser to request test.jsp every 60 seconds. For each request to test.jsp, the JSP container reevaluates the expression in line 24, creating a new Date object with the server's current date and time.

When you first invoke the JSP, you may notice a brief delay as GlassFish Server translates the JSP into a servlet and invokes the servlet to respond to your request.

3.3 Page Directives

A page directive provides attributes that get applied to entire JSP page. It defines page dependent attributes, such as scripting language, error page, and buffering requirements.

It is also used to provide instructions to a container that pertains to current JSP page.

Syntax of Page Directive:

```
<%@ page... %>
```

Example:

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
import="java.util.Date" pageEncoding="ISO-8859-1"%>
```

In the above example the attribute **language** defines the programming language used in the page, **contentType** defines the character coding scheme, **charset** defines the character set used, **import** indicates the classes which have been imported, **pageEncoding** defines the character encoding for JSP page.

1. Besides these there are many other attributes like session, extends, info, autoflush, isErrorPage etc.

3.4 Include Directives

An include directive instructs the JSP container to include an external file in the JSP page. This directive can be used anywhere in the JSP page.

General Syntax:

```
<% @ include file = "relative url" >
```

XML syntax:

```
<jsp:directive.include file = "relative url" />
```

If the file is in the same directory only file name is mentioned.

Example:

```
<% @ include file = "first.jspl" >
```

3.5 Data Declaration

A declaration statement declares variables or methods that you will use in Java code in your JSP page. It is compulsory to declare all the variables and methods before its use.

General Syntax for JSP Declarations:

```
<%! declaration; [ declaration; ]+ ... %>
```

XML equivalent of the above syntax:

```
<jsp:declaration>
  Java code fragment
</jsp:declaration>
```

Example:

```
<%! int j = 0; %>
<%! int a, b, c; %>
```

```
<%! Rectangle r = new Rectangle(25.0,30.5); %>
```

3.6 Method Definition

A function or method is used to perform a specific process.

Syntax;

```
<%!  
access_specifierreturn_datatypefunction_name(list of arguments)  
{  
java code fragment  
return expression  
}  
%>
```

The following example shows a method square which returns the square of the number passed to it.

```
<%!  
public int square(int num)  
{  
return num*num;  
}  
%>
```

3.7 Scriptlets

JavaServer Pages are dynamic pages which having content which is also dynamically generated as part of an XHTML document that is sent to the client in response to a request.

JSP programmers use Scripting concept to write Java code and logic in a JSP.

Scripting Components

The JSP scripting components may contain various components such as **expressions, declarations, scriptlets, comments, and escape sequences.**

Scriptlets are blocks of code delimited by **<% and %>**. **Scriptlets** contain Java statements that are placed in method `_jspService` at translation time by the container.

JSPs support various **comment** styles such as **XHTML comments, JSP comments and scripting-language comments**. **JSP comments** are delimited by **<%-- and --%>**. These comments can be placed throughout a JSP, but they are not allowed to be placed inside the scriptlets.

XHTML comments are delimited with **<!-- and -->**. These, statements also, can be placed in a JSP, but not inside scriptlets.

Scripting language comments are considered as **Java comments**, because currently Java is the only JSP scripting language.

Scriptlets can use Java's **end-of-line** `//` comments and traditional comments (delimited by **/* and */**).

JSP comments and scripting-language comments are discarded and are not presented in the response to a client. Client will see only the XHTML comments in the source code When they view the source code of a JSP response.

JSP expressions contain a Java expression that is evaluated when a client requests the JSP containing the expression and are delimited by **<%= and %>**. The container takes a responsibility to convert the result of a JSP expression to a String object, then result will be presented as output as part of the response to the client.

As already discussed, **Declarations** are delimited by `<%! and %>`, enable a JSP programmer to define variables and methods for use in a JSP. Variables become instance variables of the servlet class that represents the translated JSP. As well as methods is also a member of the class that represents the translated JSP. Declarations of variables and methods in a JSP having the same syntax as that of java is having. Hence, a statement of variable declaration must end with a semicolon, as in

```
<%! int counter = 0; %>
```

Character sequences or special characters that are used to delimit JSP code can be included in a JSP as literal characters in scripting elements, fixed template data and attribute values using **escape sequences**. The escape sequences used are:

Literal	Escape sequence	Description
<code><%</code>	<code><\%</code>	The character sequence <code><%</code> normally indicates the beginning of a scriptlet. The <code><\%</code> escape sequence places the literal characters <code><%</code> in the response to the client.
<code>%></code>	<code>%\></code>	The character sequence <code>%></code> normally indicates the end of a scriptlet. The <code>%\></code> escape sequence places the literal characters <code>%></code> in the response to the client.
<code>'</code>	<code>\'</code>	As with string literals in a Java program, the escape sequences for characters <code>'</code> , <code>"</code> and <code>\</code> allow these characters to appear in attribute values. Remember that the literal text in a JSP becomes string literals in the servlet that represents the translated JSP.
<code>"</code>	<code>\"</code>	
<code>\</code>	<code>\\</code>	

Table 3.1 The escape sequences used in JSP

Scriptlet Example

```
//welcome.jsp
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"><title>Processing "get" requests with data</title>
</head>
<!-- body section of document -->
<body>
  <% // begin scriptlet
    String name = request.getParameter( "firstName" );

    if ( name != null )
    {
  %><%-- end scriptlet to insert fixed template data --%>

    <h1>
      Hello <%= name %>, <br />
      Welcome to JavaServer Pages!
    </h1>

  <% // continue scriptlet

    } // end if
    else {
```

```

%><%-- end scriptlet to insert fixed template data --%>

<form action = "welcome.jsp" method = "get">
  <p>Type your first name and press Submit</p>

  <p><input type = "text" name = "firstName" />
    <input type = "submit" value = "Submit" />
  </p>
</form>

<% // continue scriptlet

  } // end else

%><%-- end scriptlet --%>
</body>
</html>

```

3.8 Implicit Objects

Implicit objects provide access to many servlet capabilities in the context of a JavaServer Page. Implicit objects have **four different kinds of scopes: request, application, page, and session**. The JSP container owns objects with application scope. Any JSP can manipulate such objects. Objects which having page scope can be exist only in the page that defines them. Each page has its own instances of the page-scope implicit objects. Request scope object can exist for the duration of the request. For example, a JSP can partially process a request, and then forward it to a servlet or another JSP for further processing. When request processing completes with a response to the client, request-scope objects go out of scope. Objects with session scope exist for the client's entire browsing session. The following list describes the JSP implicit objects and their scopes.

Table 3.2 Implicit objects

Implicit object	Description
<i>Application Scope</i>	
<code>application</code>	A <code>javax.servlet.ServletContext</code> object that represents the container in which the JSP executes.
<i>Page Scope</i>	
<code>config</code>	A <code>javax.servlet.ServletConfig</code> object that represents the JSP configuration options. As with servlets, configuration options can be specified in a Web application descriptor.
<code>exception</code>	A <code>java.lang.Throwable</code> object that represents an exception that is passed to a JSP error page. This object is available only in a JSP error page.
<code>out</code>	A <code>javax.servlet.jsp.JspWriter</code> object that writes text as part of the response to a request. This object is used implicitly with JSP expressions and actions that insert string content in a response.
<code>page</code>	An Object that represents the <code>this</code> reference for the current JSP instance.
<code>pageContext</code>	A <code>javax.servlet.jsp.PageContext</code> object that provides JSP programmers with access to the implicit objects discussed in this table.
<code>response</code>	An object that represents the response to the client and is normally an instance of a class that implements <code>HttpServletResponse</code> (package <code>javax.servlet.http</code>). If a protocol other than HTTP is used, this object is an instance of a class that implements <code>javax.servlet.ServletResponse</code> .
<i>Request Scope</i>	
<code>request</code>	An object that represents the client request and is normally an instance of a class that implements <code>HttpServletRequest</code> (package <code>javax.servlet.http</code>). If a protocol other than HTTP is used, this object is an instance of a subclass of <code>javax.servlet.ServletRequest</code> .
<i>Session Scope</i>	
<code>session</code>	A <code>javax.servlet.http.HttpSession</code> object that represents the client session information if such a session has been created. This object is available only in pages that participate in a session.

3.9 Custom Tags

User-defined tags are known as **custom tags**.

To create a custom tag, following components are required:

- 1) **Tag handler class:** This class describe, what our custom tag will do when it is used in a JSP page.
- 2) **TLD file:** Tag descriptor file used to specify tag name, tag handler class and tag attributes.
- 3) **JSP page:** It describes a JSP page in which we will be using our custom tag.

Example:

In the below example we are creating a custom tag MyTag which will display the message “This is a custom tag” when used in a JSP page.

Tag handler class:

A tag handler class should implement Tag/IterationTag/ BodyTag interface. This class can also extend TagSupport/BodyTagSupport/SimpleTagSupport class. All the classes that are present inside javax.servlet.jsp.tagext, that support custom tags. Consider the example given below which is extending the class SimpleTagSupport.

Details.java

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
public class Details extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {

        JspWriter out = getJspContext().getOut();
        out.println("This is a custom tag");
    }
}
```

TLD File

This file should present at the location: Project Name/WebContent/WEB-INF/ and it should have a **.tld** extension.

Note:

<name> tag: custom tag name. In the example mentioned above a tag having the name as MyTag
<tag-class> tag: Fully qualified class name. The package in which the tag handler class Details.java is present has to be written.

message.tld

```
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>My Custom Tag</short-name>
<tag>
<name>MyMsg</name>
<tag-class>mypackage.Details</tag-class>
<body-content>empty</body-content>
</tag>
</taglib>
```

Using custom tag in JSP:

Above we have created a custom tag named Tag Here we will be using it.

Its important to note that taglib directive should have the TLD file path in url field. In the above example we have given the path of the file message.tld file which is newly created. Choose any prefix and specify it in taglib directive's prefix field. Here this tag is specified as myprefix.

Custom tag is called like this: <prefix:tagName/>. Our prefix is myprefix and tag name is MyTag so we have called it as <myprefix:MyTag/> in the below JSP page.

```

<% @ taglib prefix="myprefix" uri="WEB-INF/message.tld"%>
<html>
<head>
<title>JSP Custom Tag Example</title>
</head>
<body>
<myprefix:MyTag/>
</body>
</html>

```

3.10 Session Tracking in JSP

Sessions allows clients to store his or her data across multiple HTTP requests. The HTTP server does not maintain a reference or keep any record of client previous request.

3.10.1 HttpSession Methods

- **getAttribute** : it returns stored value from session object. It returns null if no value is associated with name.
- **setAttribute** : It associates a value with name.
- **removeAttribute** : This method is used to removes all the values associated with name.
- **getAttributeNames** : It is used to return names of all attributes in the session.
- **getId** : it returns unique id in the session.
- **isNew** : It is used to ensure whether the session which is created is new to client.
- **getCreationTime** : It returns time of session at which it is created.
- **getLastAccessedTime** : It returns last accessed time of session by client.
- **getMaxInactiveInterval** : It gets maximum access time of session in seconds before session GET invalidated.
- **setMaxInactiveInterval** : It sets maximum amount of time session in seconds between client requests before session being invalidated.

Following are several mechanism used to preserve session between client and web server:

1.10.2 Cookies

A cookie is a small portion of data sent from a website and stored in a user's web browser while the user is browsing that website and , also are also called as an HTTP cookie, web cookie, or browser cookie.

Cookies are commonly used for session management as cookie's value can uniquely identify a client. Browser uses cookie.txt file to stores each message. Browser sends the cookie back to the server, when you request another page from the server. Cookies have lifespan and Client can flushed them from browser at the end of lifespan.

Cookie objects have following methods.

1. **getComment ()** : It is used to returns comment that describe the purpose of the cookie.
2. **getMaxAge ()** : Returns maximum specified age of the cookie.
3. **getName()** : It is used to return name of the cookie.
4. **getPath()** : It is used to return URL for which cookie is targeted.
5. **getValue()** :Returns value of the cookie.
6. **setComment(String)** :It is used to describe the purpose of cookies using this comment.
7. **setMaxAge(int)** : Sets maximum lifespan or age of the cookie. Cookie will be deleted, If zero value causes.

8. **setPath(String)** : It determines URL with which Cookie should begin.
9. **setValue(String)** : Sets the value of the cookie. The Values passed to this function as parameter should not contain with special characters such as white space, brackets, equal sign, comma, double quote, slashes , “at” sign, colon and semicolon.

10. 3.10.3 Hidden Form Fields

It is hidden (invisible) text field used for maintaining the state of user. We store the information in the hidden field and get it from another servlet. Following sentence describe how to store value in hidden field.

```
<input type="hidden" name="uname" value="prodigy">
```

Here, *name* is hidden field name and *prodigy* is hidden field value. The specified name and value are automatically included in the GET or POST data, when the form is submitted.

11. 3.10.4 URL Rewriting

To encode every URL a static HTML page or form must be dynamically generated. You must consider web container need to use URL-rewriting, If you are not able to verify that every user of web application uses cookies. Session tracking can also be enable using URL rewriting, If the browser does not support cookies, or if cookies are disabled.

To store the session ID a web container attempts to use cookies. If that fails then web container tries to use URL-rewriting. URL rewriting basically includes the session ID within the link itself as a name/value.

Adding the session ID to a link contain following methods:

- **response.encodeURL ()**: This method used to associates a session ID with a given URL.
- **response.encodeRedirectURL ()** : If you are using redirection, this method can be used.

3.11 Page Context

To store and retrieve the page-related information and sharing objects, JSP Page Context Object is used. PageContext is an instance of javax.servlet.jsp.PageContext. PageContext is used to **findAttribute**, **setAttribute**, **getAttribute** and **removeAttribute** and it has the following scope.

- Page_Context scope
- Request_Context scope
- Session_Context scope
- Application_Context scope

findAttribute(String AttributeName): This method is used to find various attributes like the page, session, request, application. Return type is an object. It may returns null, If there is no attribute.

Syntax

```
pageContext.findAttribute("name of attribute");
```

getAttribute(String AttributeName, int scope): It is used to return the attribute with specified

scope. This is same as findAttribute() method but getAttribute look for specific scope. It returns the object, if there is no attribute, it returns null.

Syntax

Object object = pageContext.getAttribute("Attribute name", PageContext.SESSION_CONTEXT);

removeAttribute(String AttributeName, int scope): This method is used to remove the attribute from given scope and there is no return type of this method.

Syntax

pageContext.removeAttribute("Attribute Name",PageContext.REQUEST_CONTEXT);

setAttribute(String AttributeName, Object AttributeValue, int scope): This method accept three parameters and used to set the name of the attribute, attribute value and scope of that object. This method has no return type.

Syntax

pageContext.setAttribute("AttributeName","AttributeValue",
PageContext. APPLICATION_CONTEXT);

Example of JSP Page Context Object.

```
<html>
  <body>
<form action="pageContext.jsp">
    Name:  <input type="text" name="Name"/></br>
    FullName:<input type="text" name="fullName"/></br></br>
    <input type="submit" value="submit"/>
  </form>
</body>
</html>
```

Here in this example we have created two text boxes named as **Name** and **fullName** and also created submit button.

pageContext.jsp

```
<html>
<body>
<form action="getPageContext.jsp">
<%
String Name = request.getParameter("Name");
String fullName = request.getParameter("fullName");
out.println("Hello "+ Name+ " ");
pageContext.setAttribute("Name", Name, PageContext.SESSION_SCOPE);
pageContext.setAttribute("fullName", fullName, PageContext.SESSION_SCOPE);
%>
<input type="submit" value="Click Here"/>
</form>
</body>
</html>
```

request.getParameter() The details which are placed in HTML page can be retrived using this method. **pageContext.setAttribute()** method is used to set the name of the attribute, attribute value and scope of that object. This method has no return type.

getPageContext.jsp

```
<html>
<body>
<%
String Name = (String) pageContext.getAttribute("Name", PageContext.SESSION_SCOPE);
String fullName = (String) pageContext.getAttribute("fullName", PageContext.SESSION_SCOPE);
out.println("Hi " + Name + " ");
out.println("This is your fullname :"+fullName);
%>
</body>
</html>
```

getAttribute(String AttributeName, int scope) This method is used to return the attribute with specified scope.

3.12 Exceptions

The exception is runtime error which is generally an object that is thrown at runtime. Exception Handling is the method to handle the runtime errors. Exception may occur any time in your web application due to logical mistakes done by the developer. Hence exception handling is a required to be done at safer side by the web developer. In JSP it offers two ways to handle exception:

1. By **errorPage** and **isErrorPage** attributes of page directive
2. By **<error-page>** element in web.xml file

Let us consider and understand the example of exception handling in jsp by the elements of page directive

In this case, developer need to define and create a page to handle the exceptions, as in the error.jsp page. The pages in which exception may occur have to define the errorPage attribute of page directive, as in the process.jsp page.

There are 3 files:

- index.jsp for input values
- arithmetic.jsp for dividing the two numbers and displaying the result
- error.jsp for handling the exception

index.jsp

```
<form action="arithmetic.jsp">
Enter first Number :<input type="text" name="num1" /><br/><br/>
Enter second Number :<input type="text" name="num2" /><br/><br/>
<input type="submit" value="Divide"/>
</form>
```

arithmetic.jsp

```
<% @ page errorPage="error.jsp" %>
<%
String n1=request.getParameter("num1");
String n2=request.getParameter("num2");

int a=Integer.parseInt(n1);
int b=Integer.parseInt(n2);
int c=a/b;
out.print("division of numbers is: "+c);
%>
```

error.jsp

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

```
<h2>Sorry something has gone wrong!</h2>
```

Exception message is: `<%= exception %>`

Example of exception handling in jsp by specifying the error-page element in web.xml file

This approach is better because you don't need to specify the `errorPage` attribute in each jsp page. Specifying the single entry in the `web.xml` file will handle the exception. In this case, either specify exception-type or error-code with the location element. If you want to handle all the exception, you will have to specify the `java.lang.Exception` in the exception-type element. Let's see the simple example:

There are 4 files:

- `web.xml` file for specifying the error-page element
- `index.jsp` for input values
- `arithmetic.jsp` for dividing the two numbers and displaying the result
- `error.jsp` for displaying the exception

1) `web.xml` file if you want to handle any exception

```
<web-app>

  <error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
  </error-page>

</web-app>
```

This approach is better if you want to handle any exception. If you know any specific error code and you want to handle that exception, specify the `error-code` element instead of `exception-type` as given below:

1) `web.xml` file if you want to handle the exception for a specific error code

```
<web-app>

  <error-page>
    <error-code>500</error-code>
    <location>/error.jsp</location>
  </error-page>

</web-app>
```

2) `index.jsp` file is same as in the above example

3) `arithmetic.jsp`

Now, you don't need to specify the `errorPage` attribute of `page` directive in the jsp page.

```
<% @ page errorPage="error.jsp" %>
<%
String n1=request.getParameter("num1");
String n2=request.getParameter("num2");
int a=Integer.parseInt(n1);
int b=Integer.parseInt(n2);
```

```
int c=a/b;  
out.print("division of numbers is: "+c); %>
```

4) error.jsp file is same as in the above example

3.13 Important Questions

1. Explain JSP and tell its uses.
2. Explain JSP Technology.
3. Explain Implicit objects in JSP.
4. How can multiple submits due to refresh button clicks be prevented?
5. How to restrict page errors display in a JSP page?
6. What are JSP Actions?
7. Explain JSP lifecycle methods.
8. Explain handling of runtime exceptions.
9. Explain the various scope values for tag.
10. Explain page Directives.
11. Explain in brief attributes of page directives.
12. What are standard actions in JSP?
13. Explain the jsp:setProperty action.
14. Explain client and server side validation.
15. How can Automatic creation of session be prevented in a JSP page?
16. Explain the jspDestroy() method.
17. How is JSP better than Servlet technology?
18. Explain in brief session maintaining between client and server.
19. What do you mean by custom tag explain it with suitable example.
20. What Is Jsp Page?
21. How Is Jsp Include Directive Different From Jsp Include Action. ?
22. What Is The Difference Between Directive Include And Jsp Include?
23. What Are The Different Ways For Session Tracking?
24. What Is Session?
25. How Method Is Declared Within Jsp Page?
26. State The Difference Between The Expression And Scriptlet?
27. What are the various exception handling methods?
28. List out any five exception handling methods in JSP.

Unit 4 Hibernate

Learning Objectives

After going through this unit, you will be able to understand

- basic concepts of ORM and its necessities
- introduction to Hibernate and building ORM with Hibernate.

4.1 INTRODUCTION TO HIBERNATE (FRAMEWORK)

Hibernate ORM which is also called as Hibernate is an object-relational mapping tool for the Java programming language. It maps an object-oriented domain model to a relational database by providing the framework for it. Hibernate replace direct, persistent database accesses with high-level object handling functions, to handle the object-relational impedance mismatch problems.

Hibernate is an open software, which is distributed under the GNU Lesser General Public License 2.1. The most important and primary feature of hibernate is mapping from Java classes to database tables, as well as also responsible for mapping from Java data types to SQL data types. Hibernate offers data query and retrieval facilities which is also considered as important feature of it. It helps to relieve the developer from the manual handling and object conversion of the result set by generating SQL calls.

4.1.1 Why Hibernate and not JDBC?

- JDBC maps Java classes to database tables (and from Java data types to SQL data types)
- Whereas the SQL queries are automatically generated in Hibernate.
- Data query and retrieval facilities are provided in hibernate which can help significantly to reduce development time. Whereas comparatively more time is required to manually handle data in SQL and JDBC.
- It makes an application portable to all SQL databases.
- To performing selective search Hibernate offers HQL.
- Hibernate also supports SQL Queries which are also known as Native Query.
- Primary and secondary level caching support is provided by Hibernate.
- Hibernate Can be used in both console and web-based applications
- The various components from the Hibernate framework selectively available to use for the developers.
- Hibernate can be used with Java Persistence API also called as JPA in short.
- In Hibernate xml file and annotations as metadata are supported.

4.1.2 What is ORM?

ORM stands for **Object-Relational Mapping** (ORM) is a most popular programming technique, which is used for converting data between relational databases and various object-oriented programming languages such as Java, C#, etc.

An ORM system has the following advantages over plain JDBC –

1. It allows the business logic code to access objects instead of DB tables.
2. The details of SQL queries are kept hidden from OO logic.
3. It does not required to deal with the database implementation.
4. In hibernate Entities are based on business concepts but not on the database structure.
5. Transaction management and automatic key generation.

An ORM solution having the various advantages that can be described as follow:

- Transparent Persistence (POJO/JavaBeans)
- Persistent/transient instances
- Automatic Dirty Checking
- Transitive Persistence

- Lazy Fetching
- Outer Join Fetching
- Runtime SQL Generation
- Three Basic Inheritance Mapping Strategies

Hibernate is a most suitable and popular tool for **ORM mappings** in Java. It helps to reduce a complexity and is specially advantage or beneficial for Java developers with limited knowledge of SQL.

4.2 HIBERNATE ARCHITECTURE

The following diagram demonstrate the main building blocks in hibernate architecture.

