Design Patterns

Table of Contents

What is a Design Pattern?

Design patterns are well-proved solution for solving the specific problem/task.
After multiple years of software engineering an understanding has been developed to adopt certain practices in order to achieve easily maintainable, retestable, reusable, low error prone code. It is like following a "Standard Operating Procedure" while building a software system.

Advantages of Design Patterns

1. They provide the solutions that help to define the system architecture.
2. They capture the software engineering experiences.
3. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
4. **Design patterns don't guarantee an absolute solution to a problem.**
5. They provide clarity to the system architecture and the possibility of building a better system

When should we use the design patterns?

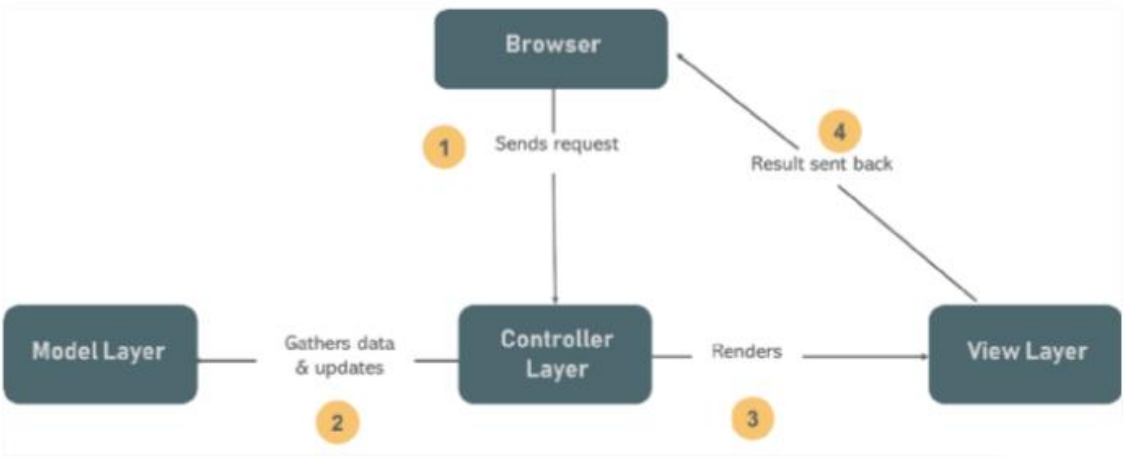
We must use the design patterns during the analysis and requirement phase of SDLC (Software Development Life Cycle).
Examples: We are going to learn about:

1. MVC Design Pattern
2. DAO Design Pattern

MVC Design Pattern

MVC design Pattern is one of the most implemented software design patterns in J2EE projects. It consists of three layers:

1. The **Model** Layer - Represents the business layer of the application
2. The **View** Layer - Defines the presentation of the application
3. The **Controller** Layer - Manages the flow of the application



Now J2EE has provided us with **3 technologies** that help us implement MVC design pattern. These are:

1. Servlets
2. JSP
3. JDBC

Servlets

- Servlets are meant to process **business logic.**
- It receives the data, processes it and produces the output.
- Servlets acts as **Controller** component
- The latest version of Servlet is 5.0

JSP

- JSP stands for **Java Server Pages**
- JSP is meant for presentational logic.
- When we want to display something to the end user, we use JSP.
- JSP acts as a **View** Component.
- The latest version of JSP is 3.0

JDBC

- JDBC stands for **Java Database Connectivity**
- JDBC connects a Java application to the database.
- It helps the java application to communicate with the database and implement CRUD operations.
- It acts as a **Model** component
- The latest version of JDBC is 4.3

DAO Design Pattern

This pattern is used to separate low level data accessing API or operations from high level business services or we can say that DAO pattern creates a persistence layer for service layer to use. Following are the participants in Data Access Object Pattern.

1. **Data Access Object Interface** - This **interface** defines the standard operations to be performed on a model object(s).
2. **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
3. **Model Object or Value Object** - This object is simple **POJO** containing get/set methods to store data retrieved using DAO class.