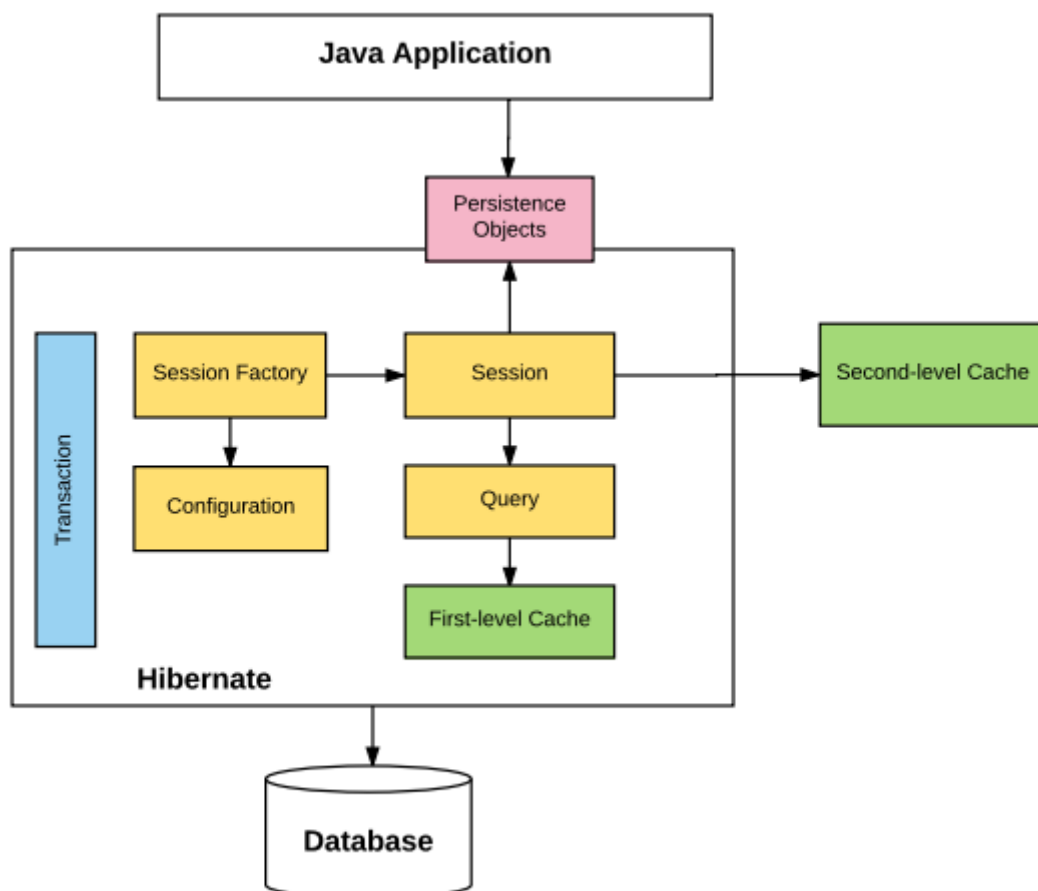


Figure 4.1: Building blocks in hibernate architecture

Explanation of each block:

1. **Configuration:** It is helpful and used by Session Factory to work with Java Application and the Database. It signifies an entire set of mappings of an application Java Types to an SQL database. It is written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class interpreted with @Configuration.
2. **Session Factory:** For an easy access to session object, any user application requests Session Factory to get access over it. Configuration information from above listed files is used by session factory, to instantiates the session object appropriately.

3. **Session:** Session describe the interaction point between the application and the database at any instance of time. This is signified by the `org.hibernate.Session` class. The instance of a session can be retrieved from the Session Factory bean.
4. **Query:** It allows to access one or more stored objects to applications by simply querying the database. Hibernate provides different techniques to query database, including NamedQuery and Criteria API.
5. **First-level cache:** It is the default cache which used while interacting with the database by Hibernate Session object. It is also known as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must remember that the first-level cache is available with the session object until the Session object is live.
6. **Transaction:** transaction allows to achieve data consistency, and also offers to rollback the transaction incase something goes unexpected.
7. **Persistent objects:** These are plain old Java objects also called as POJOs, which get persevered as one of the rows in the related table in the database by hibernate. Persistent object can be configured in configurations files represented by `hibernate.cfg.xml` or `hibernate.properties` or annotated which known with `@Entity` annotation.
8. **Second-level cache:** To store objects across sessions second level cache can be helpful and used. This type of catch required to be enabled explicitly and it is also required to provide the cache provider for a second-level cache. One of the well-known and common second-level cache providers is *EhCache*.

4.3 STEPS TO CREATE A HIBERNATE APPLICATION

1. Create POJO (Plain Old Java Object) Classes

The Hibernate application creation process begins with building the Java POJO class or classes, This is the first step and this step is depending on the application that will be persisted to the database. Let us understand with an example as mentioned in code bellow. Consider Student class with `getXXX` and `setXXX` methods to make it JavaBeans compliant class.

A POJO is a Java object that doesn't extend or implement some specialized classes and interfaces respectively required by the EJB framework. All normal Java objects are POJO.

It is important to provide JavaBeans compliant code as well as one attribute, when you design a class to be persisted by Hibernate. This would work as index like id attribute in the Student class.

```
public class Student {
    private int id;
    private String fullName;
    private int result;
    public Student () {}
    public Student (String fullName, int result) {
        this.fullName = fName;
        this.result = result;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFullName() {
        return fullName;
    }
}
```

```

}
public void setFullName( StringfullName ) {
this.fullName = fullName;
}
public int getResult() {
return result;
}
public void setResult( int result ) {
this. result = result;
}
}
}

```

2. Create Database Tables

The next and Second step for creating a hibernate application would be creating tables in your database. To provide persistence one should create the one table corresponding to each object. Let us Consider and understand with above object which need to be stored and retrieved into the following RDBMS table –

```

create table Student (
id INT NOT NULL auto_increment,
full_nameVARCHAR(100) default NULL,
result INT default NULL,
PRIMARY KEY (id)
);

```

3. Create Mapping Configuration File

The next step is to create a file that instructs Hibernate how to map the defined class or classes to the database tables and the file is called as mapping file.

```

<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name = "Student" table = "STUDENT">
<meta attribute = "class-description">
This class contains the student detail.
</meta>
<id name = "id" type = "int" column = "id">
<generator class="native"/>
</id>
<property name = "fullName" column = "full_name" type = "string"/>
<property name = "result" column = " result" type = "int"/>
</class>
</hibernate-mapping>

```

The mapping document should be saved in a file with the mentioned format as<classname>.hbm.xml. example Employee.hbm.xml. Let us understand and let us see more detail about the mapping document –

- The mapping document is an XML document having <hibernate-mapping> as the root element which contains all the <class> elements.

- To define specific mappings from a Java classes to the database tables, the elements are used called as class and represented as <class>. The database table name is stated using the table attribute where as The Java class name is stated using the name attribute of the class element.
- The <meta> element is optional element and can be used to create the class description.
- The unique ID attribute in class maps to the primary key of the database table by the <id> element. The property in the class is mentioned by the name attribute of the id element and the column attribute refers to the column in the database table. The hibernate mapping type, will convert from Java to SQL data type and it is hold by the type attribute.
- To generate the primary key values automatically the <generator> element within the id element is used. The class attribute of the generator element has to set to native, which will allow hibernate to pick up either identity, sequence or hilo algorithm to create primary key which is highly depending upon the capabilities of the underlying database.
- To map a Java class property to a column in the database table the <property> element is used. The property in the class is referred by the name attribute of the element and the column attribute mentions the column in the database table. The hibernate mapping type will convert from Java to SQL data type, this mapping type is hold by the type attribute

4. Create Application Class

The final step is to create application class which will be having the main method to run the application. In this application we will be saving few Student's records and then we will apply CRUD operations on those records.

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageStudent {
    private static SessionFactory factory;
    public static void main(String/* Method to CREATE an students in the database */)
    public Integer addStudent(String fname, int result){
        Session session = factory.openSession();
        Transaction tx = null;
        Integer studID = null;
        try {
            tx = session.beginTransaction();
            Student employee = new Student(fname, salary);
            studID = (Integer) session.save(student);
            tx.commit();
        } catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
        return studID;
    }
    /* Method to READ all the Students */
```

```

public void listStudents( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        List students = session.createQuery("FROM Students").list();
        for (Iterator iterator = students.iterator(); iterator.hasNext();){
            Student student = (Student) iterator.next();
            System.out.print("Name: " + student.getFullName());
            System.out.println(" Result: " + student.getResult());
        }
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
    } finally {
        session.close();
    }
}

/* Method to UPDATE Result for an student */
public void updateStudent(Integer StudID, int result ){
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Student student = (Student)session.get(Student.class, StudID);
        student.setResult( result );
        session.update(student);
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
    } finally {
        session.close();
    }
}

ng[] args) {
    try {
        factory = new Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {
        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
    }
    ManageStudent ME = new ManageStudent ();
    /* Add few student records in database */
    Integer sID1 = ME.addStudent ("Fred Flintstone", 60);
    Integer sID2 = ME.addStudent ("Mickey Mouse", 75);
    Integer sID3 = ME.addStudent ("John Brown", 80);
    /* List down all the student */
    ME.listStudents();
    /* Update student's records */
    ME.updateStudents(sID1, 85);
    /* Delete an student from the database */
    ME.deleteStudent(sID2);
    /* List down new list of the students*/
    ME.listStudents();
}

```

```

}
}
/* Method to DELETE an student from the records */
public void deleteStudent(Integer StudentID){
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Student student = (Student)session.get(Student.class, StudentID);
        session.delete(student);
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}
}
}

```

5. Compilation and Execution

To run the above application follow the steps mentioned below to compile and run it. Before you start compilation and execution ensure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create configuration file as hibernate.cfg.xml
- Create mapping file Student.hbm.xml as shown above.
- Create source file Student.java as shown above and compile it.
- Create another source file ManageStudent.java source file as shown above and compile it.
- Execute ManageStudent binary to run the program.

4.4 Persistent Object Life Cycle

- As you know that Hibernate works with normal Java objects that your application creates with the new operator. In raw form (without annotations), hibernate will not be able to identify your java classes; but when they are properly annotated with required annotations then hibernate will be able to recognize them and then work with them e.g. store in DB, update them etc. These objects can be said to mapped with hibernate.
- The instance of an object which is given and mapped to Hibernate, it can exist in any one of four different states: **transient, persistent, detached, or removed**.

4.4.1 Transient Object

- Transient objects exist in heap memory. Hibernate is not responsible to manage transient objects or nor to persist changes to transient objects.
-

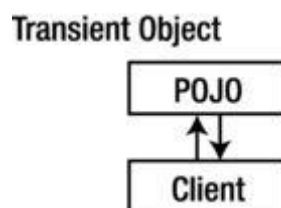


Figure 4.2 Transient Object

Transient objects are independent of Hibernate, one need to ask the session to save the transient object to the database, in order to persist the changes to a transient object. At this point Hibernate allocates the object an identifier and marks the object as being in persistent state.

4.4.2 Persistent Object

Hibernate manages the persistence for persistent objects, while Persistent objects exist in the database

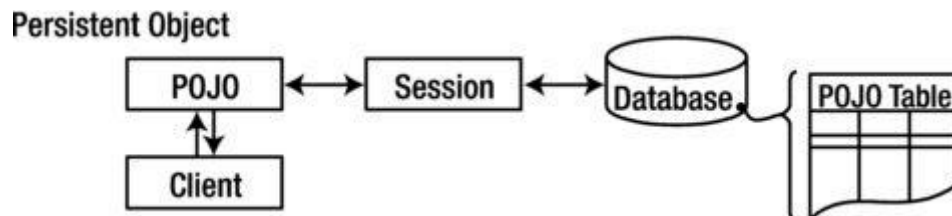


Figure 4.3 Persistent Object

Persistent objects are maintained by Hibernate. Hibernate will keep the database up to date when the application marks the changes as to be committed, If fields or properties change on a persistent object occurs

4.4.3 Detached Object

Detached objects have a existence in the database, but any changes made to the object will not be reflected in the database, and vice-versa. The image shown below describe the temporary separation of the object and the database.

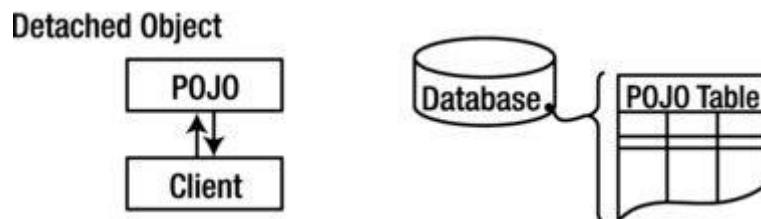


Figure 4.4 Detached Object

Detached objects exist in the database but these objects are not maintained by Hibernate. To create a detached object one can close the session that it was associated with, or by evicting it from the session with a call to the session's `evict()` method.

Some of the reason you might consider doing this would be modify the properties of the object in memory, to read an object out of the database, and then store the results some place other than your database. This would be an substitute to doing a deep copy of the object.

The application must reattach it to a valid Hibernate session, If anyone need to persist changes made to a detached object. A detached instance can be associated with a new Hibernate session when your application calls one of the `load`, `refresh`, `merge`, `update()`, or `save()` methods on the new session with a reference to the detached object. After the call, the detached object would be a persistent object managed by the new Hibernate session.

4.4.4 Removed Object

Removed objects are also called as persistent objects are objects that are being managed by Hibernate. This objects are passed to `remove()` method of Session's Class which will then

remove it. The entries in the database that correspond to removed objects are deleted, When the application makes the changes

4.4.5 Bullet Points

1. The POJO object which is recently created will be in the transient state. Transient object is e. not associated with any session object, and hence doesn't represent any row of the database. It's plain simple java object.
2. Persistent object is always associated with some unique hibernate session and represent one row of the database. Hibernate monitor all the Changes to persistent objects and are saved into database when commit call happen.
3. The objects those who were once persistent in past are known as Detached objects, and now they are no longer persistent. One must reattach detached objects to hibernate session, in-order to persist changes done in them.
4. The persistent objects that have been passed to the remove() method of session class are known as Removed objects, and these objects will be deleted as soon as modification held in the session will be committed to database.
that take place in the session as to be committed.

4.5 Hibernate with Servlets (Hibernate Servlet Integration)

4.5.1 Introduction

Developing an application using hibernate is a not a difficult task, if developer is familiar in flow of code. The developer must have knowledge that every hibernate application will be composed of minimum two files they are configuration file and hibernate mapping file, whereas remaining files are required java files. The main advantage of hibernate is tables will be created automatically after the execution of a program. Here input values are sent from the **Html page** to **Servlet** which calls **Dao class of Hibernate** and it will pass input values to **Database**.

Example

Let us understand with example such that, Employee details are given to Html page, it will store the details in the Database. If details are saved into Database, then Servlet gives the response as '**Employee details successfully saved**'. If it failed to save in that case the response will be '**Employee details already saved**'.

4.5.2 Conceptualfigure

Execution flows diagram of servlet-hibernate example.

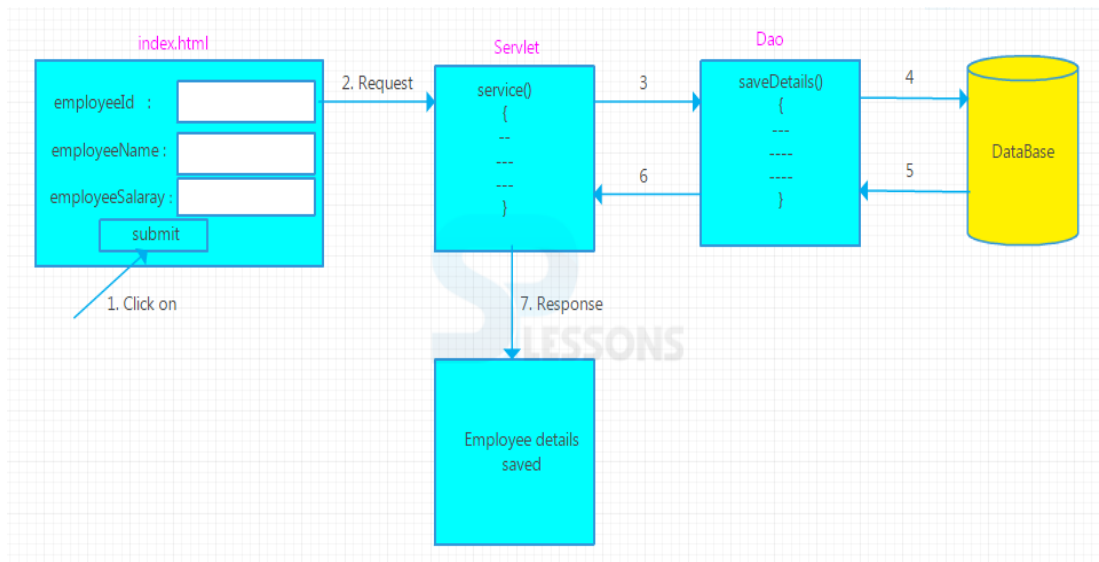


Figure 4.5: Execution flows diagram of servlet-hibernate example

- Add all the required **hibernate jar files** such as **ojdbc14.jar** and **servlet-api.jar** in **lib** folder.

Example

- Create the project directory structure.

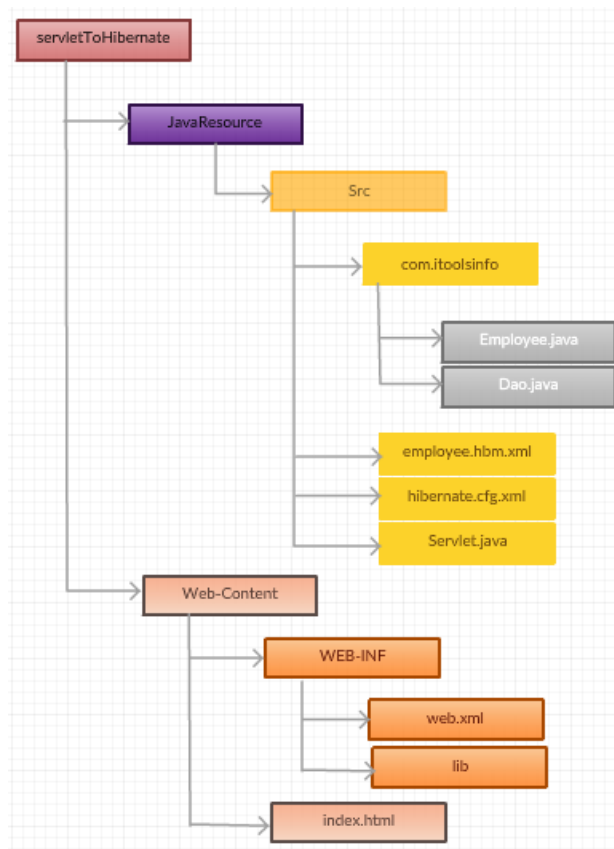


Figure 4.6: Project directory structure.

- Create the HTML file which will be responsible to forward the request to Servlet.

```

<!DOCTYPEhtml>
<html>
<head>
</head>
<body>
  <formation="ServletDemo">
    Employee Id   :<inputtype="text"name="id">
    Employee Name  :<inputtype="text"name="name">
    Employee Salary :<inputtype="text"name="salary">
    <inputtype="submit"value="submit">

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

```

ServletDemo.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;

import com.iquickinfo.Dao;

public class ServletDemo extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        System.out.println("this is servlet");

        int employeeId=Integer.parseInt(request.getParameter("id").trim());
        String employeeName=request.getParameter("name").trim();
        int salary=Integer.parseInt(request.getParameter("salary").trim());

        Dao dao=new Dao();
        boolean b=dao.saveDetails(employeeId, employeeName, salary);
        response.setContentType("text/html");
        PrintWriter out=response.getWriter();
        if(b==true)
        {
            out.println("<h1>Employee details sucessfully saved.</h1>");
        }
        else
        {
            out.println("<h1>Employee details already existed.</h1>");
        }
        out.println("");
    }
}

```

```

        out.close();
    }

}

```

The **doGet() method** is used to append the parameters to URL and are sent along with the header information. The version control in a Serializable class is represented as **serialVersionUID**. If **serialVersionUID** is not declared explicitly, then JVM will declare it automatically for you. JVM declare it based on various aspects of Serializable class, as described in the Java(TM) Object Serialization Specification. If the response has not been committed yet, then the **content type** of the response being sent to the client. The content type may have a character encoding specification.

- Set URL Pattern in web.xml file.

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="WebApp_ID" version="2.5">
    <display-name>ServletToHibernate</display-name>

    <servlet>
        <servlet-name>ServletDemo</servlet-name>
        <servlet-class>ServletDemo</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletDemo</servlet-name>
        <url-pattern>/servlet</url-pattern>
    </servlet-mapping>
</web-app>

```

- Create the persistence class.

Employee.java

```

package com.itoolsinfo;

public class Employee
{
    private int employeeId;
    private String employeeName;
    private int salary;
    public int getEmployeeId() {
        return employeeId;
    }
    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }
}

```

```

public String getEmployeeName() {
    return employeeName;
}
public void setEmployeeName(String employeeName) {
    this.employeeName = employeeName;
}
public int getSalary() {
    return salary;
}
public void setSalary(int salary) {
    this.salary = salary;
}
}

```

Set and **Get** methods are used to describe the feature of data encapsulation. Which will allow to define get methods to access these variables, and set methods to modify them, instead of giving direct accessing to class member variables.

- Now let us Map the persistence class in mapping file as shown bellow.

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

```

```

    <hibernate-mapping>
    <class name="com.itoolsinfo.Employee" table="employee">
    <id name="employeeId">
    <generator class="assigned"></generator>
    </id>
    <property name="employeeName"></property>
    <property name="salary"></property>
    </class>
</hibernate-mapping>

```

To produce the unique identifier for the objects of persistence class, **generator class** sub element of id is used. There are several generator classes described in the Hibernate Framework. All the generator classes represents the **org.hibernate.id.IdentifierGenerator interface**.

- Configure the mapping file in Configuration file.

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

```

```

<hibernate-configuration>

```

```

    <session-factory>

```

```

        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
        <property name="connection.username">system</property>
        <property name="connection.password">system</property>

        <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>

```

```

        <property name="hibernate.hbm2ddl.auto">create</property>
        <property name="show_sql">true</property>

        <mapping resource="employee.hbm.xml"/>

    </session-factory>

</hibernate-configuration>

```

- Create the Dao class and store the POJO class object details which will be sent to the servlet class.

Dao.java

```

package com.itoolsinfo;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class Dao
{
    public boolean saveDetails(int employeeId, String employeeName, int salary)
    {
        boolean flag=true;
        SessionFactory factory=new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
        Session session=factory.openSession();

        Employee employee=new Employee();
        employee.setEmployeeId(employeeId);
        employee.setEmployeeName(employeeName);
        employee.setSalary(salary);
        Transaction transaction=session.beginTransaction();
        try
        {
            session.save(employee);
            transaction.commit();
        }catch(Exception e)
        {
            transaction.rollback();
            flag=false;
        }
        session.close();
        return flag;
    }
}

```

4.6 HQL: Hibernate Query Language.

Hibernate Query Language which is called as HQL in short is an object-oriented query language, which is just analogous to SQL, HQL works with persistent objects and their properties, instead of operating on tables and columns. This is what the important difference between hql vs sql. HQL is a parent set of the JPQL, the Java Persistence Query Language. The every JPQL query is a valid HQL query, but not vice-versa that is all HQL queries are not valid JPQL queries.

HQL language is having its own syntax and grammar.

Hibernate can translate HQL queries into conventional SQL queries, which also provides an API that allows us to directly issue SQL queries as well.

4.6.1. HQL Syntax

HQL syntax is defined as an **ANTLR** (ANother Tool for Language Recognition) grammar. The Hibernate core download include the grammar files in the grammar directory. ANTLR is a tool which can be used to build the language parsers. Now let us discuss the syntax for the various fundamental CRUD operations here:

1. HQL Update Statement

UPDATE statement is used to alter the details of existing objects in the database. In-memory entities will not be updated to reflect changes resulting from issuing UPDATE statements.

The syntax of the UPDATE statement is as mentioned below:

hql update statement syntax

```
UPDATE [VERSIONED]
  [FROM] path [[AS] alias] [, ...]
  SET property = value [, ...]
  [WHERE logicalExpression]
```

- path – path in above query indicate the fully qualified name of the entity or entities
- alias – It is used to abbreviate references to specific entities or properties of the entities, and it must be used when property names in the query are mentioned, otherwise will be considered as ambiguous.
- VERSIONED – is used to update time stamps, if any, that are part of the entity being updated.
- property – It describes the names of properties of entities listed in the FROM path.
- logicalExpression – a where clause, is a condition for execution of query.
-

Let us understand the use of update statement with one example. In this example, let us try to update employee data with **hql update query multiple columns**.

hql update statement example

```
Query query=session.createQuery("update Employee set age=:age where name=:name");
query.setInteger("age", 32);
query.setString("name", "Freddy Flint");
int modifications=query.executeUpdate();
```

2. HQL Delete Statement

DELETE statement can be used to delete or removes existing objects and its details from the database. It will not be responsible to update In-memory entities to reflect changes resulting from DELETE statements. In other words we can say that Hibernate cascade

Here's the syntax of the INSERT statement:

hql insert statement example

```
INSERT  
  INTO path ( property [, ...])  
  Select
```

The name of an entity is path. The property names are the names of properties of entities listed in the FROM path of the incorporated SELECT query. The select query is an HQL SELECT query (as described in the next section).

As this HQL statement can only use data provided by an HQL select, its application can be limited. An example of copying users to a purged table before actually purging them might look like this:

hql insert statement example

```
Query query=session.createQuery("insert into purged_accounts(id, code, status) "+  
    "select id, code, status from account where status=:status");  
query.setString("status", "purged");  
int rowsCopied=query.executeUpdate();
```

4. HQL Select Statement

An HQL SELECT is used to query the database for classes and their properties. Here's the syntax of the SELECT statement:

hql select statement example

```
[SELECT [DISTINCT] property [, ...]]  
  FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]  
  WHERE logicalExpression  
  GROUP BY property [, ...]  
  HAVING logicalExpression  
  ORDER BY property [ASC | DESC] [, ...]
```

```
[SELECT [DISTINCT] property [, ...]]  
  FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]  
  WHERE logicalExpression  
  GROUP BY property [, ...]  
  HAVING logicalExpression  
  ORDER BY property [ASC | DESC] [, ...]
```


The fully qualified name of an entity is path. The alias names are used to indicate references to specific entities or their properties, and It must be used with property names in the query otherwise will be considered as ambiguous.

The property names are the names of properties of entities listed in the **FROM** path.

If **FETCH ALL PROPERTIES** is used, then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).

WHERE is used to create **hql select query with where clause**.

Important Question

1. Explain Hibernate Query Language in short.
2. List and Explain various HQL Statements.
3. What is Hibernate Servlet Integration? Explain with diagram.
4. Explain Persistent Object Life Cycle.
5. Explain Various objects with diagram in Persistent Object Life Cycle.
6. Explain the steps to create a hibernate application.
7. With neat diagram explain the architecture of Hibernate
8. Why Hibernate is advantageous over JDBC ?
9. Write a short note on Hibernate Framework

Unit 5

Java Spring Core

Introduction

The movement of Plain Old Java Object (POJO) was started in the beginning of the current century and became mainstream in the enterprise Java world. This quick popularity is certainly closely related with the open source movement during that time. Many projects appeared, and most of them helped the this programming model become mature with the time. This unit will tell how things were before this programming model existed in the enterprise Java community and discusses the problems of the old Enterprise JavaBeans (EJB) programming model. It's important that you understand the characteristics of the POJO programming model and what it provides to developers. The second half of the unit focuses on containers as well as the inversion of control patterns that are at the heart of the lightweight containers which are used today. We will learn what is a container? which services are offered? and how a container is made lightweight? We will also learn about the inversion of control patterns, its close relationship with dependency injection terminology.

The POJO means Plain Old Java Objects. The name was coined by Martin Fowler, Rebecca Parsons, and Josh MacKenzie to provide some different name. It represents a programming trend that aims to simplify the coding, testing, and deployment phases of enterprise Java applications. We will have a better understanding of what problems the POJO model solves after understanding the problems caused by EJB Programming model.

5.1 The Spring Framework

The Spring is a lightweight framework. It can be considered as a framework of frameworks because it provides support to various other Java frameworks such as Struts, Hibernate, EJB, JSF etc. The framework, can be defined as a structure where we can find solution of the various technical problems occurred in programming. The Spring is most popular application development framework used for programming in enterprise Java. Many of developers around the world use Spring Framework to create high performing, reusable code, and easily testable.

Spring framework is an open source Java platform. The first version was written by Rod Johnson, who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002. The framework was first released under the Apache 2.0 license in June 2003. The first milestone release, 1.0, was released in March 2004 with further milestone releases in September 2004 and March 2005.

Advantages of using Spring Framework

- The Spring enables developers to develop enterprise class applications using POJO programming model. The advantage of using only POJO is that we don't require an

EJB container product such as an application server but we have the option of using only robust Servlet container such as Apache Tomcat.

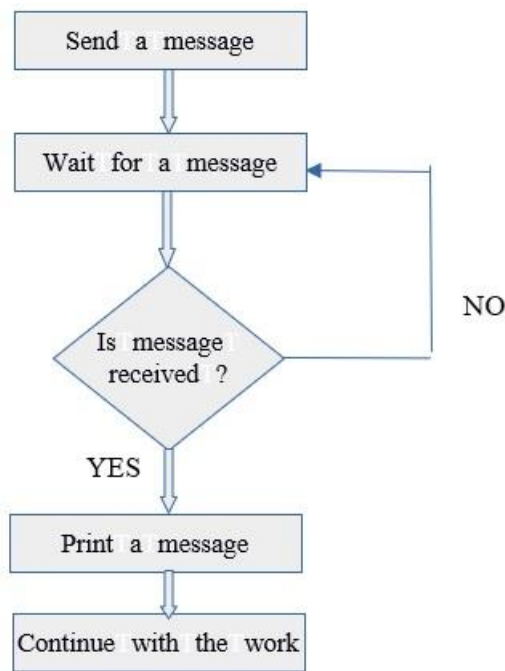
- Spring is organized in a modular fashion. Even though the number of packages and classes are significant, we have to worry only about the ones we need and dismiss the rest.
- Spring does not recreate anything, instead it makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and many other view technologies.
- Testing an application coded in Spring is very simple because environment-dependent code is moved into this framework. By using JavaBeanstyle POJO, it becomes easier to use dependency injection for injecting test data in the program.
- It's web framework is a well-designed web MVC framework, which provides a good option to web frameworks such as Struts or remaining over-engineered or less popular web frameworks.
- It provides a convenient API to translate technology-specific exceptions (thrown by JDBC, JDO or Hibernate) into consistent unchecked exceptions.
- The lightweight IoC containers lean to be lightweight, particularly when compared to Enterprise Java Beans containers. This is advantageous for developing and deploying applications on computers with small memory as well as CPU resources.
- It provides a consistent transaction management interface that can reduce to a local transaction and increase to global transactions.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. Let's understand the IOC and Dependency Injection first.

5.2 Inversion of Control

The prime aim of Inversion of control and Dependency Injection is to remove dependencies of an application. This makes the system more decoupled and rectifiable. Inversion of control is used to increase modularity of the program and make it extensible, and has applications in object-oriented programming and other programming paradigms. The term was used by Michael Mattsson in a thesis, taken from there by Stefano Mazzocchi and popularized by him in 1999 in a defunct Apache Software Foundation project, Avalon, then further popularized in 2004 by Robert C. Martin and Martin Fowler.

First let's try to understand IOC (Inversion of control). If we go back to old computer programming days, program flow used to run in its own control. For instance let's consider a simple chat application flow as shown in the below flow chart.



The communication here will be taken place in the following manner.

1. The user sends a message.
2. Our application waits for the message from the other end.
3. If no message is found it goes to Step 2 else moves to Step 4.
4. Prints the message.
5. User continues with his work after it.

Now if you examine the program flow, it is direct sequential. The program is in control of itself. Inversion of control means the program delegates control to someone else who will drive this flow. For instance if we make the chatting application event based then the flow of the program will go something as below:-

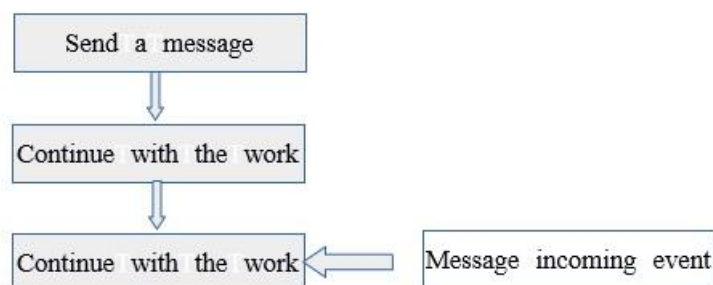


Fig. 5.2 Event based chatting application

6. The user sends a message.
7. Our application waits for the message from the other end.
8. If no message is found it goes to Step 2 else moves to Step 4.
9. Prints the message.
10. User continues with his work after it.

You can see that the program flow is not sequential, its event based. So now the control is altered. It means that the internal program controlling the flow, events drive the program flow. Event flow approach is more flexible as there's no direct invocation which leads to more flexibility.

We can say that IOC are implemented by only events. You can delegate the control flow by callback delegates, observer pattern, events, DI (Dependency injection) and lot of other ways.

IOC (Inversion of control) is a general parent term whereas DI (Dependency injection) is a subset of IOC. IOC is a concept where the flow of application is inverted. So for example rather than the caller calling the method.

```
SomeObject.Call();
```

Will get replaced with an event based approach as shown below.

```
SomeObject.WhenEvent += Call();
```

In this code the caller is exposing an event and when that event occurs he is taking action. It's based on the principle "Don't call us, we will call you". For example, in Hollywood when artists used to give auditions the judges would say them "Don't call us, we will call you".

This conceptualization makes code more flexible as the caller is not aware of the object methods and the object is not aware of caller's program flow.

Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It's most often used in the context of object-oriented programming.

By contrast with traditional programming, in which our custom code calls to a library, IoC enables a framework to take control of the program flow and make calls to our custom code. In order to enable this, frameworks use abstractions with additional behavior built -

in. If we want to add our own behavior, we need to extend the classes of the framework or plug-in our own classes.

The advantages

- The decoupling the execution of a task from its implementation
- Making it easier to switch between different implementations
- Program modularization
- Ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts

5.3 IoC Container

The Spring container is at the center of the Spring Framework. It create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration meta-data provided. The configuration meta-data can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

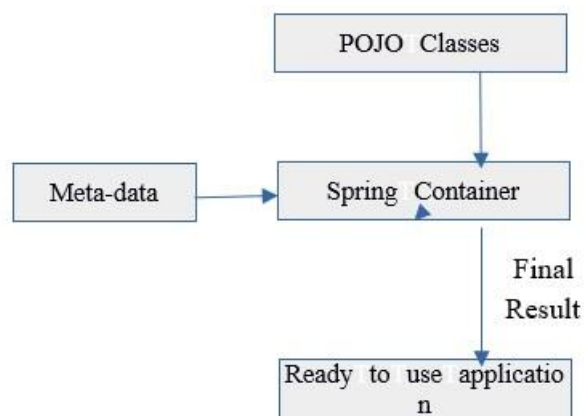


Fig. 5.3 Working of IOC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works with it accordingly. The main operations performed by IoC container are:

- Instantiate the application class
- Configure the object
- Assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. BeanFactory
2. ApplicationContext

BeanFactory

This is the simplest container providing the basic support for DI and is defined by the `org.springframework.beans.factory.BeanFactory` interface. The BeanFactory and related interfaces, such as `BeanFactoryAware`, `InitializingBean`, `DisposableBean`, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

Spring has objects of the BeanFactory type that behave like the Factory object. We specify the blueprints object in a configuration file, which is an XML file, and then supply it to BeanFactory. Afterwards, if we need the instance of any object, we can ask BeanFactory for the same, which then refers to the XML file and constructs the bean as given. This bean is now a Spring bean as it has been created by Spring Container and is returned to us. This is how it is done.

1. Spring has BeanFactory, which creates new objects for us. So, the XYZ object will call BeanFactory.
2. BeanFactory would read from Spring XML, which contains all the bean definitions. Bean definitions are the blueprints. BeanFactory will then create beans from this blueprint and then make a new Spring bean.
3. Finally, this new Spring bean is handed back to XYZ, as shown here:

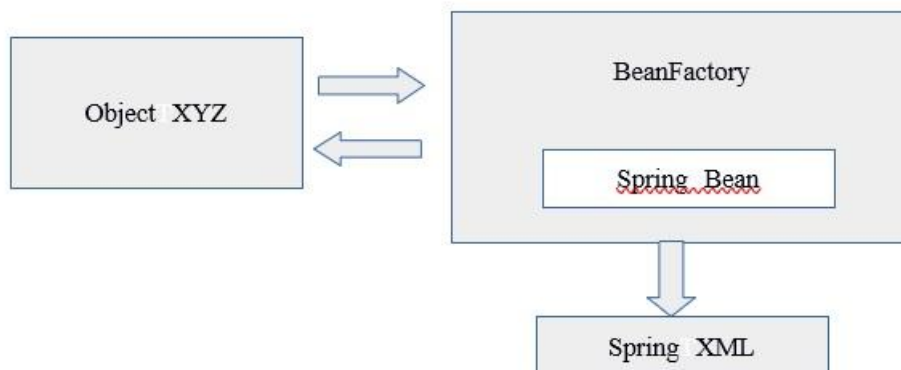


Fig. 5.4 BeanFactory Architecture

The advantage here is that this new bean has been created in this BeanFactory, which is known by Spring. Spring handles the creation and the entire life cycle of this bean. So, for this case, Spring acts as container for this newly created Spring bean. BeanFactory is defined by the `org.springframework.beans.factory.BeanFactory` interface. The BeanFactory interface is the central IoC container interface in Spring and provides the basic end point for Spring Core Container towards the application to access the core container service. It is responsible for containing and managing the beans. It is a factory class that contains a

collection of beans. It holds multiple bean definitions within itself and then instantiates that bean as per the client's demands. BeanFactory creates associations between collaborating objects as they're instantiated. This removes the burden of configuration from the bean itself along with the bean's client. It also takes part in the life cycle of a bean and makes calls to custom initialization and destruction methods.

ApplicationContext

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the `org.springframework.context.ApplicationContext` interface.

The ApplicationContext container includes all kinds of functionality of the BeanFactoryContainer, so it is generally suggested over BeanFactory. BeanFactory can still be used for lightweight applications like mobile devices or applet-based Java applications where data volume and speed is momentous.

Like BeanFactory, ApplicationContext is also used to represent Spring Container, built upon the BeanFactory interface. ApplicationContext is suitable for Java Enterprise Edition applications, and it is always preferred over BeanFactory. All functionality of BeanFactory is included in ApplicationContext.

The `org.springframework.context.ApplicationContext` interface defines ApplicationContext. ApplicationContext provides advanced features to our Spring applications that make them enterprise-

level applications, whereas BeanFactory provides a few basic functionalities. These are given as below.

- Except providing a means of resolving text messages, ApplicationContext also includes support for i18n of those messages.
- A generic way to load file resources, such as images, is provided by ApplicationContext.
- The events to beans that are registered as listeners can also be published using ApplicationContext.
- ApplicationContext handles certain operations on the container or beans in the container declaratively, which have to be handled with BeanFactory in a programmatic way.
- It provides ResourceLoader support. This is used to handle low-level system resources, Spring's Resource interface, and a flexible generic abstraction. ApplicationContext itself is ResourceLoader. Hence, access to deployment-specific Resource instances is provided to an application.
- It provides MessageSource support. MessageSource, an interface used to obtain localized messages with the actual implementation being pluggable, is implemented by ApplicationContext.

Creating a JavaBean

A bean in Java context is a simple Java class which has some properties (also called fields) and their getter and setter methods. These properties can be regarded as instance variables of the bean. The name of the properties and their getter/setter methods should adhere to the JavaBeans specifications that follow general conventions for a Java class.

How to create ?

There are many ways to create a bean in Spring.. All of the methods involve creating a class which will be regarded as a bean. So first let's create a class called User.

```
package com.mitu;

public class Manager {
    private String name;

    private String address;

    public String getName() {
        return name;
    }

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

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

In this Java program, we are creating an instance of this class using new keyword but in Spring, there are different ways to create an instance of a bean class without using new. Although, using new keyword is permitted but is not recommended since injection of properties and Auto-wiring does not work if you create a bean using new. Following are the ways to create a bean in Spring.

Method 1 : Declaring a bean in XML configuration file

This is the most primitive approach of creating a bean. The bean is declared in Spring's XML configuration file. Upon startup, Spring container reads this configuration file and creates and initializes all the beans defined in this file which can be used anytime during application execution. Here is how we define a bean in configuration XML.

```
<bean id= "user" class="com.mitu.Manager"></bean>
```

This method assumes that you are familiar with Spring XML configuration and how it is configured.

Details

A bean in Spring XML configuration file is defined by using element declaration. The id attribute is identifier of the bean and get its reference during execution. The class attribute should contain the fully qualified class name, including top level packages. This bean can be retrieved and used in the application in the following way.

```
public class Appl {  
  
    @SuppressWarnings("resource")  
    public static void main(String[] args) {  
  
        ApplicationContext context = new AnnotationConfigApplicationContext(Manager.class);  
        Manager man = (Manager)context.getBean("manager");  
        System.out.println(man.getName());  
        System.out.println(man.getAddress());  
    }  
}
```

Lets get in to this code. The id attribute is not mandatory and is only required when you want to access this bean in application or provide this bean as a dependency to some other bean.

The id of a bean should be unique otherwise we will get an error at start up like org.springframework.beans.factory.parsing.BeanDefinitionParsingException: Configuration problem: Bean name 'test' is already used in this <beans> element.

The scope of bean will be singleton by default, which means every time the bean is referred, same instance will be given. The scope of the bean can be changed by adding a scope attribute in bean declaration.

Bean properties referred in the property element should have valid setter methods otherwise following error will be received.

Bean property 'name' is not writable or has an invalid setter method.

Spring automatically performs the string to number conversion when the type of a class property is numeric while the values provided in XML configuration file are there in string format. But, we will get a `java.lang.NumberFormatException` if the property is not possible to get converted to number.

Method 2 : Using @Component annotation

@Component annotation above a class indicates that this class is a component and should be automatically detected and instantiated. Thus a Spring component bean will be created in this way.

```
package com.mitu;

@Component
public class Manager {
    private String name;

    private String address;

    public String getName() {
        return name;
    }

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

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

Explanation:

We need to instruct Spring container to find our beans and tell it about the package where it should find the beans, in order for a bean to be auto-discoverable and instantiated. Hence, for instantiating annotated beans, we require to add below declaration in our Spring XML configuration file :

```
<context:component-scan base-package= "com.mitu" />
```

If the bean annotated with `@Component` annotation is outside of the package given in `basePackage` attribute of this above element, then it will not be scanned and instantiated and thus won't be found. This bean can also be used in the same way as the bean in the previous method. Thus, only bean declaration methods differs while the method of using the bean remains the same as of the first.

`@Component` annotation can only be applied at the class level. Applying it to some other location such as above a field will result in a compiler error. The annotation `@Component` is not allowed for this location.

If a bean is referred and it has not been annotated with `@Component` or declared in XML configuration then we will get an exception as shown below.

`org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type [com.mitu.Manager] is defined`

Automatic instantiating a bean using `@Component` annotation needs that there must be a default constructor present in the class. As we know that, default constructor is a constructor without any arguments. If there is no constructor present in a class, then a default constructor is automatically created and used. If there is a constructor in your bean class which accepts some arguments then you have to create a default constructor yourself. If default constructor is not available, then following error will be generated.

`org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.mitu.User]: No default constructor found`

5.4 Dependency Injection

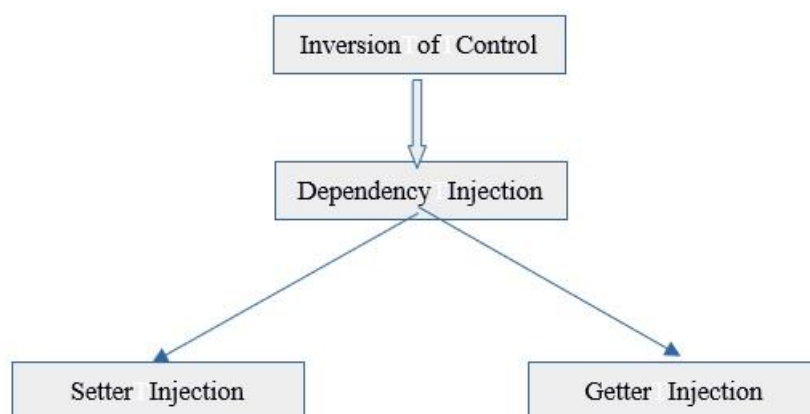


Fig. 5.6 Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that is used to remove the dependency from the programming code so that it can be easy to manage and test our application. Dependency Injection makes our programming code loosely coupled.

It solves problems such as-

How can an application or class be independent of how its objects are created?
How can the way objects are created be specified in separate configuration files?
How can an application support different configurations?

Creating objects directly within the class that requires the objects is inflexible because it commits the class to particular objects and makes it impossible to change the instantiation later independently from the class. It stops the class from being reusable if other objects are required, and it makes the class hard to test because real objects can't be replaced with mock objects.

A class is no longer responsible for creating the objects it needs, and it doesn't have to delegate instantiation to a factory object as in the Abstract Factory design pattern.

In order to learn the DI, let's understand the Dependency Lookup (DL).

Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

```
X obj = new Ximpl();
```

In such way, we get the resource (instance of X class) directly by using new keyword. Another way is to use factory method:

```
X obj = X.getX();
```

Using this way, we get the resource (instance of X class) by calling the static factory method getX().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) such as:

1. Context con = new InitialContext();
2. Context environmentCon = (Context) con.lookup("java:comp/env");
3. X obj = (X) environmentCon.lookup("X");

There can be various ways to obtain the resource. But there are several problems in this approach.

Problems of Dependency Lookup

There are mainly two difficulties of dependency lookup approach.

Tight coupling -

The dependency lookup approach makes the code tightly coupled. If resource is changed, we have to perform a lot many changes in the code.

Not easy for testing -

This approach creates many problems while testing the application especially while performing black box testing.

Constructor-based Dependency Injection

Using constructor-based dependency injection, the container will raise a constructor with arguments each representing a dependency that we want to fix.

The Spring resolves each argument primarily by type, followed by name of the attribute and index for clarity. See the below example of the configuration of a bean and its dependencies using annotations:

```
@Configuration
public class Application {

    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

The @Configuration annotation indicates that the class is a source of bean definitions. Also, it is possible to add it to multiple configuration classes.

The @Bean annotation is used on a method for a bean definition. If we don't specify a custom name, the bean name will default to the method name.

For a bean with the default singleton scope, Spring first checks if a cached instance of the bean already exists and only creates a new one if it is not existing. If we're using the prototype scope, the container returns a new bean instance for each of the method call.