Implementation

We are going to create a Student object acting as a Model or Value **Pojo.StudentDao** is Data Access Object Interface.StudentDaoImpl is concrete class implementing Data Access Object Interface. DaoPatternDemo, our demo class, will use StudentDao to demonstrate the use of Data Access Object pattern.

Java Database Connectivity

Table of Contents

What is JDBC?

- JDBC stands for **J**ava **D**ata**B**ase **C**onnectivity.
- JDBC is a **Java API** used to connect and execute queries with the database.
- JDBC API consists of a set of **classes**, **interfaces** and **methods** to work with databases
- JDBC can be used to interact with every type of RDBMS such as MySQL, Oracle, Apache Derby, MongoDB, PostgreSQL, Microsoft SQL Server etc.
- It is a **part of JavaSE** (Java Platform, Standard Edition).
- JDBC API uses **JDBC drivers** to connect with the database.
- The current version of JDBC is **4.3**.

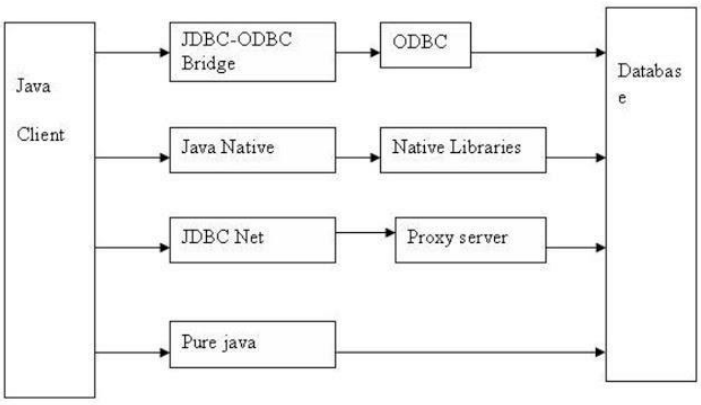
What is a Driver?

A driver is nothing but a piece of software required to connect to a database from Java program. JDBC drivers are client side adapters (**installed on the client machine**, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.

Types of JDBC Drivers

There are four types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver



Pictorial representation of JDBC Drivers

Type-1 Driver

- This is the oldest JDBC driver, mostly used to connect database like MS Access from Microsoft Windows operating system.
- Type 1 JDBC driver actually translates JDBC calls into ODBC (Object Database connectivity) calls, which in turn connects to database.
- It converts the JDBC method calls into ODBC function calls.

Pros:

- Any database that provides an ODBC driver can be accessed

Cons:

- Features are limited and restricted to what ODBC driver is capable of
- Platform dependent as it uses ODBC which in turn uses **native O/S libraries**
- ODBC driver must be installed on client machine
- **Limited portability** as **ODBC driver is platform dependent** & may not be available for all platforms
- **Poor Performance** because of several layers of translation that take place before the program connects to database
- It is now **obsolete** and only used for development and testing.
- It has been removed from JDK 8 (1.8)

Type-2 Driver

- This was the second JDBC driver introduced by Java after Type 1, is hence it known as type 2.
- In this driver, performance was improved by **reducing communication layer**.
- Instead of talking to ODBC driver, **JDBC driver directly talks to DB** client using **native API**.
- That's why it's also known as **native API or partly Java driver**
- Type 2 drivers use the client side libraries of the database.
- The driver converts JDBC method calls into native database API calls.

Pros:

- Faster than JDBC-ODBC bridge as there is no conversion like ODBC involved
- Since it required native API to connect to DB client it is also less portable and platform dependent.

Cons:

- Client side libraries needs to be installed on client machine
- Driver is platform dependent
- Not all database vendors provide client side libraries
- Performance of type 2 driver is slightly better than type 1 JDBC driver.