Following is one more way to create the configuration of the beans is through XML configuration:

```
<bean id="item1" class="org.baeldung.store.ItemImpl1" />
<bean id="store" class="org.baeldung.store.Store">
    <constructor-arg type="ItemImpl1" index="0" name="item" ref="item1" />
</bean>
```

Constructor Injection is the process of injecting the dependencies of an object through its constructor argument at the time of instantiating it. In other words, we can say that dependencies are supplied as an object through the object's own constructor. The bean definition

n can use a constructor with any number of arguments to initiate the bean, as shown here:

```
public class EmplServiceImpl implements EmplService
{
    private EmplDao emplDao = null;
    public EmplServiceImpl(EmplDao emplDao)
    {
        this.emplDao = emplDao;
    }
}
```

In the preceding code, the object of the EmplDao emplDao type is injected as a constructor argument to the EmplServiceImpl class. We need to configure bean definition in the configuration file that will do Constructor Injection. The Spring bean XML configuration tag `<constructor-arg>` is used for Constructor Injection:

```
...
<bean id="emplService"
    class="org.mitu.Spring.second.dependencyinjection.EmplServiceImpl">
    <constructor-arg ref="emplDao" />
</bean>
<bean id="emplDao"
    class="org.mitu.Spring.second.dependencyinjection.EmplDaoImpl">
</bean>
...
```

In the above pseudo-code, there is a Has-A relationship between the classes, which is EmplServiceImpl HAS-A EmplDao . Here, we inject a user-defined object as the source bean into a target bean using Constructor Injection. Once we have the emplDao bean to inject it into the target emplService bean, we need another attribute called ref — its value is the name of the ID attribute of the source bean, which in our case is "emplDao" .

The `<constructor-arg>` element

The `<constructor-arg>` sub-element of the `<bean>` element is used for creating the Constructor Injection. This tag element supports four attributes. They are explained in the following table:

Table 5.1 The property attributes

Attributes	Description	Occurrence
index	It takes the exact in the constructor argument list. It avoids the ambiguities like when two arguments are having same type.	Optional
type	This takes the type of this constructor argument	Optional
value	It describes the content in a simple string representation, which is then converted to the type using PropertyEditors JavaBeans	Optional
ref	It refers to another bean in this factory	Optional

Constructor Injection – injecting simple Java types

We inject simple Java types into a target bean using the Constructor Injection. The Empl class has emplName as String , emplAge as int , and married as boolean. The constructor initializes all these three fields. Following is the Empl.java file:

```
package org.mitu.Spring.second.constructioninjection.simplejavatype;
public class Empl
{
    private String emplName;
    private int emplAge;
    private boolean married;
    public Empl(String emplName, int emplAge, boolean married)
    {
        this.emplName = emplName;
        this.emplAge = emplAge;
        this.married = married;
    }
    @Override
    public String toString()
    {
        return "Empl Name: " + this.emplName + " , Age:" + this.emplAge + " , IsMarried: " +
            married;
    }
}
```

Following is beans.xml file:

```
...
<bean id="empl" class="org.mitu.Spring.second.constructioninjection.simplejavatype.Empl">
    <constructor-arg value="Rashmi Thorave" />
</bean>
```



```

    <constructor-arg value="28" />
    <constructor-arg value="True" />
</bean>
...

```

Constructor Injection with resolving ambiguity

In the Spring Framework, whenever we create a Spring bean definition file and provide values to the constructor, Spring decides implicitly and assigns the bean's value in the constructor by means of following main factors:

- Matching the number of parameters
- Matching the argument's data type
- Matching the argument's sequence

Whenever Spring tries to create bean using Construction Injection by following the mentioned rules, it tries to resolve the constructor to be chosen while creating Spring bean and hence results in the following circumstances.

No ambiguity

When Spring tries to create a Spring bean using the preceding rule if no matching constructor is found, it throws `BeanCreationException` exception with the message: Could not resolve matching constructor .

We will understand this scenario in more detail by taking the `Empl` class from earlier, which has three instance variables and a constructor to set the value of this instance variable.

The `Empl` class has a constructor in the order of `String`, `int`, and `boolean` to be passed while defining the bean in the definition file. In the `beans.xml` file, you'll have the following code:

```

...
<bean id="empl" class="org.mitu.Spring.second.constructioninjection.simplejavatype.Empl">
<constructor-arg value="Rashmi Thorave" />
<constructor-arg value="True" />
<constructor-arg value="28" />
</bean>
...

```

If the orders in which constructor-arg is defined are not matching, then we will receive the following exception:

Exception in thread "main" org.springframework.beans.factory.
UnsatisfiedDependencyException:

Error creating bean with name empl defined in the classpath resource [beans.xml]: Unsatisfied dependency expressed through constructor argument with index 1 of type [int]: Could not convert constructor argument value of type [java.lang.String] to required type [int]: Failed to convert value of type 'java.lang.String' to required type 'int'; nested exception is java.lang.NumberFormatException: For input string: "True"

Solution

The solution to this problem is to fix the order of elements sent. Either we modify the constructor-

arg order of the bean definition file or we use the index attribute of constructor-arg as follows:

```
...
<bean id="empl"
class="org.mitu.Spring.second.constructioninjection.simplejavatype.Empl">
<constructor-arg value="Rashmi Thorave" index="0" />
<constructor-arg value="True" index="2" />
<constructor-arg value="28" index="1" />
</bean>
...
```

Remember that the index attribute in all collection elements always starts with 0.

Parameter ambiguity

Sometimes, there is no problem in resolution of the constructor, but the constructor chosen is leading to in-

convertible data. In this case, org.springframework.beans.factory.UnsatisfiedDependencyException is thrown just before the data is converted to the actual data type.

We will understand this scenario in more detail; the Empl class contains two constructor methods and both accept three arguments with different data types.

The following code snippet is also present in Empl.java :

```
package org.mitu.Spring.second.constructioninjection.simplejavatype;
public class Empl
{
    private String emplName;
    private int emplAge;
    private String emplId;
    Empl(String emplName, int emplAge, String emplId)
    {
        this.emplName = emplName;
        this.emplAge = emplAge;
        this.emplId = emplId;
    }
}
```

```

Empl(String emplName, String emplId, int emplAge)
{
    this.emplName = emplName;
    this.emplId = emplId;
    this.emplAge = emplAge;
}
@Override
public String toString()
{
    return "Empl Name: " + emplName + ", EmplAge: " + emplAge + ", Empl Id: " + e
mplId;
}
}

```

The beans.xml file, will be like this:

```

...
<bean id="empl" class="org.mitu.Spring.second.constructioninjection.simplejavatype.Empl">
<constructor-arg value="Rashmi Thorave" />
<constructor-arg value="534" />
<constructor-arg value="28" />
</bean>
...

```

Spring chooses the wrong constructor to create the bean. The preceding bean definition has been written in the hope that Spring will choose the second constructor as Rashmi Thorave for emplName , 534 for emplId , and 28 for emplAge . But the actual output will be:

Empl Name: Rashmi Thorave, Empl Age: 534, Empl Id: 28

The preceding result is not the expected result; the first constructor is run instead of the second constructor. In Spring, the argument type 534 is converted to int, so Spring converts it and takes the first constructor even though we assume it should be a string. In addition, if Spring can't resolve which constructor to use, it will prompt the following error message:

constructor arguments specified but no matching constructor found in bean 'CustomerBean' (hint: specify index and/or type arguments for simple parameters to avoid type ambiguities)

Solution

The solution to this problem uses the type attribute to specify the exact data type for the constructor:

```
...
<bean id="empl"
class="org.mitu.Spring.second.constructioninjection.simplejavatype.Empl">
<constructor-arg value="Rashmi Thorave" type="java.lang.String"/>
<constructor-arg value="534" type="java.lang.String"/>
<constructor-arg value="28" type="int"/>
</bean>
...
```

The output will be printed as:

Empl Name: Rashmi Thorave, Empl Age: 28, Empl Id: 534

The setter-based Dependency Injection

Setter Injection in Spring is a type of dependency injection in which the framework injects the dependent objects into the client by the setter method. The container first calls the default argument constructor and then calls the setters. The setter based injection can work even if some dependencies have been injected using the constructor.

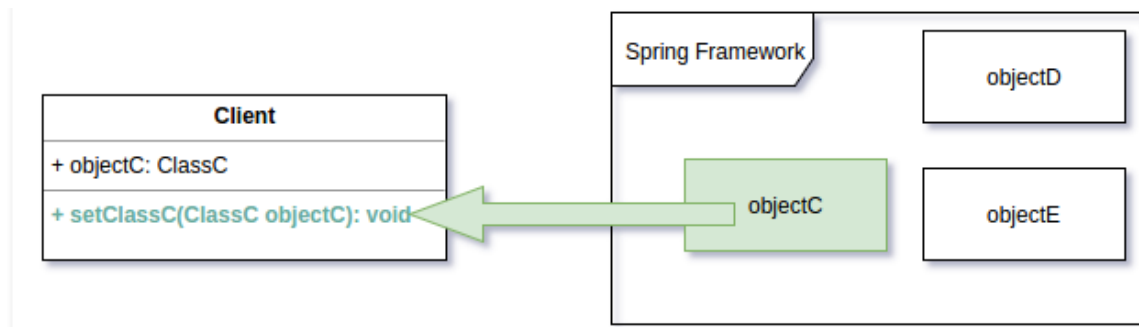


Fig. 5.7 Setter Injection

For setter-based dependency injection, the container will call setter methods of the class, after invoking a default constructor or no-argument static factory method to instantiate the bean. We will create this configuration using annotations:

```
@Bean
public Store store()
{
```

```

Store store = new Store();
store.set(item1());
return store;
}

```

We can also use XML for the same configuration of the beans:

```

<bean id="store" class="org.baeldung.store.Store">
  <property name="item" ref="item1" />
</bean>

```

Constructor-based and setter-

based types of injection can be combined together for the same bean. The Spring documentation recommends using constructor-based injection for compulsory dependencies, and setter-based injection for optional ones.

The setter-

based D.I. is the method of injecting the dependencies of an object using the setter method. In setter injection, the Spring container uses set() of the Spring bean class to assign a dependent variable to the bean property from the bean configuration file. The setter method is more convenient to inject more dependencies since a large number of constructor arguments makes it clumsy.

In the EmplServiceImpl.java class, you'll find the following code:

```

public class EmplServiceImpl implements EmplService
{
    private EmplDao emplDao;
    public void setEmplDao(EmplDao emplDao)
    {
        this.emplDao = emplDao;
    }
}

```

In the EmplDaoImpl.java class, we will find the following code:

```

public class EmplDaoImpl implements EmplDao
{
    // ...
}

```

In the given pseudo-

code, the EmplServiceImpl class defined the setEmplDao() method as the setter method where EmplDao is the property of this class. This method injects values of the emplDao bean from the bean configuration file before making the emplService bean available to the application.

The Spring bean XML configuration tag `<property>` is used to configure properties. The `ref` attribute of property elements is used to specify the reference of another bean.

In the `beans.xml` file, will have the following code:

```
...
<bean id="emplService" class="org.mitu.Spring.second.dependencyinjection.EmplServiceImpl">
<property name="emplDao" ref="emplDao" />
</bean>
<bean id="emplDao" class="org.mitu.Spring.second.dependencyinjection.EmplDaoImpl">
</bean>
...
```

The `<property>` element

The `<property>` element invokes the setter method. The bean definition can be describing the zero or more properties to inject before making the bean object gettable in the application. The `<property>` element tallies to JavaBeans' setter methods, which are exposed by bean classes. The `<property>` element supports the following three attributes:

Table 5.2 The property element attributes

Attributes	Description	Occurrence
name	It takes the name of JavaBean based property	Optional
value	The value describes the content in a simple string representation, which will be converted into the argument type using Java Bean PropertyEditors	Optional
ref	It store reference to a bean	Optional

Setter Injection with simple Java type

Here, we inject string-based values using the setter method. The `Empl` class contains the `emplName` field with its setter method.

In the `Empl.java` class, we will have the following code:

```
package org.mitu.Spring.second.setterinjection;
public class Empl
{
    String emplName;
    public void setEmplName(String emplName)
```

```

{
this.emplName = emplName;
}
@Override
public String toString()
{
return "Empl Name: " + emplName;
}
}

```

In the beans.xml file, we will have the following code:

```

...
<bean id="empl" class="org.mitu.Spring.second.setterinjection.Empl">
<property name="emplName" value="Rashmi Thorave" />
</bean>
...

```

In this pseudo-code, the bean configuration file set the property value. In the Payroll.java class, lets have the following code.

```

package org.mitu.Spring.second.setterinjection;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Payroll
{
    public static void main(String[] args)
    {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("beans.xml");
        Empl empl = (Empl)
        context.getBean("empl");
        System.out.println(empl);
    }
}

```

The output after running the Payroll class will be as follows:

```

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext
@1ba94d: startup date [Sun Jun 28 10:11:36 IST 2019]; root of context hierarchy
Jan 25, 2015 10:11:36 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beans.xml]
Empl Name: Rashmi Thorave

```

Setter Injection for injecting collections

In the Spring IoC container, beans can also access collections of the objects. Spring allows us to inject a collection of objects in a bean using Java's collection framework. Setter Injection can be used to inject collection of values into the Spring Framework. If we have a dependent object in the collection, we can inject this information using the ref element in the list, set, or map. See the below elements.

- **<list>** : It describes a java.util.List type. A list can contain multiple bean , ref , value , null , another list , set , and map elements. The required conversion is automatically performed by BeanFactory.
- **<set>** : It describes a java.util.Set type. A set will have multiple bean , ref , value , null , another set , list , and map elements.
- **<map>** : It describes a java.util.Map type. A map can contain zero or more <entry> elements, which describes a key and value.

The Empl class is a class with an injecting collection. In the Empl.java class, we will have the following code:

```
package org.mitu.Spring.second.setterinjection;
import java.util.List;
import java.util.Map;
import java.util.Set;
public class Empl
{
    private List<Object> lists;
    private Set<Object> sets;
    private Map<Object, Object> maps;

    public void setLists(List<Object> lists)
    {
        this.lists = lists;
    }
    public void setSets(Set<Object> sets)
    {
        this.sets = sets;
    }
    public void setMaps(Map<Object, Object> maps)
    {
        this.maps = maps;
    }
}
```

The bean configuration file of this code is the one that injects each and every property of the Empl class.

In the beans.xml file, we will have the following code:


```

...
<bean id="empl" class="org.mitu.Spring.second.setterinjection.Empl">
<property name="lists">
<list>
  <value>Rashmi Thorave</value>
  <value>Aniket Thorave</value>
  <value>Rahul Thorave</value>
</list>
</property>
<property name="sets">
<set>
  <value>Vihaan Thorave</value>
  <value>Pravina Thorave</value>
</set>
</property>
<property name="maps">
  <map>
    <entry key="Key 1" value="Shiroli Bk"/>
    <entry key="Key 2" value="Maharashtra"/>
  </map>
</property>
</bean>
...

```

In this pseudo code, we have injected values of all three setter methods of the Empl class. The List and Set instances are injected with the <list> and <set> tags. For the map property of the Empl class, we injected a Map instance using the <map> tag. Each entry of the <map> tag is specified with the <entry> tag that contains a key-value pair of the Map instance.

Difference between Constructor Injection and setter injection -

There are many differences between constructor injection and setter injection. Some prime differences are as listed below.

Partial dependency: can be injected using setter injection but it is not affirmative by constructor. Suppose there are three properties in a class, having three argument constructor and setters methods. In such case, if we want to pass information for only one property, it is possible by setter method only.

Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.

Changes: we can easily change the value by setter injection. it doesn't create a new bean instance always like constructor. so setter injection is flexible than constructor injection.

Exercises:

1. What is the application of inversion of control in Spring? Explain with detailed description.
2. What do you mean by constructor based dependency injection. Illustrate with a practical application in Java.
3. How to declare a bean in XML configuration file. Explain with a Java application program.
4. Explain the detailed procedure of creating an enterprise java bean.
5. Differentiate between constructor injection and setter injection

Objective Questions:

1. Which among the below type contains the key-value paired data?
 - a. List
 - b. Tuple
 - c. Map
 - d. Set
2. What kind of annotation is used to denote the Component?
 - a. @Compoment
 - b. @Comp
 - c. @CC
 - d. @Container

Unit 6

Spring Web MVC Framework

Introduction

The presentation layer in an enterprise application is the front door to your application created in Java. It provides users a interactive view of the information, allowing them to perform business functions provided and managed by the application. The development of the presentation layer is a difficult task these days because of the rise of cloud computing and different kinds of devices that people are using on regular basis. Many technologies and frameworks are now getting used to develop enterprise web applications, such as Spring Web MVC, Java Server Faces (JSF), Struts, Google Web Toolkit (GWT), as well as as jQuery. These provide rich component libraries that can help develop the interactive web front-ends. Many frameworks also provide widget libraries and tools targeting mobile devices, including tables and smart-phones.

6.1 The Spring Web Model View Controller (MVC)

The Spring Web Model View Controller (MVC) framework supports web application development by providing broad as well as intensive support for the Java application. The framework is flexible, robust, and well-designed and is used to develop web applications. It is designed in such a way that development of a web application is highly configurable into Model, View, and Controller. In the MVC design pattern, Model represents the information or data of a web application; View represents the User Interface (UI) components by which user interacts with it, such as checkbox, textbox, radio buttons and so forth that are used to display web pages; and Controller processes as per the user request. The Spring MVC framework helps in integrating other frameworks, such as Struts and WebWork, with a Spring application. This framework also supports the integration of other view technologies such as Java Server Pages (JSP), FreeMarker, Tiles, and Velocity in a Spring web application.

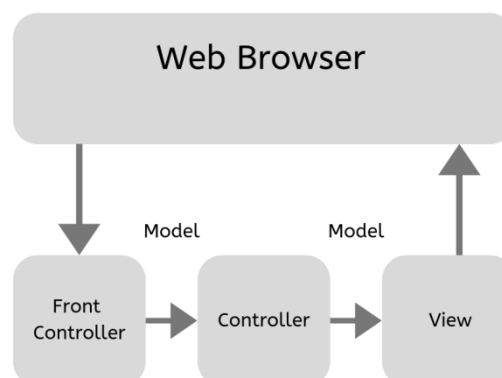


Fig. 6.1 Spring MVC Architecture

Model It contains the data of the application. A data can be a single object or a collection of objects.

Controller –

It contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

View –

It represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

Front Controller –

In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Flow of Spring MVC

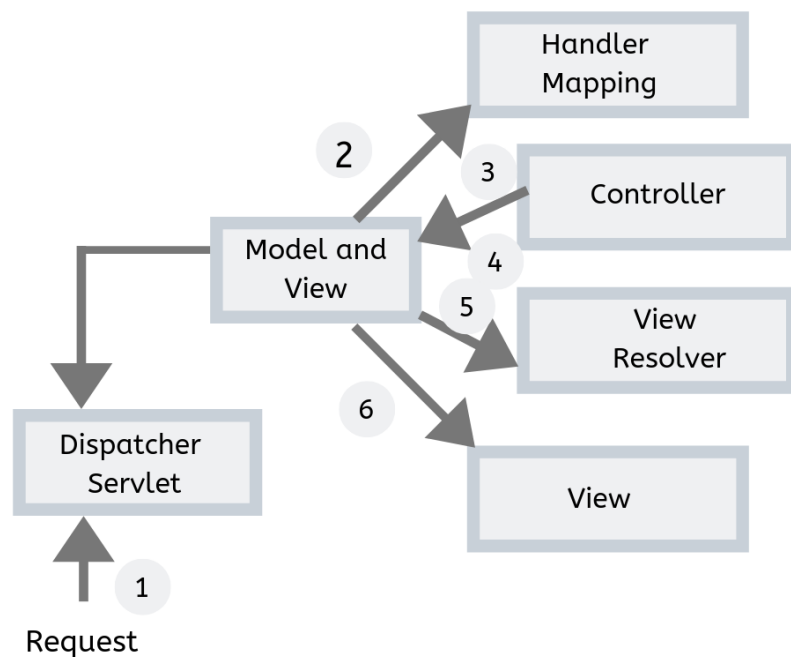


Fig. 6.2 Flow of Spring MVC

As displayed in the figure above, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.

The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.

The controller returns an object of ModelAndView.

The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

Advantages of using the Spring MVC Framework

Following are the advantages of Spring MVC Framework.

Isolated roles -

The Spring MVC isolates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, can be fulfilled by a specialized object.

Light-weight - This uses light-

weight Servlet container to develop and deploy your application.

Mighty Configuration -

It provides a robust configuration for both framework and application classes which includes easy referencing across contexts, such as from web controllers to business objects and validators.

Fast development - The Spring MVC facilitates rapid and parallel development.

Reusable business code -

Instead of creating new objects, it allows us to use the existing business objects.

Effortless to test -

In Spring, generally we create JavaBeans classes that enables us to inject test data using the setter methods.

Flexible Mapping - It provides the specific annotation that easily redirects the page.

Spring Web MVC Framework Example

Let's see the simple example of a Spring Web MVC framework.

The steps are as follows:

- Load the spring's jar files or add dependencies in the case of Maven
- Create the controller class
- Provide the entry of controller in web.xml file
- Define the bean in the separate XML file
- Display the message in the JSP page created
- Start the server and then deploy the project

6.2 The MVC architecture

MVC is an architectural pattern used in the development of web applications; it provides separation of concern in the architecture of an application and separates it into three software modules which communicate with each other using a relatively easy interface. This model holds the business entities that can be passed to the View via Controller to have exposure them to the end user. The View is not dependent on the Model and Controller; it represents the presentation form of an application.

The Controller is independent of the Model and View with the exclusive purpose of handling requests and performing business concern logic. Thus, the model (business entities), controllers (business logic), and views (presentation logic) lie in logical/physical layers, independent of each other. The presentation layer of an application is normally implemented using the MVC patterns. MVC offers more organized and reparable code. It is popularly known as a software design pattern used to develop the web applications.

The prime three components of MVC are as listed below.

- **Model:** The Model represents the business entity where the application's data is stored. It is the formulation of the objects that the user works with and the mapping of those concepts into data structures: the user model and data model.
- **View:** The View is obligated for preparing the presentation for the client based on the conclusion of the request processing, without considering any business logic. It renders the model data into the client's user interface type.
- **Controller:** The Controller is responsible for controlling the flow request to response flow in the middle-ware. It starts back-end services for businesses after receiving a request from the user, and updates the model. It also prepares models for the View to present. It is answerable for determining which view should be rendered.

The following figure illustrates Model, View, and Controller:

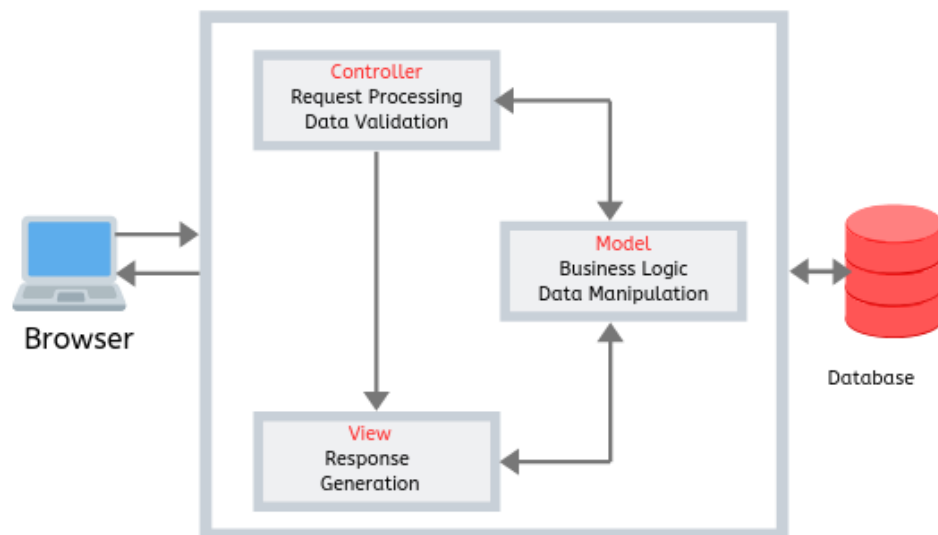


Fig 6.3 The MVC Architecture

The above figure shows MVC in a web application. The Controller is typically used to process requests from the client and forward requests for changes to the Model. The View code accesses the Model to render the response to the client.

6.3 Front Controller Design Pattern

A pattern represents the strategies that permit programmers to share their knowledge regarding recurring problems and their answers. As we have seen in the previous section, the MVC pattern separates the user interface logic from the business logic of web applications. When we wish to achieve re-usability and flexibility while avoiding redundancy and decentralization, we should structure the controller for a very complex web application in the best possible manner.

The Front Controller is used at the initial point of contact to handle all Hyper Text Transfer Protocol (HTTP) requests; it enables us to centralize logic to debar duplicated code, and manages the key HTTP request-handling activities, such as navigation and routing, dispatch, and context transformation. The front controller design pattern enables centralizing the handling of all HTTP requests without limiting the number of handlers in the system.

The Front Controller does not just capture HTTP requests; it also initializes some of the very important components of the framework to run, as shown in the following figure. It helps in loading the map of URLs and the components that need to be invoked when a request lands with the URLs. It can also load some of the other components, such as views.

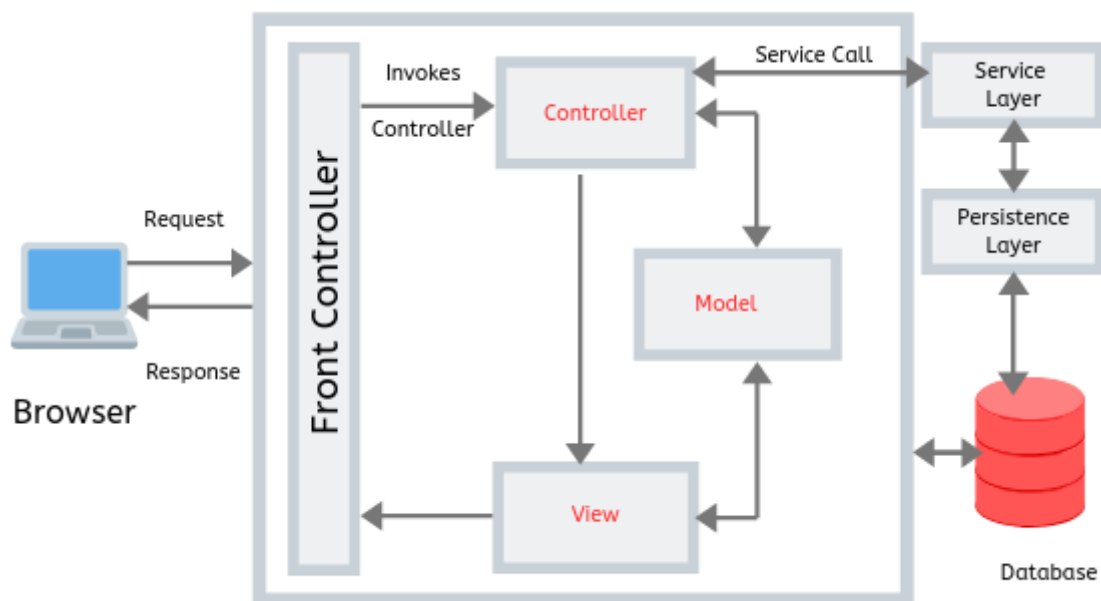


Fig. 6.4 Front controller design pattern

The above figure illustrates the front controller design pattern in web applications. The user or browser will interact with only one controller, which is front controller. The front controller intercepts the user request, does common operations, and dispatches the request to the respective controller based on web application configuration and HTTP request information. The controller then interacts with the service layer to do business logic and persistence logic. Then it updates the model, and the view renders the model data to produce pr

resentation view and return the view to the user. The front controller responds to the client in the form of the view.

In Spring MVC, the Dispatcher Servlet does the task of a front controller. As we know about the MVC and the front controller, which are the important to understand the Spring MVC framework, starting with Spring MVC framework followed by its architecture and its elements.

Spring MVC

A web application created using the Spring MVC framework is easier to develop, understand, and keep the maintenance. Spring MVC is an open source web framework; which allows us to download the source code and modify it to support user extensions as per the user requirements. Its code is exposed to the developer and this enables rapid development and maintenance cycle. As a result, we can have a quick result from the Spring team in fixing the bugs in code and responding to new requirements in the business market.

The Spring MVC framework is implemented using standard Java technologies such as Java Servlet and JSP. Thus, we are allowed to host Spring MVC projects on any Java enterprise web server just by adding the Spring JAR files into the lib of our web application or project.

The Spring MVC module in the Spring Framework provides a big support for the MVC design for features such as internationalization, theming, validation, and so on, to make easier implementation of the presentation layer.

The Spring MVC framework is configured around a DispatcherServlet. The DispatcherServlet dispatches the HTTP request to the handler, which is a very plain Controller interface.

Spring MVC allows us to use any form object or command object. Struts built around needed base classes such as Action class and ActionForm class; however, the Spring MVC application doesn't need to utilize a framework-specific interface or base class.

Features of the Spring MVC framework

The Spring MVC framework provides a set of the following web support features:

- **Powerful framework configuration and application classes:** The Spring Web MVC framework provides straightforward and powerful configuration of the framework as well as of application classes such as JavaBeans.
- **Easier testing:** Most of the Spring classes are designed as JavaBeans, which allows us to inject the test data using the setter method of these JavaBeans classes. The Spring MVC framework also provides classes to handle HTTP requests, which make the unit testing of the web application much easier.
- **Separation of roles:** Each component of a Spring MVC framework does a different role during request handling. A request is handled by components such as the Controller, Validator, Model Object, View Resolver and HandlerMapping interfaces. The whole work is dependent on these components and provides a clear isolation of roles.

- **No need for the duplication of code:** In the Spring MVC framework, we can use the existing business code in any component of the Spring MVC application. Therefore, no duplication of the code arises in a Spring MVC application.
- **Specific validation and binding:** Validation errors are displayed when any unmatched data is entered in a form.

6.4 Flow of request handling in Spring MVC

The `DispatcherServlet` is the head-on controller for the Spring MVC application, which provides centralized access to the application for various requests and collaborating with various other objects to complete the request handling and present the response to the client. In Spring MVC, the `DispatcherServlet` gets requests and dispatches requests to the appropriate controller. There can be any number of `DispatcherServlet` in a Spring application to manage user interface requests or Restful Web Services requests, as shown in the following figure. Each `DispatcherServlet` uses its own `WebApplicationContext` configuration to situate the various objects registered in the Spring container, such as the controller, handler mapping, view resolving, i18n, theming, type conversion and formatting, validation, and many more.

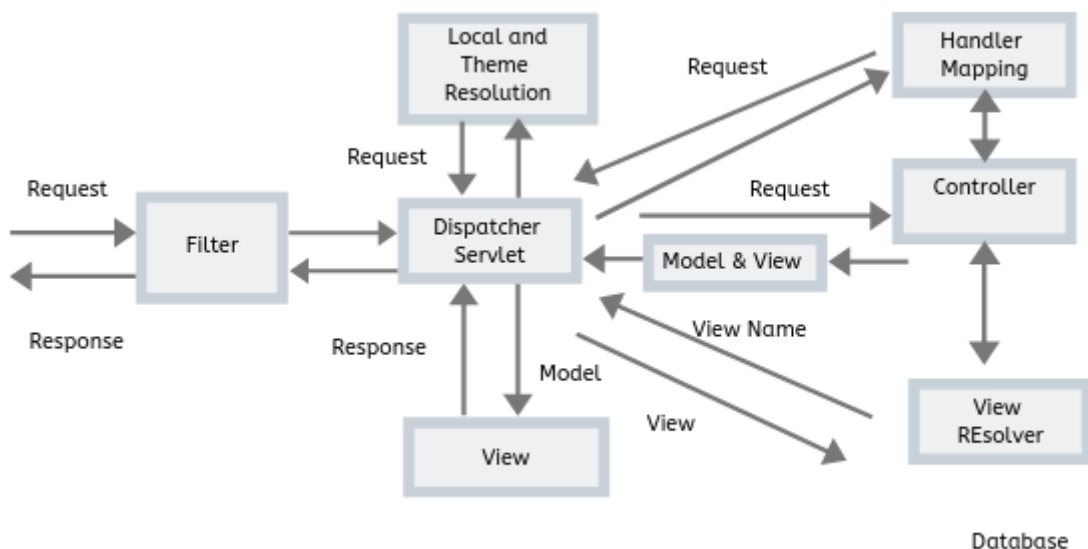


Fig. 6.5 Request handling in Spring MVC

The figure above shows the flow of request handling in Spring MVC, along with its components. They are explained as below.

- **Filter:** The Filter component applies to all the HTTP requests. Various commonly used filters can be used to fulfill the HTTP requests.
- **DispatcherServlet:** The Servlet intercepts and analyzes the incoming HTTP request and dispatches them to the suitable controller to be processed. It is always configured in the `web.xml` file of any web application.

- **Local resolution and theme resolution:** The configuration of i18n and themes is defined in DispatcherServlet file's WebApplicationContext . It provides support to all the requests.
- **Handler mapping:** This maps the HTTP request to the handler, that is, a method within a Spring MVC controller class, based on the HTTP paths declared through the @RequestMapping annotation at the method or type level inside the controller class.
- **Controller:** The Controller in Spring MVC acquires requests from the DispatcherServlet class and does some business logic in accordance with the client.
- **ViewResolver:** The ViewResolver is the interface of Spring MVC supports view resolution based on the view name returned by controller. The URLBasedViewResolver class supports the direct resolution of view name to URLs. The ContentNegotiatingViewResolver class supports the dynamic resolution of views based on the media type supported by the client, such as PDF, XML, JSON, and many more.
- **View:** In Spring MVC, the View components are user-interface elements, such as textbox items and many others, which are responsible for displaying the output of a Spring MVC application. Spring MVC provides a set of tags in the form of a tag library, which is used to construct views. Whenever an HTTP request from a browser comes to a Spring MVC application, it is first intercepted by DispatcherServlet , which acts like the front controller for a Spring MVC application. The DispatcherServlet class intercepts the incoming HTTP request and determines which controller handles the request, and then sends the HTTP request to a Spring MVC controller.
- **The controller** implements the activity of the Spring MVC application. The controller gets the request from the DispatcherServlet class and does some business logic in agreement with the client request. A Spring MVC application may have various controllers, and to determine on the controller to send the request, DispatcherServlet takes assist from one or more handler mappings. The handler mapping makes its decision based on the URL transferred by the request. After the business logic is performed by controller, some information referred to as the model is generated, that needs to be carried back to the client and display in the browser. But it is not adequate to send raw information to the client. So the raw information required to be given to the view, which can be JSP or similar tool. The Controller also packs up the model data and identifies the view name that will render the output. Then, it sends the request along with view name and model back to DispatcherServlet.
- **The DispatcherServlet** class enquires the view resolver to map the view name to a specific view implementation, which may or may not be JSP, FreeMarker, JSON, Thymeleaf, and many more. A good point here is that Spring is nescient of the view technology. So, at this point, the request job is almost over and DispatcherServlet knows about the view which will provide the result. It delivers the model collection to the view component, and the request job is in the end done here. This model data will be used by the view to render the output, which will be carried back by the response object to the client.

Front Controller Design Pattern

The front controller design pattern refers that all requests that come for a resource in an application will be handled by a single handler and then dispatched to the suitable handler for that type of request. The main front controller may use other helpers to accomplish the dispatching mechanics.

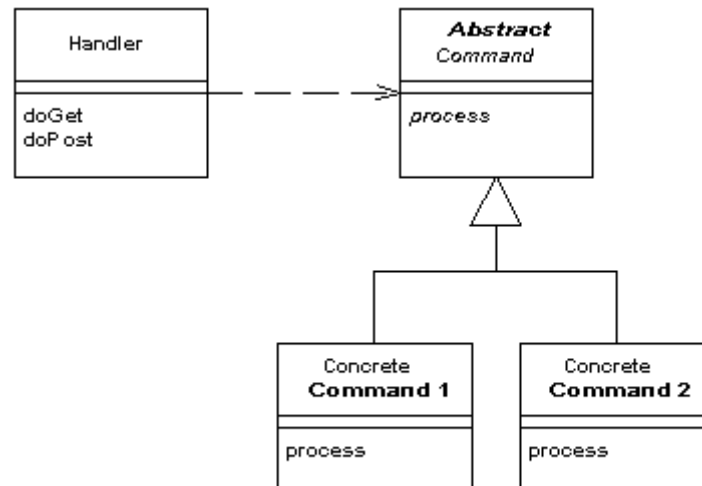


Fig. 6.6 Front Controller Design Pattern

The Design components

Controller : The controller is the first contact point for handling all requests in the system. The controller may delegate to a helper to finish authentication and authorization of a user or to begin contact retrieval.

View: A view represents and shows information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data a model for using in the display.

Dispatcher: A dispatcher is answerable for view management and navigation, management of the choice of the next view to represent to the user, and providing the mechanics for vectoring control to that resource.

Helper : A helper is responsible for helping a view or controller complete its processing. So, helpers have many responsibilities, including gathering data needed by the view and storing this intermediate model, in which case the helper is erstwhile referred to as a value bean.

Let's see an example of Front Controller Design Pattern.

```
class TView
```

```
{
```

```
    TView
```

```
public void show()
{
    System.out.println("Teacher View");
}
}
```

```
class SView
{
    public void show()
    {
        System.out.println("Student View");
    }
}
```

```
class Dispatch
{
    private SView SView;
    private TView TView;
```

```
public Dispatch()
{
    SView = new SView();
    TView = new TView();
}
```

```
public void dispatch(String request)
{
    if(request.equalsIgnoreCase("Student"))
    {
        SView.display();
    }
    else
    {
```

```
    TView.display();  
}  
}  
}
```

```
class FrController  
{  
    private Dispatch Dispatch;  
  
    public FrController()  
    {  
        Dispatch = new Dispatch();  
    }  
  
    private boolean isAuthenticatedUser()  
    {  
        System.out.println("Authentication is successful.");  
        return true;  
    }  
  
    private void trackRequest(String request)  
    {  
        System.out.println("Requested View: " + request);  
    }  
  
    public void dispatchRequest(String request)  
    {  
        trackRequest(request);  
  
        if(isAuthenticatedUser())  
        {  
            Dispatch.dispatch(request);  
        }  
    }  
}
```

```

    }
}

class FrControllerPattern
{
    public static void main(String[] args)
    {
        FrController frController = new FrController();
        frController.dispatchRequest("Teacher");
        frController.dispatchRequest("Student");
    }
}

```

Output is a shown below:

Requested View: Teacher

Authentication successful.

Teacher View

Requested View: Student

Authentication successful.

Student View

Benefits:

Focused control : The Front controller handles all the requests to the Web application. This implementation of focused control that ignores using multiple controllers is desirable for imposing application-wide policies such as users tracking and safety.

Thread safety : A new command object arises when receiving a new request and the command objects are not meant to be thread-safe. Thus, it will be safe in the command classes. Although safety is not guaranteed when threading issues are collected, codes that act with the command are still thread safe.

Drawbacks:

- It is not possible to scale an application using a front controller. The scaling up the application requires additional programming.
- It has several performance issues. Performance is better if you deal with a single request uniquely.

6.5 DispatcherServlet in Spring MVC

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a centralized Servlet that dispatches requests to controllers and provides other functionality that serves the development of web applications. Spring's DispatcherServlet however, does more than just that. It is totally integrated with the Spring IoC container and as such allows us to use every other characteristic that Spring class is having.

The request processing work-

flow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram 6.7. The pattern-

savvy reader can recognize that the DispatcherServlet is an expression of the “Front Controller” design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).

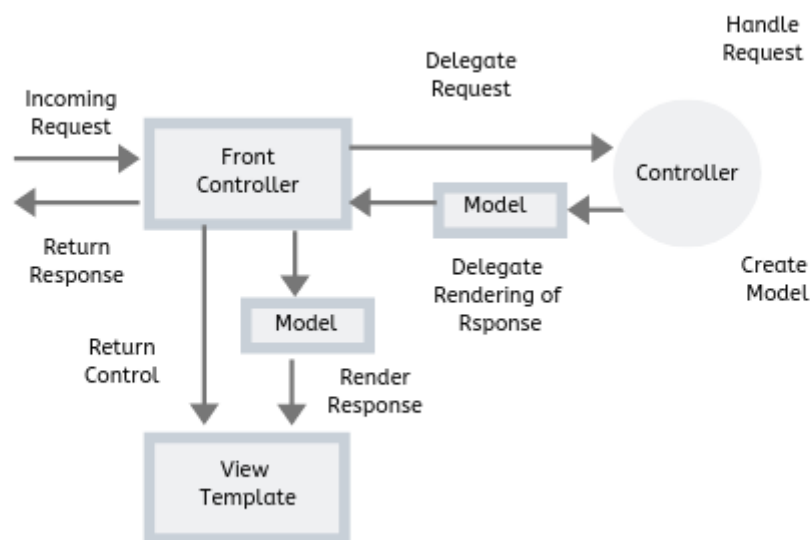


Fig. 6.7 The request processing workflow in Spring Web MVC

The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class of servlet package), and as such is declared in the web.xml of our web application. We need to map requests that we want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file. This is standard J2EE Servlet configuration; the example given below shows such a DispatcherServlet declaration and mapping:

```
<web-app>
  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
```

```

</servlet>

<servlet-mapping>
<servlet-name>example</servlet-name>
<url-pattern>/example/*</url-pattern>
</servlet-mapping>
</web-app>

```

In the given example, all requests starting with /example will be handled by the DispatcherServlet instance named example. In a Servlet 3.0 and ahead environment, we also have the option of configuring the Servlet container by programs. The code shown below is based equivalent of the above web.xml example:

```

public class WebAppInitializer implements WebApplicationInitializer
{
    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }
}

```

WebApplicationInitializer is an interface provided by Spring MVC that ensures our code-based configuration is detected and automatically used to initialize any Servlet 3 container.

An abstract base class implementation of this interface named AbstractDispatcherServletInitializer makes it even simpler to register the DispatcherServlet by just specifying its servlet mapping.

The above code is only the first step in setting up Spring Web MVC. We now need to configure the various beans used by the Spring Web MVC framework (over and above the DispatcherServlet itself).

The ApplicationContext instances in Spring can be scoped. In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already specified in the root WebApplicationContext. These inherited beans can be overridden in the servlet-specific scope, and we can define new scope-specific beans local to a given Servlet instance.

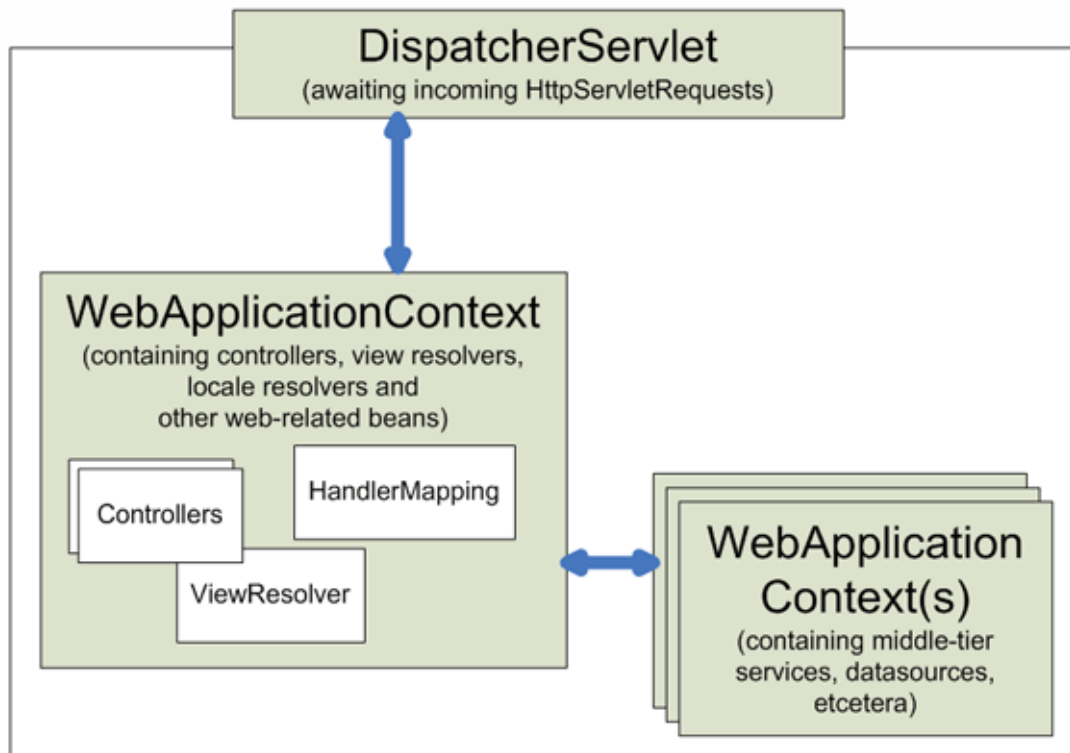


Fig. 6.8 Context hierarchy in Spring web MVC (Ref. docs.spring.io)

The `DispatcherServlet` class of the Spring MVC framework is an implementation of front controller and is a Java Servlet component for Spring MVC applications. It is a front controller class that receives all incoming HTTP client requests for the Spring MVC application. It is also responsible for initializing framework components used to process the request at various stages.

The `DispatcherServlet` class is fully configured with the Inversion of Control

(IoC) container that allows us to use various Spring features such as Spring context, Spring Object Relational Mapping (ORM), Spring Data Access Object (DAO), and many more.

`DispatcherServlet` is a Servlet that handles HTTP requests and is inherited from `HttpServlet` base class.

Configuring `DispatcherServlet` in our Spring web application into the web application deployment descriptor (that is, `web.xml`) is essential, just like any other servlet. Using URL mapping in the configuration file, the HTTP requests to be handled by `DispatcherServlet` are mapped. A Spring MVC application can have any number of `DispatcherServlet` classes and each `DispatcherServlet` class will have its own `WebApplicationContext` .

DispatcherServlet in deployment descriptor

web.xml

For a Java web application, the web deployment descriptor web.xml is the essential configuration file. In web.xml, we define the Servlet for our web application and how the web request should be mapped to them. In the Spring MVC application, we only have to define a single DispatcherServlet instance, which acts as the front end controller for the Spring MVC application, even though we are allowed to define more than one if necessary.

The following pseudo code declares the DispatcherServlet in web.xml :

```
<servlet>
<servlet-name>SpringDispatcher</servlet-name>
<servlet-class>
org.springframework.web.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>SpringDispatcher</servlet-name>
<url-pattern>/**</url-pattern>
</servlet-mapping>
```

In the above pseudo code, SpringDispatcher is the user-defined name of the DispatcherServlet class, which is enclosed with the <servlet-name> element. When this newly created SpringDispatcher class is loaded in a web application, it loads an ApplicationContext from an XML file.

The next task to do after creating the SpringDispatcher class is to map this class with the incoming HTTP request that signals what URLs are handled by the DispatcherServlet class. To map the DispatcherServlet class, we use the <servlet-mapping> element and to handle URLs, we use the <url-pattern> tag in the web.xml file, as seen in the preceding pseudo code.

The /** (slash with **) pattern doesn't express any specific type of response and simply indicates that DispatcherServlet will serve all incoming HTTP requests, including the request for any static content.