Type-3 Driver

- This was the third JDBC driver introduced by Java, hence known as type 3.
- Type 3 driver makes use of middle tier between the Java programs and the database.
- Middle tier is an **application server** that converts JDBC calls into vendor-specific database calls.
- It was very different than type 1 and type 2 JDBC driver in sense that **it was completely written in Java** as opposed to previous two drivers which were not written in Java.
- That's why this is also known as all **Java driver**.
- This driver uses **3 tier approach i.e. client (java program), server and database**.
- So you have a Java client talking to a Java server and Java Server talking to database.
- Java client and server talk to each other using net protocol hence this type of JDBC driver is also known as **Net protocol JDBC driver**.
- This driver **never gained popularity** because database vendor was reluctant to rewrite their existing native library which was mainly in C and C++

Pros:

- No need to install any client side libraries on client machine
- Middleware application server can provide additional functionalities
- Database independence

Cons:

- Requires middleware specific configurations and coding
- May add extra latency as it goes through middleware server

Type-4 Driver

- Type 4 drivers are also called **Pure Java Driver**.
- This is the driver you are most likely using to connect to modern database like Oracle, SQL Server, MySQL, SQLLite and PostgreSQL.
- This driver is implemented in Java and directly speaks to database using its native protocol.
- It converts JDBC calls directly into vendor-specific database protocol.
- This driver includes all database calls in **one JAR file**, which makes it very easy to use.
- All you need to do to connect a database from Java program is to include JAR file of relevant JDBC driver.
- Because of **light weight**, this is also known as **thin JDBC driver**.
- Since this driver is also written in **pure Java**, **it is portable across all platforms**, which means you can use same JAR file to connect to MySQL even if your Java program is running on Windows, Linux or Solaris.
- Performance of this type of JDBC driver is also best among all of them because **database vendor liked this type** and all enhancements they make they also port for type 4 drivers.

Pros:

- Written completely in Java hence platform independent
- Provides better performance than Type 1 and 2 drivers as there is no protocol specific conversion is required
- Better than Type 3 drivers as it doesn't need additional middleware application servers
- Connects directly to database drivers without going through any other layer

Cons:

- Drivers are database specific

Questions and Answers

Question 1: Which driver should we use?

Answer:

- A Type 4 driver is preferred if Java application is accessing any 1 database such as Oracle, Sybase etc.
- In case multiple databases are accessed then a Type 3 driver would be preferable.
- Type 2 drivers are recommended, if Type 3 or 4 drivers are not available for the database.
- Type 1 drivers are not recommended for production deployment.

Question 2: What factors decide the choice of drivers?

Answer: There are 2 factors that decide the choice of drivers: **portability** and **performance**.

Question 3: Which driver is least used?

Answer: Type 1 JDBC driver is the poorest in terms of portability and performance. **It is no longer used.**

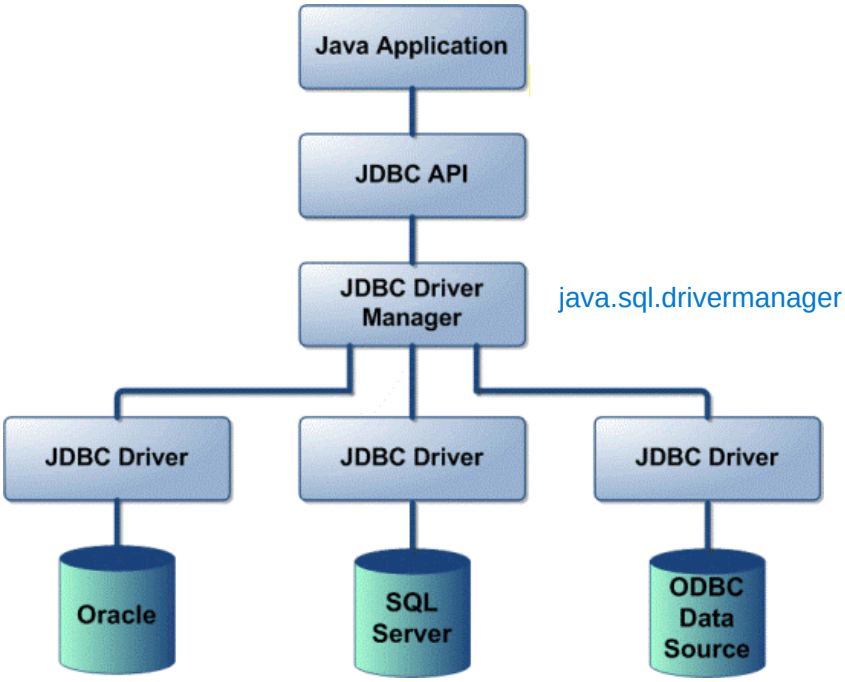
Question 4: Which is the best driver?

Answer: Type 4 JDBC driver is highly portable and gives the best performance.

Question 5: Which driver do we use to connect and transact with MySQL DB?

Answer: We use Type 4 driver

JDBC Architecture



1. **Java Application** is any application that likes to connect and transact with any database.
2. **JDBC API**
    1. It provides the application-to-DB Connection
    2. It provides the driver manager (**java.sql.DriverManager**)
    3. It uses the database specific driver to connect to heterogeneous databases.
3. **Driver Manager**
    1. The JDBC driver manager ensures that the **correct driver is used** to access each data source
    2. The driver manager is capable of supporting **multiple concurrent drivers** connected to **multiple heterogeneous databases**.
    3. One application can connect to different databases simultaneously.
4. **JDBC driver API** supports the JDBC Database connection.
    1. **Database vendors** provide the **JDBC drivers.**
    2. For example: MySQL vendor provides "**mysql-connector-java-8.0.19"** jar file that contains "**com.mysql.cj.jdbc.Driver**"

5. **Databases**
    1. **A Java application** can connect and transact with **multiple databases** simultaneously or one at a time.
    2. The vendors provide their specific drivers
    3. The Driver Manager takes care of all the drivers

Operations With DB

The following are the key operations we do with a database frequently.

1. Connect to DataBase
2. Execute Queries
    1. **C**reate/Insert Data
    2. **R**etrieve Data
    3. **U**pdate Data
    4. **D**elete Data
3. Close Connections/Resources

JDBC 4.3 API

JDBC 4.0 API is mainly divided into two package

- java.sql
- javax.sql

java.sql package

This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

Important classes and interface of java.sql package

classes/interface	Description
java.sql.BLOB	Provide support for BLOB(Binary Large Object) SQL type.
java.sql.Connection	Creates a connection with specific database
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB(Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	Create an instance of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result row-by-row.
java.sql.Savepoint	Specify savepoint in transaction.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statements.

First JDBC Program

A succesfull operation/transaction with the database involves the exceution of **several small steps.** Each step has a **significance** of its own. To write efficient database code, all these steps must be correctly implemented. These steps are:

- **Step 1.** Pre-requisites
- **Step 2.** Connect to database
- **Step 3.** Create Statement
- **Step 4.** Prepare the Query
- **Step 5.** Execute the Query and Collect Data
- **Step 6.** Close Resources

Lets ponder over the above steps one by one

Step 1. Pre-requisites

Before writing a JDBC API, we need to:

1. Install any database server (here we will use **MySQL** Server).
2. Install a **GUI to operate on** the database server (here, MySQL GUI - **MySQL Workbench**)
3. While installation, set a **username** and a **password** (We set username: **root** and password: **1234**)
4. Create a schema (We start with **schema - school**)
5. Create the first table (We create **table - students**)
6. Create columns

No	lumn Name	lumn Properties
	l	, Primary Key, Not Null, Unique, Auto Increment
	dent_name	rchar (45), Not Null
	dent_class	(2), Not Null
	dent_fees	uble, Not Null

7. Make sure that MySQL Server is running before writing the JDBC API.
8. Place the suitable connector jar file in **WEB-INF>>lib** folder. (We use **mysql-connector-java-8.0.23.jar**)

Step 2. Connect to Database

1. Import the correct packages.

```
port java.sql.*;
```

2. Load the JDBC Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

where **com.mysql.cj.jdbc.Driver** is the location of Driver Class

3. Establish the connection

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/<schema-name>",
"<username>", "<password>");
```

where jdbc:mysql://localhost:3306/<schema-name> is **Connection String**

Step 3. Create a Statement

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

Step 4. Prepare the query

```
String insertQuery = "INSERT INTO students (student_name, student_class, student_fees) values('Abhishek Verma', 1, 5000.0)";
```

Step 5. Execute the query and collect data

```
noOfRowsInserted = stmt.executeUpdate(insertQuery);
```

Step 6. Close the Resources

```
statement.close();
nnection.close();
```

Connect to Database

The first step before we can do any database transactions is to **connect with the database.**

JDBC API to Connect with DB

1. Import the correct packages.