Registering Spring MVC configuration file location

As we discussed in the previous topic, DispatcherServlet loads the [servlet-name]-servlet.xml file in the WEB-INF

folder to compose WebApplicationContext. In order to define this file as a random file in a random location, or as a multi-file we use <init-param> under <servlet> to define an initialization parameter named contextConfigLocation.

```
<init-param>
```

```

<param-name>contextConfigLocation</param-name>
<param-value>/config/springmvc/someCommon-servlet.xml,
/config/springmvc/someUser-servlet.xml</param-value>
</init-param>

```

Spring configuration – SpringDispatcher, servlet.xml

By default, when the DispatcherServlet class is loaded, it loads the Spring application context from the XML file whose name is based on the name of the Servlet. In the previous code, as the name of the Servlet has been defined as SpringDispatcher, DispatcherServlet will try to load the application context from a file named SpringDispatcher-servlet.xml located in the application's WEB-INF directory.

The DispatcherServlet class will use the SpringDispatcher-servlet.xml file to create an ApplicationContext, which is a standard Spring bean configuration file, as shown below.

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-3.0.xsd http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
<mvc:annotation-driven />

<context:component-scan base-package="org.mitu.Spring.third.springmvc" />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/views/" />
<property name="suffix" value=".jsp" />
</bean>
</beans>

```

Let's take a look at some of the MVC features used in the pseudo code above.

`<mvc:annotation-`

`driven/>` : This tells the Spring Framework to support annotations like `@Controller` , `@RequestMapping` , and others, all of which simplify the writing and configuration of controllers.

`InternalResourceViewResolver` : The Spring MVC framework supports various types of views for presentation technologies, including JSPs, HTML, PDF, JSON, and many more. When the `DispatcherServlet` class defined in the application's `web.xml` file gets a view name returned from the handler, it resolves the logical view name into a view implementation for rendering.

In the above pseudo code, we have configured the `InternalResourceViewResolver` bean to resolve the bean into JSP files in the `/WEB-INF/views/` directory.

`<context:component-`

`scan>` : This tells Spring to automatically detect annotations. It takes the value of the `basePackage` attribute, which corresponds to the one used in the Spring MVC controller.

6.6 Controllers in Spring MVC

The `DispatcherServlet` class delegates the incoming HTTP client request to the controllers to execute the functionality specific to it. The controller interprets user input and converts this input into a specific model which will be represented by the view to the user.

While developing web functionality, we will develop resource-oriented controllers. Rather than each use case having one controller in the web application, we will have a single controller for each resource that the Spring web application serves. An abstract implementation method is provided by Spring for the user to develop the controller without being dependent on a specific API. We don't need to inherit any specific interface or class while developing a controller based on Spring MVC using the `@Controller` annotation.

The `@Controller` annotation to define a controller

The `@Controller` annotation is used to define a class as a controller class without inheriting any interface or class. The following code snippet defines the `EmplController` class as a controller using the `@Controller` annotation:

```
package org.mitu.Spring.third.springmvc.controller;

import org.springframework.stereotype.Controller;

@Controller

public class EmplController

{

    // ...

}
```

The @Controller annotation indicates the role to the annotated class. Such an annotated class is scanned by the dispatcher for mapped methods and finds the @RequestMapping annotation. This defined controller can be automatically registered in the Spring container by adding –

```
<context:component-scan/> in SpringDispatcher-servlet.xml file.  
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:p="http://www.springframework.org/schema/p"  
  xmlns:context="http://www.springframework.org/schema/context"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans.xsd  
    http://www.springframework.org/schema/context  
    http://www.springframework.org/schema/context/spring-context.xsd">  
  <context:component-scan base-package="org.mitu.Spring.third.springmvc"/>  
  <!-- ... -->  
</beans>
```

Exercises:

1. Explain the MVC architecture in detail.
2. What is the roll of controller in Spring MVC. Explain with example.
3. What do you mean by front controller design pattern? Illustrate with diagram.
4. Draw the process of request handling in Spring MVC and explain the same.
5. What is front controller design pattern? Draw the architecture and explain in detail.

Objective Questions

1. Which component represents and shows information to the client?
 - a. View
 - b. Component
 - c. Model
 - d. Demodeler
2. What is the business entity where the application's data is stored?
 - a. Model

- b. View
- c. Controller
- d. Component

Unit 7

Introduction to Java API

Introduction

An e-mail plays an important role in all daily activities in this era of inter-connected networks. For example, we want to get regular updates of a particular feature on a website, by just subscribing to that feature, we will start receiving e-mails regarding these updates. E-mails also allow us to receive notifications, business related communications, or any periodical reports of a producer. Oracle JAVA provides a simple yet mighty API called as a JavaMail API for creating an application to have an e-mail support. This API is actually a set of classes and interfaces for creating the e-mail applications. Writing the program using this API in Java, we can send as well as receive the e-mails, which can be scaled up to work with different protocols associated with the mailing system like POP and SMTP. Being a very powerful API, it is complex, and using the JavaMail API directly in our application is a slightly tiresome work as it involves writing a lot of programming lines.

7.1 What is an API?

API means Application Programming Interface which is a collection of communication protocols and subroutines used by various programs of language to convey data among them. A programmer can use various Application Programming Interface tools to make its program easy. Also, an API facilitates the programmers with an efficient way to develop their programs.

Thus in simple term, an Application Programming Interface helps two programs or applications to communicate with each other by providing them with essential tools as well as functions and methods. It receives the request from the user and transmits it to the service provider and again sends the result generated from the service provider to the desired user.

The developers use Application Programming Interfaces in their software to implement different features by using an API call without writing the complicated codes for it. We can write an API for an operating system, database systems, hardware system etc. An API is similar to a GUI (Graphical User Interface) with one prime difference.

Real life example of an API:

For example, we are looking for a hotel room on an online website. In this case, we have a huge number of options to choose from and this may include the hotel's location, the check-in time and check-out dates, cost, accommodation details and many more factors. So in order to reserve the room online, we have to interact with the hotel booking's website which will let us to know if there is a room available on that particular date or not and at what price?

Now in this example, the API is the interface that actually communicates in between programs. It takes the request of the user to the hotel booking's website and in turn returns back the most relevant data from the website to the intended user. We can see from this example how an API works and it has numerous applications in real life from switching on mobile phones to maintaining a huge amount of databases from any part of the world.

There are various kinds of API's available as per their applications and uses like the Browser API which is created for the web browsers to abstract and to return the data from environment or the Third party API's, for which we have to get the codes from other sites on the web like social networking websites.

7.2 Types of APIs

There are basically three types of APIs available as listed and explained below.

1. WEB APIs:

The Web API is also known as Web Services is an enormously used API over the web and can be easily accessed through the HTTP protocols. It is an open source interface and can be used by a huge number of clients using their mobile phones, tablets or desktop computers.

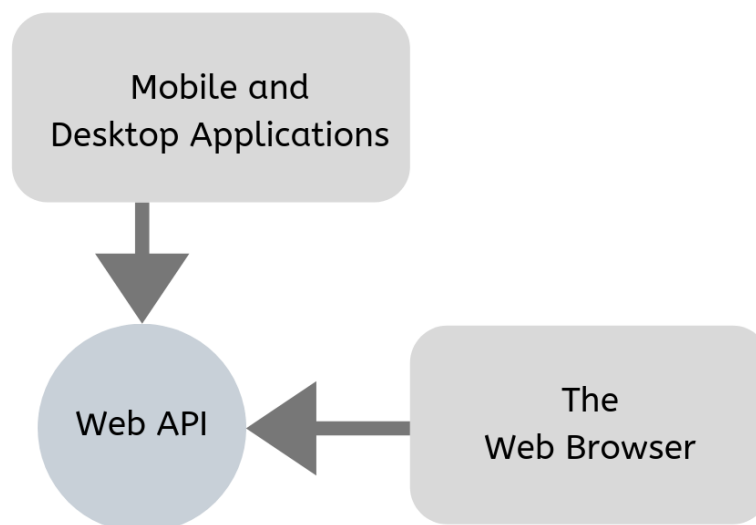


Fig. 7.1 Web APIs

2. LOCAL APIs:

Using the local API, the programmers receives the local middleware services. TAPI (Telephony Application Programming Interface), and .NET are basic examples of Local API's.

3. PROGRAM APIs:

The Program APIs make a remote program appears to be local by making use of RPC's (Remote Procedure Calls). SOAP is a example of this type of API.

Other types of APIs

REST (Representational State Transfer): It uses the HTTP to GET, POST, PUT, or DELETE data. It is basically used to take benefit of the existing data.

SOAP (Simple Object Access Protocol): It is used to define messages in XML format used by web applications to communicate with each other.

XML-

RPC: It is based on XML and uses HTTP for data communication. This API is generally used to exchange information between two or among more networks.

JSON-RPC: It uses JSON for data transfer and is a lightweight remote procedural call (RPC) for defining few data structure types.

These various types and forms of API's largely used over web networks to exchange information and to raise communication among them.

Advantages of using APIs

Efficiency: The API generates efficient, quicker and more useful results than the outputs produced by human beings in an organisation.

Flexible delivery of services: The API provides rapid and flexible delivery of services according to developers necessity.

Integration: The best feature of API is that it allows movement of data between various sites and thus raises coordinated user content.

Automation: API make use of robotic computers rather than human being, it generates better and automated results.

New functionality: While using API the developers find novel tools and functionality for API exchanges.

Disadvantages of APIs

Cost: Development and implementation of the API is costly at times and needs high maintenance and support from developer.

Security: The API adds another layer of surface which is then prone to attacks, and hence the security risk problem occurs in APIs.

7.3 The JavaMail API

The JavaMail API provides platform-independent as well as protocol-independent framework to have e-mail support for a Java application. It is a collection of classes and interfaces that comprise of an e-mail system. Following are the steps performed in sending a simple e-mail using the JavaMail API:

- Send a request for connection to an e-mail server by defining the username and password. For example, if we want to send an e-mail from xyz@abc.com , then we need to connect to the e-mail server of abc.com .
- Write a message by specifying the receiver's addresses that can include CC and BCC email addresses also.
- Add attachments to the message if required.
- Transport the message to the e-mail server.

Sending a simple e-mail requires the use of a number of classes and interfaces that are present in the javax.mail and javax.mail.internet packages. The important classes and interfaces in the JavaMail API are as follows.

- Session: Represents an e-mail session.
- Message: This abstract class models an email message.
- Transport: This represents protocol used to sending and receiving emails.
- Authenticator: This represents authentication for email provider.
- PasswordAuthentication: It holds the username and password for Authenticator.
- MimeMessage: This represents the multimedia messages.
- InternetAddress: This represents the internet email addresses of to, cc and bcc.

The JavaMail program uses the JavaMail API to exchange e-

mails, as shown in the figure below:

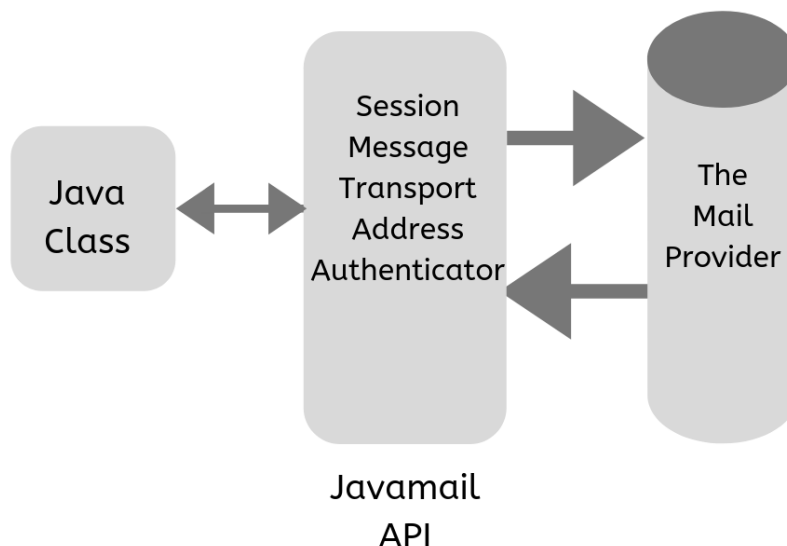


Fig. 7.2 The JavaMail API

In the above figure, the Java classes use the Spring API, which indirectly uses the JavaMail API to send and receive e-mails.

7.4 Sending a Simple E-mail

The code given below is the example to send a simple e-mail from our machine. It is assumed that our computer system is connected to the Internet and capable of sending an e-mail.

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class EmailSend
{
    public static void main(String [] args)
    {
        // Senders email ID
        String to = "hello@abc.com";

        // Sender's email ID
        String from = "data@xyz.com";

        // We are sending email from localhost
        String host = "localhost";

        // Get system properties using Properties class
        Properties properties = System.getProperties();

        // Setup the mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default object of Session.
        Session session = Session.getDefaultInstance(properties);
```

```

try
{
// Create a MimeMessage class object.
MimeMessage mime = new MimeMessage(session);

// Set From: header field of the header.
mime.setFrom(new InternetAddress(from));

// Set To: header field of the header.
mime.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

// Set Subject: header field
mime.setSubject("The Subject is here.");

// Send the message
message.setText("Message: Hi Friend");

// Send message
Transport.send(message); // This is javax.mail package
System.out.println("The message is sent successfully!");
}
catch (MessagingException e)
{
e.printStackTrace();
}
}

```

When we compile and run this program, we will get the following output

Output

```
$ java EmailSend
```

```
The message is sent successfully!
```

If we want to send an e-mail to multiple recipients then the following methods would be used to specify multiple e-mail IDs:

```
void addRecipients(Message.RecipientType type, Address[] addresses) throws MessagingException
```

Parameters:

type – This would be set to TO, CC or BCC. Here TO represents original receiver of message, CC represents Carbon Copy and BCC represents Black Carbon Copy. Example: Message.RecipientType.TO

addresses – This is an array of e-mail IDs. We would need to use InternetAddress() method while specifying email IDs.

7.5 Developing an application for email using Spring

We develop a basic e-mail application that creates simple e-mails containing text only.

Configuration file – Spring.xml

Let's create the configuration file, Spring.xml , and configure the mailSender bean of the JavaMailSender class and define its properties:

- host
- port
- username
- password

Configuration of the bean for the MyEmailService class with the mailSender property will be like this-

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSender">
  <property name="host" value="smtp.gmail.com" />
  <property name="port" value="25" />
  <property name="username" value="username" />
  <property name="password" value="password" />
  <property name="javaMailProperties">
    <properties>
      <prop key="mail.smtp.auth">true</prop>
      <prop key="mail.smtp.starttls.enable">true</prop>
```

```

    </properties>
</property>
</bean>

    <bean id="emailService" class="org.mitu.Spring.fourth.mail">
        <property name="mailSender" ref="mailSender" />
    </bean>

```

This configuration file sets the host as "smtp.gmail.com" and the port as "25." The username and the password properties need to be set with reader's username and password of their own Gmail account. The username is used as the sender of the e-mail.

Spring's e-mail sender

It is the e-mail API-specific Java file. It provides the definition of the `sendEmail()` method, which is used to send the actual e-mail to the recipient:

```

package org.mitu.Spring.fourth.mail;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class MyEmailService
{
    private MailSender ms;

    public void sendEmail(String to, String subject, String message)
    {
        // creates a simple e-mail object
        SimpleMailMessage mail = new SimpleMailMessage();
        mail.setTo(to);
        mail.setSubject(subject);
        mail.setText(message);

        // sends the e-mail
        ms.send(mail);
    }
}

```

The MailerTest class

The `MailerTest` class has the `main()` method that will call the `sendEmail()` method of the `MyEmailService` class and send an e-mail:

```

package org.mitu.Spring.fourth.mail;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
public class MailerTest
{
    public static void main( String[ ] args )
    {
        //Create the application context
        ApplicationContext context = new
        ClassPathXmlApplicationContext("Spring.xml");
        //Get the mailer instance
        EmailService emailService = (EmailService)context.getBean("emailService ");
        //Send a composed mail
        emailService.sendEmail("*****@gmail.com",
        "Email Test Subject",
        "Email Testing body");
    }
}

```

The output of this application can be checked by opening the inbox of your email.
Now check the Spring Java Messaging Service.

7.6 Spring Java Messaging Service

What is a message and messaging?

A message is nothing but just bytes of data or information exchanged between two parties. By using various specifications, a message can be described in several ways. However, it is nothing but an entity for communication. A message can be used to send a piece of information from one application to another application, which may run or may not run in the similar platform.

Messaging is the communication among different applications (in a distributed environment) or system components, which are loosely coupled unlike its peers such as TCP sockets, Remote Method Invocation (RMI), or Common Object Request Broker Architecture i.e. CO RBA, which is tightly coupled. The advantage of Java messaging includes the power to integrate various platforms, and reliability of message transfer, and bring down the system bottlenecks. Using messaging, we can increase the systems and clients who are consuming and generating the message as much as we require.

We have a lot of ways by which we communicate right from the instant messenger, to the stock taker, to the mobile-based messaging system, to the age-old messaging system; they are all part of messaging. We realize that a message is a portion of data transmitted from one system to another system and it can be between humans as well, but it is mainly between systems rather than human beings when we use the messaging using Java Messaging Service.

The Java Messaging Service?

The Java Message Service (JMS) is a Java Message Oriented Middleware (MOM) API for sending messages between two or multiple clients. It is a part of the Java Enterprise edition, J2EE. It is a broker like a postman who acts as an intermediary between the message sender and the receiver. JMS is a specification that describes an ordinary way for Java programs to make up, transmit, and read distributed enterprise messages. It advocates the loosely coupled communication without worrying about the sender and the receiver. It supplies asynchronous messaging, which means that it doesn't matter whether the sender and the receiver are present at the same time or not. The two systems that are sending or receiving messages need not be up at a time.

The JMS application

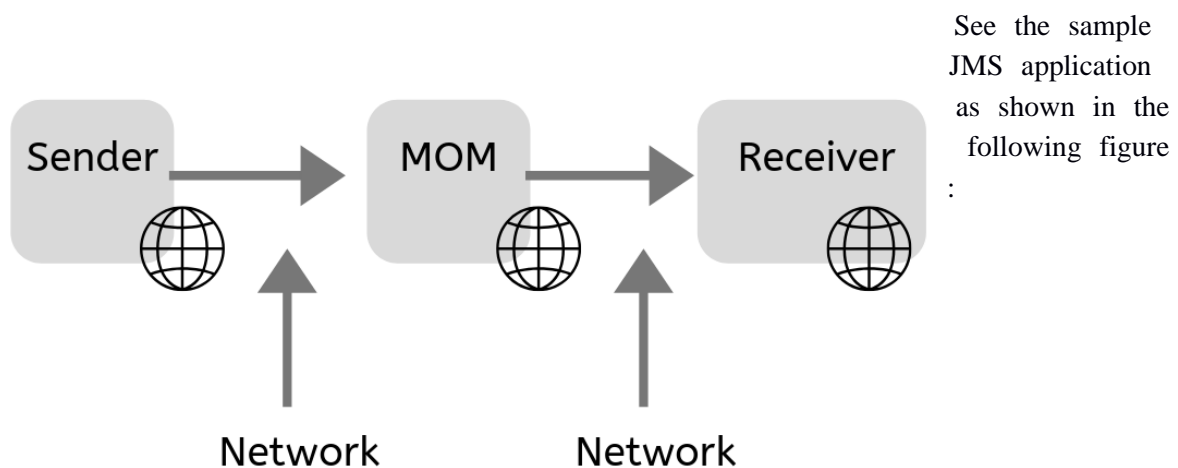


Fig. 7.3 The JMS System

We have a Sender and a Receiver. The Sender transmits a message while the Receiver gets one. We need to have a broker that is MOM between the Sender and the Receiver who takes the sender's message and passes it to the network and to the receiver. MOM is basically an MQ application such as ActiveMQ or IBM-MQ, which are two different message providers. The Sender promises the loose coupling and it can be a .NET or mainframe-based application. The Receiver can be a Java or Spring-based application, and it sends back the message to the Sender as well. This is a two-way communication that is loosely coupled.

JMS components

A JMS system contains the following components:

- **JMS Client:** It is a Java program used to send, produce, publish, receive, consume or subscribe messages.
- **JMS Sender:** It is used to send messages to the destination system. It is also known as JMS Producer or Publisher.
- **JMS Receiver:** It is used to receive messages from Source system, also known as JMS Consumer or Subscriber.
- **JMS Provider:** JMS Provider is a third-party system which is responsible to implementing the JMS API to provide messaging features to the clients. It is also known as MOM (Message Oriented Middleware) software or Message Broker. It provides some GUI components to administrate and control this MOM software.
- **ConnectionFactory:** ConnectionFactory's object is used to make a connection between Java Application and JMS Provider.
- **Destination:** These JMS Objects used by a JMS Client to specify the destination of messages it is sending and the source of messages that it receives. There are two types of Destinations available: Queue and Topic.
- **JMS Message:** It is an object that contains the data being communicated between JMS clients.

Following are the most popular JMS Providers.

S.No. JMS Provider Software Organization

1. TIBCO EMS TIBCO
2. Open MQ Oracle Corporation
3. WebSphere MQ IBM
4. Weblogic Messaging Oracle Corporation
5. Active MQ Apache Foundation
6. Rabbit MQ Rabbit Technologies (now with Spring Source)
7. SonicMQ Aurea Software
8. HornetQ JBoss
9. Sonic MQ Progress Software

The JMS Communication

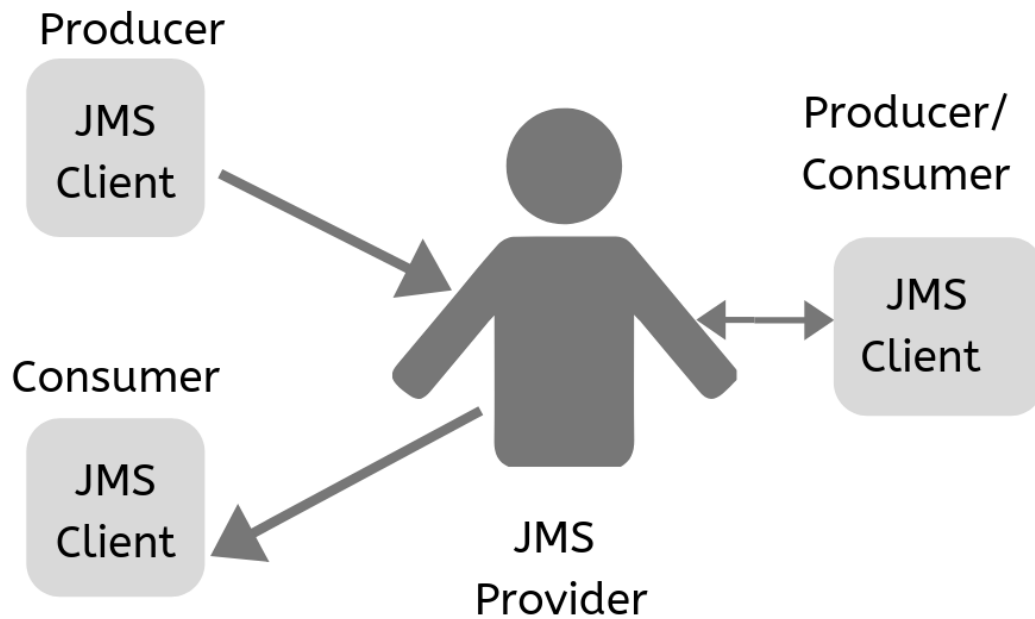


Fig. 7.4 The JMS Communication

There are three JMS clients as given in above figure. The Producer can be assumed as it's we are going to send a message to our friend. The Consumer can be assumed to be our friend who will get this message. The Producer or Consumer could be someone else who will get as well as send a message also. The JMS Provider can be assumed as the post office or postman through which the whole delivery things happen and which guarantee that the confirmed delivery happens only one time.

The Spring bean configuration (Spring.xml) Create the configuration file Spring.xml and define the respective bean definitions such as ActiveMQ ConnectionFactory, ActiveMQ queue destination, and JMS template as follows:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jms
        http://www.springframework.org/schema/jms/spring-jms.xsd
        http://activemq.apache.org/schema/core
```

```

http://activemq.apache.org/schema/core/activemq-core.xsd">
<context:component-scan base-package="org.mitu.Spring.fourth.JMS" />
<bean id="jt"class="org.springframework.jms.core.JmsTemplate">
<property name="connectionFactory" ref="connectionFactory" />
<property name="defaultDestination" ref="destination" />
</bean>
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
<property name="brokerURL">
<value>tcp://localhost:61616</value>
</property>
</bean>
<bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
<constructor-arg value="myMessageQueue" />
</bean>
</beans>

```

The Spring Framework supports JMS with the help of the classes as mentioned below:

- **ActiveMQConnectionFactory** : This class is used to create a JMS ConnectionFactory for ActiveMQ that connects to a remote broker on a specific host name and the port.
- **ActiveMQQueue**: This class will configure the ActiveMQ queue name, as in our case it is myMessageQueue.
- **JmsTemplate** : This class allows us to hide some of the lower-level JMS description while sending a message.