```
port java.sql.*;
```

2. Load the JDBC Driver

The first thing you need to do before you can open a JDBC connection to a database is to load the JDBC driver for the database. You load the JDBC driver like this:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

where **com.mysql.cj.jdbc.Driver** is the location of Driver Class

- You only have to load the driver **once for the whole application**.
- You do not need to load it before every connection opened.
- Only before the first JDBC connection opened.

3. Establish the connection

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/<schema-name>", "<username>", "<password>");
```

where jdbc:mysql://localhost:3306/<schema-name> is **Connection String**

The following classes and interfaces are used to connect to the database:

S. No	Class/Interface Name	Function
1	Class (C)	Class is a class in which all the drivers should be registered which will be used by the Java Application
2	forName(String)	forName() is a static method in class Class which loads and register our driver dynamically (by calling DriverManager.registerDriver())
3	Connection (I)	The JDBC Connection class, java. sql. Connection , represents a database connection to a relational database. Before we can read or write data from and to a database via JDBC, we need to open a connection to the database.
4.	DriverManager (C)	The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method Class.forName() that calls DriverManager.registerDriver() automatically
5.	getConnection (String, String, String)	This method of Java DriverManager class attempts to establish a connection to the database by using the given database url, username and password

Close the Resources

You should explicitly close *Statements*, *ResultSets*, and *Connections* when you no longer need them, unless you declare them in a *try*-with-resources statement (available in JDK 7 and after).

To close a Statement, ResultSet, or Connection object that is not declared in a **try-with-resources** statement, use its close method.

```
resultSet.close();
statement.close();
nnection.close();
```

Statement

- Once a connection is obtained we can interact with the database.
- The JDBC framework provides **3 interfaces** and related methods and properties with which we can send SQL or PL/SQL commands to operate on the database.
- These interfaces are:
    1. **Statement interface**
    2. **PreparedStatement interface**

    Let us look at them one by one

Statement (I)

1. Statement is an **Interface** in Java (JDBC)
2. It is present in java.sql package
3. We can obtain a JDBC Statement from a JDBC Connection
4. It is used to execute **static SQL statements** at runtime against an RDBMS.
5. It can be used to execute SQL **DDL** statements, for example data retrieval queries (**SELECT**)
6. It can be used to execute SQL **DML** statements, such as **INSERT**, **UPDATE** and **DELETE**

Syntax

```
Statement stmt = null;
try {
  stmt = connection.createStatement( ); // conn is Connection object
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  if(stmt!=null)
    stmt.close();
}
```

CRUD Operations

Create Operation – INSERT

```
Statement statement = connection.createStatement();


String insertQuery = "INSERT INTO students (student_name, student_class, student_fees) values('Kinjal', 1,
5000.0)";


int noOfRowsInserted = statement.executeUpdate(insertQuery);
```

Retrieve Operation – SELECT

```
Statement statement = connection.createStatement();


String selectQuery = "SELECT * FROM students";
ResultSet resultSet = statement.executeQuery(selectQuery);
while (resultSet.next()) {
   resultSet.getInt("_id"); // resultSet.getInt(1);
   resultSet.getString("student_name"); // resultSet.getString(2);
```

```
    resultSet.getInt("student_class"); // resultSet.getInt(3);
    resultSet.getDouble("student_fees"); // resultSet.getDouble(4);
}
```

Update – UPDATE

```
Statement statement = connection.createStatement();

String updateQuery = "UPDATE students SET student_class = 12 WHERE student_class = 11";

int noOfRowsUpdated = statement.executeUpdate(updateQuery);
```

Delete – DELETE

```
Statement statement = connection.createStatement();

String deleteQuery = "DELETE FROM students WHERE student_class = 12";

int noOfRowsDeleted = statement.executeUpdate(deleteQuery);
```

Methods

int executeUpdate(String sqlQuery)

This method is used to execute a DML SQL Query (Insert, Updaate and Delete queries)

ResultSet executeQuery(String sqlQuery)

This method is used to retrieve data from the table and returns a result set

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

It has the following steps:

Create Statement

Step 4. Execute SQL command

Execute the Query via the Statement

You do so by calling its **executeQuery()** method, passing an SQL statement as parameter.