The MessageSender.java file – Spring JMS Template

The MessageSender class used for sending a message to the JMS queue as given in the code below:

```

package org.mitu.Spring.fourth.JMS.Message;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
@Component
public class MsgSender

```

```

{
    @Autowired
    private JmsTemplate jt;
    public void send(final Object Object)
    {
        jt.convertAndSend(Object);
    }
}

```

The Application.java file:

The Application class will have the main method, which calls the send() method to send a message, as shown in the following pseudo code:

```

package org.mitu.Spring.fourth.JMS.Main;
import java.util.HashMap;
import java.util.Map;
import org.mitu.Spring.mitu.JMS.Message.MessageSender;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Application
{
    public static void main(String[] args)
    {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("Spring.xml");
        MessageSender messageSender = (MessageSender) context
        .getBean("messageSender");
        Map<String, String> message = new HashMap<String, String>();
        message.put("Hello", "India");
        message.put("city", "Nashik");
        message.put("state", "Maharashtra");
        message.put("country", "India");
        messageSender.send(message);
        System.out.println("Message Sent to JMS Queue: " + message);
    }
}

```

```
}
```

Start ActiveMQ

Before we run Application.java , we have to start ActiveMQ, which allows us to run a broker; it will run ActiveMQ Broker using the out-of-the-box configuration.

Output:

Run Application.java and get the output on the console as shown below:

Message Sent to JMS Queue: {state=Maharashtra, Hello=India, country=India, city=Nashik}

Exception handling running Application.java

We can get the following while connecting to broker URL exception:

```
tcp://localhost:61616. Reason: Java.net.ConnectException: Connection refused: connect.
```

This exception will come if the message broker service is not up, so we need to make sure that ActiveMQ is running, as shown here:

Exception in thread "main"

```
org.springframework.jms.UncategorizedJmsException: Uncategorized
```

```
exception occurred during JMS processing; nested exception is
```

```
javax.jms.JMSException: Could not connect to broker URL:
```

```
tcp://localhost:61616. Reason: java.net.ConnectException:
```

```
Connection refused: connect at
```

```
org.springframework.jms.support.JmsUtils.convertJmsAccessException (JmsUtils.java:316)
```

Sending an HTML E-mail

It is assumed that our computer, localhost is connected to the Internet and capable of sending an e-mail.

Here we are using setContent() method to set content whose second argument is "text/html" to specify that the HTML content is included in the message. Using this example, we can send as big as HTML content we like.

```
// File Name HtmlEmail.java
```

```
import java.util.*;
```

```
import javax.mail.*;
```

```
import javax.mail.internet.*;
```

```
import javax.activation.*;
```

```
public class HtmlEmail
```

```
{
```

```
public static void main(String [] args)
{
    // Email Id of receiver
    String to = "abc@xyz.com";

    // Sender's email ID
    String from = "data@xyz.com";

    // We are sending email from localhost
    String host = "localhost";

    // Get properties of system
    Properties properties = System.getProperties();

    // Setup the mail server
    properties.setProperty("mail.smtp.host", host);

    // Get the default Session object.
    Session ssn = Session.getDefaultInstance(properties);
    try
    {
        // Create a default MimeMessage object.
        MimeMessage msg = new MimeMessage(ssn);

        // From: header field of the header.
        msg.setFrom(new InternetAddress(from));

        // To: header field of the header.
        msg.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

        // Subject: header field
        msg.setSubject("Hi! This is subject.");
    }
}
```

```

// Actual message
msg.setContent("<h1>This is your message</h1>", "text/html");

// Send message
Transport.send(msg);
System.out.println("Message is sent successfully!");
}
catch (MessagingException e)
{
e.printStackTrace();
}
}
}

```

Output:

```
$ java HtmlEmail
```

```
Message is sent successfully!
```

7.7 Sending email with attachment

JavaMail API provides some useful classes like BodyPart, MimeBodyPart etc for sending email with attachment. Let's see the steps of sending email using JavaMail API first. For sending the email using JavaMail API, we need to load the two jar files:

mail.jar

activation.jar

So we need to download these jar files (or) go to the Oracle site to download the latest version of them.

We need to go with 7 steps for sending attachment with email.

- Create the session object
- Compose the message
- Create object of MimeBodyPart and set your message text
- Create new MimeBodyPart object and set DataHandler object to this object
- Create Multipart object and add MimeBodyPart objects to this object
- Set the multipart object to the message object
- Finally, send the message

Example of sending email with attachment in Java

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

class SendWithAttachment
{
    public static void main(String [] args)
    {
        String to="abc@xyz.com"; // Receiver's email ID
        final String user="sonoojaiswal@javatpoint.com"; // Sender's email ID
        final String password="xxxxx"; // Password

        // First: get the session object
        Properties properties = System.getProperties();
        properties.setProperty("mail.smtp.host", "mail.xyz.com");
        properties.put("mail.smtp.auth", "true");

        Session ssn = Session.getDefaultInstance(properties, new
        javax.mail.Authenticator()
        {
            protected PasswordAuthentication getPasswordAuthentication()
            {
                return new PasswordAuthentication(user,password);
            }
        });

        // Second: compose message
        try{
            MimeMessage msg = new MimeMessage(ssn);
            msg.setFrom(new InternetAddress(user));
            msg.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
            msg.setSubject("Hi Message");
```



```

// Third: create MimeBodyPart object and set our message text
BodyPart msgBodyPart1 = new MimeBodyPart();
msgBodyPart1.setText("This is actual message");

//Fourth: create new MimeBodyPart object and set DataHandler object to this object
MimeBodyPart messageBodyPart2 = new MimeBodyPart();

String file = "SendWithAttachment.java";
DataSource source = new FileDataSource(file);
msgBodyPart2.setDataHandler(new DataHandler(source));
msgBodyPart2.setFileName(file);

//Fifth: Create Multipart object and add MimeBodyPart objects to this object
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(msgBodyPart1);
multipart.addBodyPart(msgBodyPart2);

//Sixth: Set the multipart object to the message object
msg.setContent(multipart);

//Seventh: send the message
Transport.send(msg);

System.out.println("The Message is sent successfully!");
}
catch (MessagingException e)
{
e.printStackTrace();
}
}
}

```

As you can see in the above code, total 7 steps are followed to send email with attachment. Now run this program by :

Load the jar file c:\> set classpath=mail.jar;activation.jar,;

Compile the source file c:\> javac SendWithAttachment.java

Run using c:\> java SendWithAttachment

Sending Email in Java through Gmail Server

We can send email by using the SMTP server of gmail. It is good if you are don't have any SMTP server and reliable. The SSL, Secure Socket Layer is basically used for security if we are sending email through gmail server.

Sending Email through Gmail Server with SSL

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
class Gmailer
{
    public static void send (String from,String password,String to,String sub,String message)
    {
        //Get properties object
        Properties properties = new Properties();
        properties.put("mail.smtp.host", "smtp.gmail.com");
        properties.put("mail.smtp.socketFactory.port", "465");
        properties.put("mail.smtp.socketFactory.class",
            "javax.net.ssl.SSLSocketFactory");
        properties.put("mail.smtp.auth", "true");
        properties.put("mail.smtp.port", "465");
        // Get Session object
        Session session = Session.getDefaultInstance(properties, new
            javax.mail.Authenticator()
            {
                protected PasswordAuthentication getPasswordAuthentication()
                {
                    return new PasswordAuthentication(from,password);
                }
            });
```

```

// No compose the message
try {
MimeMessage msg = new MimeMessage(session);
msg.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
msg.setSubject(sub);
msg.setText(messsage);
// Now send the message
Transport.send(message);
System.out.println("The message sent successfully!");
}
catch (MessagingException e)
{
throw new RuntimeException(e);
}
}
}
}
public class SendMailSSL
{
public static void main(String[] args)
{
// from, password, to, subject, message
Mailer.send("from@gmail.com", "xxxxx", "to@gmail.com", "Welcome!", "Hi Friend");
}
}
}

```

As we can see in the above example, userid and password need to be authenticated. As, this program illustrates, we can send email easily but change the username and password as per the account used.

Receiving email in Java

For receiving email Store and Folder classes are used in collaboration with MimeMessage, Session and Transport classes.

For sending the email using JavaMail API, we need to load the same jar files:

mail.jar

activation.jar

download these jar files (or) go to the Oracle site to download the latest version.

Steps for receiving the email using JavaMail API

- Create the session object
- Create the POP3 store object and connect with the pop server
- Create the folder object and open it
- Retrieve the messages from the folder in an array format and print it on screen
- Close the store and folder objects

Example : Receiving email in Java

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.Session;
import java.io.IOException;
import java.util.Properties;
import javax.mail.Folder;
import com.sun.mail.pop3.POP3Store;
public class ReceiveMail
{
    public static void receiveEmail(String pop3Host, String storeType, String user, String password) {
        try {
            //First: Create the session object
            Properties properties = new Properties();
            properties.put("mail.pop3.host", pop3Host);
            Session emailSession = Session.getDefaultInstance(properties);

            //Second: create the POP3 store object and connect with the pop server
            POP3Store emailStore = (POP3Store) emailSession.getStore(storeType);
            emailStore.connect(user, password);
```

```

//Third: create the folder object and open it
Folder emailFolder = emailStore.getFolder("MYDATA");
emailFolder.open(Folder.READ_ONLY);

//Fourth: acquire the messages from the folder in an array element and print it
Message[ ] messages = emailFolder.getMessages();
for (int i = 0; i < messages.length; i++)
{
    Message message = messages[i];
    System.out.println("-----");
    System.out.println("Email Number: " + (i + 1));
    System.out.println("Subject: " + message.getSubject());
    System.out.println("From: " + message.getFrom()[0]);
    System.out.println("Text: " + message.getContent().toString());
}

//Fifth: close the store and folder objects
emailFolder.close(false);
emailStore.close();
}
catch (IOException e)
{
    e.printStackTrace();
}
catch (NoSuchProviderException e)
{
    e.printStackTrace();
}
catch (MessagingException e)
{
    e.printStackTrace();
}
}

```

```

    }
    public static void main(String[] args)
    {
        String host = "mail.gmail.com"; // Change as per requirement
        String mailStoreType = "pop3";
        String username= "data@gmail.com";
        String password= "xxxxx"; // User password
        receiveEmail(host, mailStoreType, username, password);
    }
}

```

In the above example, userid and password need to be authenticated. We can receive email easily but change the username and password as per our account.

Receiving email with attachment

As we receive the email, we can receive the attachment also by using Multipart and BodyPart classes available in JavaMail API.

```

import java.util.*;
import javax.mail.internet.*;
import javax.activation.*;
import javax.mail.*;
import java.io.*;

class ReadAttachments
{
    public static void main(String [] args) throws Exception
    {
        String host="mail.zoho.com";
        final String user="mail@zoho.com";
        final String password="xxxxx"; // Actual password

        Properties properties = System.getProperties();
        properties.setProperty("mail.smtp.host",host );
        properties.put("mail.smtp.auth", "true");

        Session session = Session.getDefaultInstance(properties,

```

```
new javax.mail.Authenticator()
{
protected PasswordAuthentication getPasswordAuthentication()
{
return new PasswordAuthentication(user, password);
}
});
```

```
Store store = session.getStore("pop3");
store.connect(host,user,password);
```

```
Folder folder = store.getFolder("MYDATA");
folder.open(Folder.READ_WRITE);
```

```
Message[] message = folder.getMessages();
for (int a = 0; a < message.length; a++)
{
System.out.println(message[a].getSentDate());
Multipart multipart = (Multipart) message[a].getContent();
for (int i = 0; i < multipart.getCount(); i++)
{
BodyPart bodyPart = multipart.getBodyPart(i);
InputStream stream = bodyPart.getInputStream();
BufferedReader reader = new BufferedReader
(new InputStreamReader(stream));
while (reader.ready()) {
System.out.println(reader.readLine());
}
System.out.println();
}
System.out.println();
}
folder.close(true);
```

```
store.close();  
}  
}
```

7.8 Forwarding an email

We can forward the received mail to someone else as we send emails. There are several JavaMail classes that are used to forward the messages to the destination. See the below example for forwarding email.

```
import java.util.*;  
import javax.mail.*;  
import javax.mail.internet.*;  
public class ForwardingEmail  
{  
    public static void main(String[] args) throws Exception  
    {  
        final String user="maya@abc.com"; // User ID  
        final String password="xxxxx"; // Password  
  
        // Get the session object  
        Properties properties = new Properties();  
        properties.put("mail.smtp.host", "mail.abc.com");  
        properties.put("mail.smtp.auth", "true");  
  
        Session session = Session.getDefaultInstance  
(properties, new javax.mail.Authenticator()  
        {  
            protected PasswordAuthentication getPasswordAuthentication()  
            {  
                return new PasswordAuthentication(user,password);  
            }  
        });
```



```

// Get a Store object and connect to current host
Store store = session.getStore("pop3");
store.connect("mail.abc.com", user, password);

// Create a Folder object and open the folder
Folder folder = store.getFolder("MYDATA");
folder.open(Folder.READ_ONLY);

Message msg = folder.getMessage(1);

// Retrive all the information from the message
String from = InternetAddress.toString(msg.getFrom());
if (from != null)
{
    System.out.println("From: " + from);
}

String replyTo = InternetAddress.toString( msg.getReplyTo());
if (replyTo != null)
{
    System.out.println("Reply-to: " + replyTo);
}

String to = InternetAddress.toString ( msg.getRecipients
(Message.RecipientType.TO));
if (to != null)
{
    System.out.println("To: " + to);
}

String subject = msg.getSubject();
if (subject != null)
{
    System.out.println("Subject: " + subject);
}

```

```
Date sent = msg.getSentDate();
if (sent != null)
{
    System.out.println("Sent: " + sent);
}
System.out.println(msg.getContent());

// Compose the message to forward
Message msg2 = new MimeMessage(session);
msg2.setSubject("Fwd: " + message.getSubject());
msg2.setFrom(new InternetAddress(from));
msg2.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Original message is: ");

// Create a multi-part to combine the parts
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Create and fill part for the forwarded content
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler(message.getDataHandler());

// Add part to multi part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
msg2.setContent(multipart);

// Send message
Transport.send(msg2);
```

```
System.out.println("The message is forwarded!");  
}  
}
```

Exercises:

1. What is Java Messaging Service. Draw the simple diagram to explain the same.
2. How to forward the email. Explain with example.
3. Write a note on Javamail API. Explain all the components of the same.
4. What are different types of API? Enlist and explain the functionality of each.
5. How JMS communication is taken place? Draw neat diagram and explain the process in detail.

Objective Questions:

1. Which class This class allows us to hide some of the lower-levelJMSdescription while sending amessage?
 - a. JMSTemplate
 - b. LowerJMS
 - c. JMSActivate
 - d. JavaMessagingService
2. What kind of object is used to make a connection between Java Applicationand JMS Provider.?
 - a. ConnectionTemplate
 - b. ConnectionFactory
 - c. TempConnection
 - d. FactConnection

Unit 8

JSON

Introduction

JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transfer data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common type of data format used for asynchronous browser–server communication.

JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to produce and parse JSON-format data. The official Internet media type for JSON is application/json. The JSON filenames use the extension .json.

Developer Douglas Crockford originally specified the JSON format in the early 2000s. It was first standardized in 2013 as RFC 7158 and ECMA-404.

8.1 What is JSON ?

- JSON is an acronym of JavaScript Object Notation.
- It is a lightweight data-interchange file format.
- It is simple to read and write than XML.
- It is language independent file format.
- It supports array, object, string, number and values.

Applications of JSON

- It is used while writing JavaScript based applications that has browser extensions and websites.
- JSON data format is used for serializing and transmitting structured data all over network connection.
- It is primarily used to transfer data among a server and web applications.
- Web services and APIs use JSON format to provide public data for various services.
- It is compatible with all modern programming languages like Python, Ruby and R.

Differentiating the JSON and XML formats.

Sr. No.	JSON	XML
1	JavaScript Object Notation	Extensible markup language
2	Based on JavaScript programming language.	Derived from SGML.
3	Representation is done in objects.	This markup language uses tag structure to represent data items.
4	No support for name-spaces.	It supports namespaces.
5	Array support present	No Array support present
6	Files are very easy to read as compared to XML.	Its documents are comparatively difficult to read and interpret than JSON.
7	No end tag available.	Start and end tags are available.
8	Less security.	It is more secured than JSON.
9	No support to comments.	Support for comments.
10	Supports UTF-8 encoding.	Support for various encoding is present.

Similarities between JSON and XML

- Both are simple and open.
- Both supports unicode. So internationalization is supported by JSON and XML both.
- Both represents self describing data formats.
- Both are inter-operable or independent of programming languages.

8.2 Simple Example in JSON

The example below shows how to use JSON to store information related to books having properties like topic and edition.

```
{  
  "book": [  
    {  
      "id": "01",  
      "language": "C",  
      "edition": "15th",  
      "author": "Yashwant Kanetkar"  
    },  
  ],  
}
```

```
{
  "id": "02",
  "language": "Python",
  "edition": "Fifth",
  "author": "Yela Rawbundy"
}
```

This file data format can be added for the HTML code and JavaScript. Following is the code for json.htm –

```
<html>
  <head>
    <title>JSON </title>
    <script language = "javascript" >
      var object1 = { "language" : "C", "author" : "Yashwant Kanetkar" };
      document.write("<h1>JSON with JavaScript example</h1>");
      document.write("<br>");
      document.write("<h3>Language = " + object1.language+"</h3>");
      document.write("<h3>Author = " + object1.author+"</h3>");

      var object2 = { "language" : "Python", "author" : "Yela Rawbundy" };
      document.write("<br>");
      document.write("<h3>Language = " + object2.language+"</h3>");
      document.write("<h3>Author = " + object2.author+"</h3>");

      document.write("<hr />");
      document.write(object2.language + " programming language can be studied from book written by " + object2.author);
      document.write("<hr />");
    </script>
  </head>
  <body>
  </body>
</html>
```

Now let's try to open json.htm using any web browser or any other javascript enabled browser that produces the following result –

JSON with JavaScript example

Language = C

Author = Yashwant Kanetkar

Language = Python

Author = Yela Rawbundy

Python programming language can be studied from book written by Yela Rawbundy

Fig. 8.1 Output of JavaScript using JSON data format

8.3 JSON – Syntax

Let's look at the basic syntax of JSON. JSON syntax is basically considered as a subset of JavaScript syntax; it includes the following –

- Data is represented in the format of key/value pairs.
- Four symbols are used in JSON. The curly braces { } hold objects and each name is followed by ':' (colon), and the key/value pairs are separated by , (comma).
- Square brackets hold arrays and values are separated by , (comma).

Below is a simple example.

```
{  
  "student": [  
    {  
      "roll": "01",  
      "name": "Rashmi",  
      "class": "SY",  
      "marks": "68.77"  
    },  
    {
```

```

"roll": "02",
"name": "Akash",
"class": "SY",
"marks": "58.23"
},
{
"roll": "03",
"name": "Paramanand",
"class": "TY",
"marks": "66.20"
}
]
}

```

8.4 DataTypes of JSON

JSON format supports the following data types.

Sr.No.	Type & Description
1	Number It is double-precision floating-point format of JavaScript
2	String It is double-quoted Unicode set of characters
3	Boolean either true or false
4	Array It is an ordered sequence of values of same type
5	Value It can be a string, a number, a boolean, or null value
6	Object It is an unordered collection of key-value pairs
7	Whitespace This can be used between any pair of tokens

8	null The empty character (referred as nothing)
---	----------------------------------------------------------

Number

It is a double precision floating-point format in JavaScript and it depends on its implementation.

Unlike other programming languages, Octal and hexadecimal formats are not used. It does not have a NaN or Infinity number.

The following table shows the number types data.

Sr.No.	Type & Description
1	Integer All decimal Digits from 0 to 9 including the positive and negative numbers
2	Fraction Floating point values with a decimal dot (.)
3	Exponent The float values including e, -e, E or -E e.g. 4.5e12

Syntax:

The following way is used to declare the variable with the data type and constant.

```
var json-object-name = { string : number_value, .....}
```

Example:

The example below shows Number Datatype, value should not be quoted –

```
var obj = {marks: 59}
```

String

It is a sequence of double quoted Unicode characters which also allows the escape sequence characters. Character is a single character string i.e. a string with length 1. The table below shows various special characters that you can use in strings of a JSON document.

Sr No.	Type & Description
1	\" double quotation character

2	<code>\\</code> backslash character
3	<code>\/</code> forward slash character
4	<code>\b</code> backspace character
5	<code>\f</code> form feed character
6	<code>\n</code> new line character
7	<code>\r</code> carriage return
8	<code>\t</code> horizontal tab character
9	<code>\u</code> four hexadecimal digits or unicode character

Syntax:

```
var json-object-name = { string : "string value", .....}
```

Example

The example below shows the String type data.

```
var obj = { name: "Rashmi" }
```

Boolean:

It includes either true or false values. The concept is similar to the one in Java

Syntax

```
var json-object-name = { string : true/false, .....}
```

Example

```
var obj = {name: 'Rashmi', marks: 59, distinction: false}
```

8.5 Array

It is an ordered collection of values stored in homogeneous order in memory. The values of array are enclosed in pair of square brackets which means that array begins with '[' and ends with ']'.. The values are separated by, (comma) within it. Array indexing can be started at 0 and ends till n-1, where n is total number of elements in array. Arrays should be used when the key names are sequential integers.

Syntax

```
[ value, .....]
```

Example

Example showing array containing multiple objects –

```
{
  "books": [
    { "language":"Marathi" , "Region":"Goa" },
    { "language":"Urdu" , "Type":"Indo-Aryan" },
    { "language":"Kannada" , "Region":"Karnataka" }
  ]
}
```

Object

The object is an unordered set of key – value pairs. They are enclosed in a pair of curly braces that is, it starts with '{' and ends with '}'. Each key in this pair is followed by ':'(colon) and the key-value pairs are separated by , (comma). The keys must be strings and should be unique in all the set. The objects should be used when the key names are arbitrary strings.

Syntax

```
{ string : value, .....}
```

Example

Example showing Object –

```
{
  "pin": "411061",
  "city": "Pune",
  "ranking": 2,
}
```

Whitespace

The whitespace can be inserted between any pair of tokens. It is added to make a code more readable. The example shown below gives declaration with and without whitespace.

Syntax:

```
{string:" ",....}
```

Example:

```
var obj1 = {"name": "Amar Choudhary"}
var obj2 = {"name": "Shane Warne"}
var obj3 = {"name": "Aniket Vishwasrao"}
```

null

It means empty type.

Syntax

null

Example:

```
var i = null;
if(i == null)
{
    document.write("<h3>value is null</h3>");
}
else
{
    document.write("<h3>value is not null</h3>");
}
```

JSON Value

The JSON value includes the following types of data.

- number (integer or floating point)
- string
- boolean
- array

- object
- null

Syntax:

String | Number | Object | Array | TRUE | FALSE | NULL

Example

```
var i = 45.44;  
var j = "Hi Python";  
var b = true;  
var k = null;
```

8.6 JSON Object

The JSON object holds a key-value pair. In which each key is represented as a string in JSON and value can be of any type. The keys and values are separated by colon. Each key/value pair is separated by comma. The curly brace { represents JSON object. Let's understand it by an example of JSON object.

```
{  
  "player": {  
    "name": "Ajinkya Rahane",  
    "runs": 4867,  
    "bowler": false  
  }  
}
```

In the given example, player is an object in which "name", "runs" and "bowler" are the keys. And there are string, number and boolean value are used for the keys.

JSON Object with Strings

The string value must be enclosed within double quote.

```
{  
  "name": "Rashmi Thorave",  
  "email": "rashmi1988@gmail.com"  
}
```

JSON Object with Numbers

The JSON object supports numbers in double precision floating-point format i.e. 64 bit float format. The number can have decimal digits (0 to 9), the fraction number with a decimal point (.245, .102 etc) and exponents (e, e+, e-, E, E+, E-).

```
{  
  "integer": 15,  
  "fraction": 0.4527,  
  "exponent": 95.22e12  
}
```

JSON Object with Booleans

The JSON object also supports boolean values i.e. true or false.

```
{  
  "first": true,  
  "second": false  
}
```

JSON Nested Object Example

A JSON object can contain another object also. The example below shows the JSON object having another object within it.

```
{  
  "firstName": "Rakesh",  
  "lastName": "Sharma",  
  "age": 35,  
  "address" : {  
    "streetAddress": "301, Akanksha Plaza",  
    "city": "Nashik",  
    "state": "Maharashtra",  
    "postalCode": "422006"  
  }  
}
```

JSON Array

The JSON array represents ordered list of values stored in a homogeneous locations. The JSON array can store multiple values of same type of data. It can store string, number, boolean or object in JSON array. The values here must be separated by comma. The [(square bracket) represents JSON array.

JSON Array of Strings

The below example shows the JSON arrays storing string values.

```
["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"]
```

JSON Array of Numbers

The below example shows the JSON arrays storing number values.

```
[45,67,88,16,39,83,47,33,18,78]
```

JSON Array of Booleans

The below example shows the JSON arrays storing boolean values.

```
[true, true, false, false, true, false]
```

JSON Array of Objects

The below example shows the JSON arrays storing array of 4 object values.

```
{ "employees": [
  { "name": "Bahubali", "email": "bahubali@gmail.com", "age": 26 },
  { "name": "Chitti", "email": "chittibhau@gmail.com", "age": 56 },
  { "name": "Chulbul Pandey", "email": "chulbulpandey@gmail.com", "age": 33 },
  { "name": "Sir Jadeja", "email": "jaddusuperkings@gmail.com", "age": 31 }
]}
```

JSON Multidimensional Array

We can store array inside JSON array, it is known as array of arrays or a multidimensional array.

```
[
  [ "y", "z", "x" ],
  [ "q", "w", "e" ],
  [ "m", "a", "n" ]
]
```

8.7 JSON library in Java

As we have already studied, the JSON format is one of the most favourite formats to transmit and exchange data in web. Almost all RESTful web services take JSON format data as input and provide JSON output. But unluckily Oracle Java Development Kit doesn't have built-

in support for this one of the most common web standard. As a Java developer if we want to develop RESTful web service and produce JSON data or if we are developing a client to an existing RESTful web services and want to consume JSON response, we don't have to worry about this. Luckily, there are so many open source libraries and APIs available for making, parsing and processing JSON response in Java e.g. Jackson, Google Gson, json-simple etc.

There are numerous JSON libraries exist in Java but we don't need to learn all of them, learning just one of them e.g. Jackson should be enough, but, another side, it's worth knowing what are some of the most popular JSON parsing library exists in our disposal. Let us see the top five useful JSON libraries which every Java Enterprise Edition developer should be aware of.

Useful JSON libraries in Java

There are a lot many JSON libraries and APIs are available for Java but depending upon your need, we can choose any of them.

Jackson

The Jackson is a multi-purpose Java library for processing JSON data format. Jackson aims to be the best possible combination of fast, correct, lightweight, and ergonomic for developers.

The Jackson offers three methods for processing JSON format, each of them having its own advantages and disadvantages.

- Streaming API or incremental parsing/generation: reads and writes JSON content as discrete events.
- Tree model: provides a mutable in-memory tree representation of a JSON document. The data representation is done in graphical format.
- Data binding: converts JSON to and from POJO's

If we are only interested in converting Java object to and from JSON string then the third method can be used more effectively.

The advantage of Jackson is that it supplies the heaps of features, and looks to be a good tool for reading and writing JSON in a variety of ways, but same time its size becomes a disadvantage if our requirement is just to serialize and deserialize Java object to JSON String.

In order to use Jackson, we can include following maven dependency or manually include jackson-core-2.3.1.jar, jackson-databind-2.3.1.jar, and jackson-annotations-2.3.1.jar in Classpath.

Jackson Serialization Annotations

@JsonAnyGetter

The @JsonAnyGetter annotation allows the flexibility of using a Map field as standard properties to us.

Lets see an example. The ExtendableBean entity has the name property and a set of extendable attributes in the form of key-value pairs:

```
public class ExtendableBean
{
    public String name;
    private Map<String, String> properties;
    @JsonAnyGetter
    public Map<String, String> getProperties()
    {
        return properties;
    }
}
```

When we serialize an instance of this entity, we get all the key-values in the Map as standard, plain properties:

```
{
    "name":"The First Bean",
    "attr2":"val2",
    "attr1":"val1"
}
```

And here how the serialization of this entity looks like in practice:

@Test

```
public void whenSerializingUsingJsonAnyGetter_thenCorrect() throws JsonProcessingException
{
```

```

ExtendableBean bean = new ExtendableBean("My bean");
bean.add("attr1", "val1");
bean.add("attr2", "val2");

String result = new ObjectMapper().writeValueAsString(bean);

assertThat(result, containsString("attr1"));
assertThat(result, containsString("val1"));
}

```

We can also use optional argument `enabled` as `false` to disable `@JsonAnyGetter()`. In this case, the Map will be converted as JSON and will appear under properties variable after serialization.

GSON

The full form of GSON is the google-gson library. Gson is a Java library capable of converting Java objects into their JSON representation and JSON strings to an equivalent Java object without the requirement of placing Java annotations in our classes.

Features of Gson library:

- It provides simple `toJson()` and `fromJson()` methods to convert Java objects to JSON and vice-versa.
- It Supports arbitrarily complex objects
- It has extensive support of Java Generics library
- It allows custom representation for Java objects
- It allows pre-existing unmodifiable objects to be converted to and from JSON.

How to use GSON?

1. Maven Dependency

First of all, we have to include the gson dependency in our pom.xml file as-

```

<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.5</version>
</dependency>

```

We can find the latest version of gson on Maven Central online website.

2. Using JsonParser

The first approach is for converting a JSON String to a JsonObject is a two-step process that uses the JsonParser class. In this step, we need to parse our original String. Gson provides us a parser called JsonParser, which parses the specified JSON String into a parse tree of JsonElements:

```
public JsonElement parse(String json) throws JsonSyntaxException
```

Once we have our String parsed in a JsonElement tree, we'll use the getAsJsonObject() method, which will return the desired result to us.

Let's see how we get our final JsonObject:

```
String json = "{ \"name\": \"Schildt\", \"python\": false }";  
JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();  
Assert.assertTrue(jsonObject.isJsonObject());  
Assert.assertTrue(jsonObject.get("name").getString().equals("Baeldung"));  
Assert.assertTrue(jsonObject.get("java").getAsBoolean() == true);
```

3. Using fromJson() method.

In the second approach, we create a Gson instance and use the fromJson method. This method deserializes the specified JSON String into an object of the specified class:

```
public <T> T fromJson(String json, Class<T> classOfT) throws JsonSyntaxException
```

The following way is used to see this method to parse our JSON String, passing the JsonObject class as the second parameter:

```
String json = "{ \"name\": \"Schildt\", \"python\": false }";  
JsonObject convertedObject = new Gson().fromJson(json, JsonObject.class);  
Assert.assertTrue(convertedObject.isJsonObject());  
Assert.assertTrue(convertedObject.get("name").getString().equals("Schildt"));  
Assert.assertTrue(convertedObject.get("python").getAsBoolean() == true);
```

8.8 json-simple

The json-

simple is one of the simplest JSON library used by developers, which is also lightweight.

We can use this library to encode or decode JSON text in any format specified. It's an open source library which is flexible and simple to be used by reusing Map and List interfaces from Java Development Kit. A good thing about this library that it has no external dependency present and both source and binaries are JDK 1.2 and above compatible.

The benefit of using Json-

simple is that it is lightweight, having just 12 classes and it provides support for Java IO

readers and writers classes. We can take our decision better if we know about JSON format i.e. how information is represented there in JSON.

If we are looking for a simple lightweight Java library that reads and writes JSON and supports Streams, json-

simple is probably a good match. It does what it says with just 12 classes, and works on legacy (1.4) JREs also.

In order to use JSON-

Simple API, we need to include maven dependency in our project's pom.xml file or alternatively, we can also include following JAR files in your classpath. You can also see this tutorial to learn about how to read JSON String in Java using json-simple.

How to use the json-simple?

Following steps are undertaken to use the json-simple library in Java.

1. Install json.simple

In order to install json.simple, we need to set classpath of json-simple.jar or add the Maven dependency from their online website.

2. Add maven dependency, in pom.xml file.

```
<dependency>
<groupId>com.googlecode.json-simple</groupId>
<artifactId>json-simple</artifactId>
<version>1.2</version>
</dependency>
```

3. Java JSON Encode

See the below code, which shows simple example to encode JSON object in Java.

```
import org.json.simple.JSONObject;
public class JsonEncode
{
    public static void main(String args[])
    {
        JSONObject obj=new JSONObject();
        obj.put("name","Amarpreet");
        obj.put("age",new Integer(22));
        obj.put("salary",new Double(22000));
        System.out.print(obj);
    }
}
```

Output:

```
{"name":"Amarpreet","salary":22000.0,"age":22}
```

JSON Encode using Map

Apart from previous method, the following example shows the procedure to encode JSON object using map in Java.

```
import java.util.Map;
import org.json.simple.JSONValue;
import java.util.HashMap;
public class JsonEncode2
{
    public static void main(String args[])
    {
        Map obj=new HashMap();
        obj.put("name","Rashmi");
        obj.put("age",new Integer(30));
        obj.put("salary",new Double(67000));
        String jText = JSONValue.toJSONString(obj);
        System.out.print(jText);
    }
}
```

Output:

```
{"name":"Rashmi","salary":67000.0,"age":30}
```

JSON Array Encode method

The following example shows the procedure to encode JSON array.

```
import org.json.simple.JSONArray;
public class JsonEncode2
{
    public static void main(String args[])
    {
        JSONArray arr = new JSONArray();
```

```

arr.add("Ranaji");
arr.add(new Integer(35));
arr.add(new Double(10000));
System.out.print(arr);
}
}

```

Output:

```
["Ranaji", 35, 10000.0]
```

JSON Array Encode using List

The ArrayList class object can be used to encode JSON array using List also.

```

import java.util.List;
import org.json.simple.JSONValue;
import java.util.ArrayList;
public class JsonEncode3
{
    public static void main(String args[])
    {
        List arr = new ArrayList();
        arr.add("Parampara");
        arr.add(new Integer(42));
        arr.add(new Double(111000));
        String jText = JSONValue.toJSONString(arr);
        System.out.print(jText);
    }
}

```

Output:

```
["Parampara",42,111000.0]
```

4. Java JSON Decode

The following code shows the way to decode JSON string.

```

import org.json.simple.JSONObject;
import org.json.simple.JSONValue;

public class JsonDecode
{
    public static void main(String[] args)
    {
        String s="{\"name\":\"Rajinikanth\",\"salary\":55000,\"age\":66}";
        Object obj=JSONValue.parse(s);
        JSONObject jsonObject = (JSONObject) obj;

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

```

Output:

Rajinikanth 55000.0 66

Flexjson

The Flexjson is another lightweight library for serializing and deserializing Java objects into and from JSON format. It allows creation of both deep and shallow copies of objects. The depth to which an object is serialized can be controlled with Flexjson and thus making it similar to lazy-loading, allowing us to extract only the information that we need. This is not the case since we want an entire object to be written to file, but it's good to know that it can do that.

If we know that we are going to use only a small amount of data in our application and we wish to store or read it to and from JSON format, we should consider using Flexjson or Gson.

How to use Flexjson?

The Flexjson takes a different approach allowing us to control the depth to which it will serialize. It's very similar in concept to lazy loading in Hibernate which allows us to have a connected object model, but control what objects are loaded out of our database for performance. Let's look at a simple example first to get a feel for how the library works. Say we are serializing an instance of Animal. We might do it in the following way.

```

public String do( Object arg1, ... )
{
    Animal a = ...load an animal...;
    JsonSerializer serializer = new JsonSerializer();
    return serializer.serialize( a );
}

```

Output:

```

{
    "class": "Animal",
    "name": "Xenial Xerus",
    "nickname": "Wolf"
}

```

JSON-lib

JSON-

lib is a Java library, based on the original work by creator of JSON, Douglas Crockford, capable of transforming beans, maps, collections, Java arrays and XML to JSON and back again to beans and DynaBeans.

If we are going to use big amounts of data and wish to store or read it to and from JSON format, we should consider using Jackson or JSON-lib.

How to use this library ?

1. Here is our pom.xml:

```

<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20180130</version>
</dependency>

```

The latest version can be found in the Maven Central repository online. This package has already been included in Android SDK, so we shouldn't include it while using the same

.

2. JSON in Java [package org.json]

The JSON-

Java library is also known as org.json (not Google's org.json.simple) provides us with classes that are used to parse and manipulate JSON.

Furthermore, this library can also convert among JSON, XML, HTTP Headers, Cookies, Comma-Delimited List or Text, etc.

Following are the classes used to JSON embedding in Java.

- **JSONObject** – similar to Java's native Map like object which stores unsorted key-value pairs in it.
- **JSONArray** –
This is ordered sequence of values similar to Java's native Vector class implementation.
- **JSONTokener** –
This is a tool that breaks a piece of text into a series of tokens which can be used by JSONObject or JSONArray to parse JSON strings from it.
- **CDL** – This tool provides methods to convert comma-delimited text into a JSONArray and vice versa.
- **Cookie** – It is used to convert from JSON String to cookies and vice versa.
- **HTTP** – It is used to convert from JSON String to HTTP headers and vice versa.
- **JSONException** –
This is subclass of Exception and a standard exception thrown by this library.

JSONObject

The A JSONObject is an unsorted collection of key and value pairs, corresponding Java's native Map implementations. Keys are unique Strings that cannot be null. The values can be anything from a Boolean, Number, String, JSONArray or even a JSONObject.NULL object.

The A JSONObject can be represented by a String enclosed within curly braces { } with keys and values separated by a colon, and pairs separated by a comma like general JSON format. It has several constructors with which to construct a JSONObject. It also supports the following methods:

- **get(String key)** –
It is used to get the object associated with the supplied key, throws JSONException if the key is not found in it.
- **opt(String key)**–
It is used get the object associated with the supplied key, null otherwise.
- **put(String key, Object value)** – It is used to insert or replace a key-value pair in current JSONObject.

The put() method is an overloaded method which accepts a key of type String and multiple types for the value.

Following are the operations supported by JSON library here.

1. Creating JSON Directly from JSONObject

The JSONObject explores an API similar to Java's Map interface. We can use the put() method and supply the key and value as an argument:

```
JSONObject jo = new JSONObject();  
jo.put("name", "Aniket");  
jo.put("age", "28");  
jo.put("city", "Pune");
```

Now our JSONObject would look like:

```
{"city":"Pune","name":"Aniket","age":"28"}
```

There are seven different overloaded signatures of JSONObject.put() method are available. The data present in the object will have the key only be unique, non-null String, the value can be anything.

2. Creating JSON from Map

Instead of directly putting key and values in a JSONObject, we can construct a custom Map and then pass it as an argument to JSONObject's constructor as shown in the example below.

```
Map<String, String> map = new HashMap<>();  
map.put("name", "Aniket");  
map.put("age", "28");  
map.put("city", "Pune");  
JSONObject jo = new JSONObject(map);
```

3. Creating JSONObject from JSON String

In order to parse a JSON String to a JSONObject, we can just pass the String to the constructor. The example below shows the JSON String.

```
JSONObject jo = new JSONObject(  
    "{\"city\":\"Pune\",\"name\":\"Aniket\",\"age\":\"28\"}"  
);
```

The passed String argument must be a legal JSON string otherwise this constructor may throw a JSONException.

4. Serialize Java Object to JSON

One of JSONObject's constructors takes a POJO as its argument. In the example below, the package uses the getters from the DemoBean class and creates an appropriate JSONObject for it.

In order to get a JSONObject from a Java Object, we have to use a class that is a legal Java Bean object:

```
DemoBean demo = new DemoBean();  
demo.setId(1);  
demo.setName("Mahabharat");  
demo.setActive(true);  
JSONObject jo = new JSONObject(demo);
```

The JSONObject jo for this example is going to be:

```
{"name":"Mahabharat","active":true,"id":1}
```

Although we have a way to serialize a Java object to JSON string, there is no way to convert it back using this library.

8.9 Publishing a Service using JSON in JSP

In order to publish a service using JSON, at plain jsp project, it will need json library. We may use any json library. Here we used json-simple. We need to download the library from Maven repository. The Apache Tomcat 6 i



s also required, so jar file will put at lib folder, just paste and restart server.

Fig. 8.1 The JSON and JSP Communication

The concept is index.jsp will request some params send through ajax to services.jsp. From there it will process as params and take data from database employees, return to index.jsp as json.

Our project I contain the following.

- index.jsp
- services.jsp
- mysql_employees_jspjson.sql
- jquery-1.10.2.min.js (jQuery Library of JS)

1. index.jsp (front-end view that show what will be requested)

```
<%@ page contentType="text/html; charset=iso-8859-1" language="java" import="java.sql.*" errorPage="" %>
<!DOCTYPE html>
<html>
<head>
<title>JSP JSON Tester</title>
<script src="jquery-1.11.1.min.js"></script>
</head>
<body bgcolor="white">
  <div id="con">
```

<label>What department ? (IT / COMM / HR / FINANCE)</label>

<input id="dept" type="text" name="dept" placeholder="Department"/>

<button id="conBtn" type="submit">Submit</button>

</div>

<div id="json" style="display: none;">

<h3>JSON Data</h3>

<p id="conCode"></p>

</div>

<div id="table" style="display:none;">

<h3>Table Result</h3>

<table border="1" padding="1">

<thead>

<th>No</th>

<th>Name</th>

<th>Department</th>

<th>Address</th>

</thead>

<tbody id="tableBody">

</tbody>

</table>

</div>

<script>

function submitForm(){

var inp = \$('#dept').val().toUpperCase();

\$.ajax({

async: false,

type: "GET",

contentType: "application/json; charset=utf-8",

url: "services.jsp",

data: { dept: inp}

}).success(function(data){

var jsondata = JSON.parse(data);

```

loadData(jsondata);
$('#conCode').html(data);
}).fail(function(jqXHR, textStatus){
console.log("[AJAX] Error: JSON WSDL Lookup>>>" + textStatus);
}).done(function(){
console.log("[AJAX] Complete: JSON WSDL Lookup");
});
}

```

```

function loadData(datas){
var sb = "";
var table = $('#tableBody');
$('#json').show();
$('#table').show();
table.html("");

```

```

$.each(datas, function(index, value){
sb += '<tr>';
sb += ' <td>' + value.no + '</td>';
sb += ' <td>' + value.name + '</td>';
sb += ' <td>' + value.dept + '</td>';
sb += ' <td>' + value.addr + '</td>';
sb += '</tr>';
});

```

```

table.append(sb);
}
$(function(){
$('#conBtn').click(function(e){
e.preventDefault();
submitForm();
});
});

```

```
</script>
</body>
</html>
```

2. services.jsp (will process as params requested and response data as json)

```
<% @page language="java" import="java.sql.*"%>
<% @page import="java.util.*" %>
<% @page contentType="text/html; charset=UTF-8"%>
<% @page import="org.json.simple.JSONArray"%>
<% @page import="org.json.simple.JSONObject"%>
<% @page import="org.json.simple.parser.JSONParser"%>
<% @page import="org.json.simple.parser.ParseException"%>
<%

String dept = (String)request.getParameter("dept");
String sql = "SELECT * FROM employees WHERE department='"+dept+"'";
try
{
Class.forName("com.mysql.jdbc.Driver"); // Database connectivity
Connection conn=null;
conn=DriverManager.getConnection("jdbc:mysql://localhost/jspjsons","root","123456");
ResultSet rs=null;
Statement stm1=conn.createStatement();

JSONArray list = new JSONArray();
rs=stm1.executeQuery(sql);
while(rs.next())
{
JSONObject obj=new JSONObject();
obj.put("no", rs.getString("id"));
obj.put("name", rs.getString("name"));
obj.put("dept", rs.getString("department"));
obj.put("addr", rs.getString("address"));
list.add(obj);
```

```

    }
    out.print(list);
}
catch(Exception ex)
{
    out.println("<h1>"+ex+"</g1>");
}
%>

```

3. mysql_employees_jspjson.sql (sample database structure)

```

CREATE TABLE 'employees' (
    'id' int(10) AUTO_INCREMENT,
    'name' varchar(20) ,
    'department' varchar(20) ,
    'address' varchar(50),
    PRIMARY KEY ('id')
);

```

```

INSERT INTO 'employees' VALUES ('1', 'Akbar', 'IT', '301, Pimple Anex, Pune');
INSERT INTO 'employees' VALUES ('2', 'Baban', 'COMM', 'Rno. 34, Rajat Soc, Pune');
INSERT INTO 'employees' VALUES ('3', 'Chandrakant', 'IT', 'Katepuram Chowk, Pune');
INSERT INTO 'employees' VALUES ('4', 'Dattu', 'HR', 'Akashay Plaza First, Pune');
INSERT INTO 'employees' VALUES ('5', 'Eknath', 'FINANCE', 'Ambar Apartment, Pune');

```

In the above example our MySQL Database with database name is jspjson, username root, with no password. The json-simple library is installed and setup first. The example at services can accept more params and use more fun query. The main purpose is with response as JSON data, we could retrieve it at jsp or android or anywhere we want.

Exercises:

1. What is JSON? What are the properties and applications of JSON?
2. Describe JSON objects with different kinds of examples.

3. Write a note on JSON library in Java. Explain the classes used to interact with JSON files.
4. What is GSON? What are the features of it? Write a simple program to illustrate the concept.
5. What is Jackson? Write the note on annotations used in it.

Objective Questions

1. What is full form of JSON?
 - a. Java Standard Object Notation
 - b. Java Script Oriented Numbering
 - c. Java Script Object Notation
 - d. Java Subscribed Operation Numbers
2. Which kind of data is not possible to get stored in JSON format?
 - a. Array
 - b. String
 - c. Integer
 - d. None of the above