Step 5. Collect Information

There are different types of operations that we do with the DB such as insert, update, delete and retrieve. In every case we get different informations.

- In case of insert, update and delete operations, we get number of records inserted, updates or deleted.
- In case of retrieval, we get rows of data in a ResultSet

ResultSet

1. ResultSet is an Interface in Java (JDBC)
2. It is used to hold records that are returned as a result of a SELECT query
3. We need to travel/traverse/iterate through the ResultSet to get records

Create ResultSet

```
ResultSet resultSet = statement.executeQuery(sql);
```

Get the data from ResultSet

1. To iterate the ResultSet you use its **next()** method.
2. The next() method returns **true** if the ResultSet has a next record, and moves to the next record.
3. If there were no more records, next() returns **false**, and you can no longer extract data.

```
while(resultSet.next()) {
    System.out.println("Id = " + resultSet.getInt(1));
    System.out.println("Name = " + resultSet.getString(2));
System.out.println("Class = " + resultSet.getInt(3));
```

Step 5. Close the Resources

```
resultSet.close();
statement.close();
nnection.close();
```

Web Server Architecture

Table of Contents

J2EE

J2EE is a set of rules and protocols given by Sun/Oracle also known as specifications

Server Runtime Environment

Server vendors provide J2EE specification's implementation. There are many such implementations in the industry such as:

1. Apache Tomcat
2. J BOSS
3. Web sphere
4. Glassfish
5. Wild Fly

Server

A piece of hardware where we put our website for everyone to access it 24 by 7 is known as a Server Machine and the operating system that manages the incoming requests, takes the request to appropriate resource and send back the prepared response is known as Server Software. Both the server machine and server software are jointly called as Server. **The Server Software is also known as Web Server**

Web Server (HTTP Server)

- A Web Server handles HTTP requests sent by a client and responds back with an HTTP response
- It provides an **environment** in which requests and responses are handled and web pages are viewed.
- A Web server is also known as an **HTTP server**

There are many web servers in the industry, for example:

- Apache HTTP Server – **Tomcat**
- Internet Information System (IIS)
- Lighttpd, Resin, Jigsaw
- XAMPP, WAMP, LAMP etc.

The most used Web Server is **Apache Tomcat.**

Web Container

- A Web Container is used to generate dynamic content on user request
- It is server side JVM that **resides** either **inside** a **web server** to handle requests to servlets, JSPs, and other types of files that include server-side code.
- The web container
  - Creates servlet instances
  - Loads and Unloads servlets
  - Creates and manages request and response objects
  - Performs other servlet-management tasks.
- The Web Container is also known as **Servlet Container or J2EE container**

Tomcat Web Server

What is Tomcat?

Apache Tomcat is an open-source **web server** and **servlet/jsp container** developed by the Apache Software Foundation (ASF).
Tomcat implements several Java EE specifications including Java Servlet, Java Server Pages (JSP), Java EL, and WebSocket, and provides a “pure Java” HTTP web server environment for Java code to run in

Is Tomcat a Web Server or a Web Container?

Tomcat is **both** a web server (supports HTTP protocol) and a web container (supports JSP/Servlet API, also called "**servlet container**" at times).

The Tomcat Web Container contains 2 components:

- Servlet Container
- JSP Container

Servlet Container

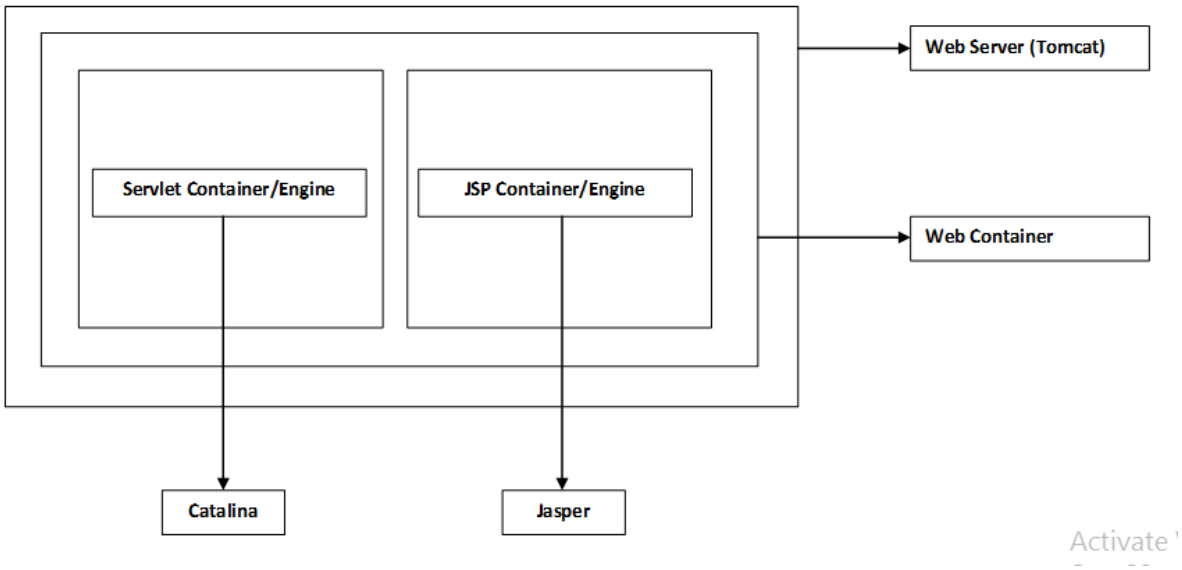
- A servlet container is essentially a part of the web container that **interacts** with the **servlets**.
- The Servlet Container is also known as **Servlet Engine**
- In Tomcat, the name of the Servlet Container Component is **Catalina**

JSP Container

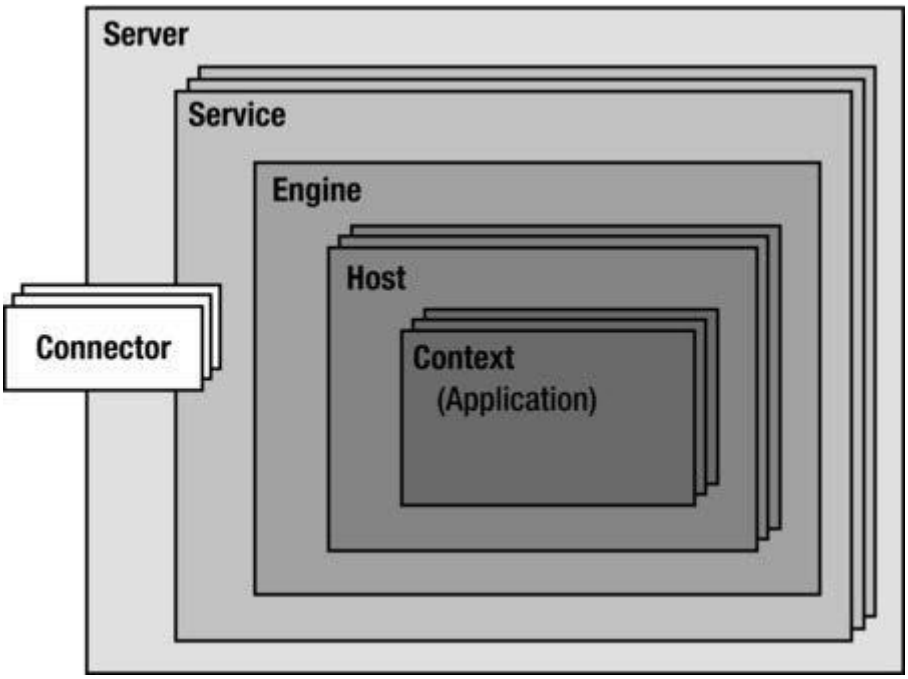
- A JSP container is a part of a web container that **interacts** with **JSP** requests.
- The JSP Container is also known as **JSP Engine**

- In Tomcat, the name of the JSP Container Component is **Jasper**

Schematic Diagram of Tomcat Web Server



Tomcat Architecture



The structure of each server installation (via these functional components) is defined in the file **server.xml**, which is located in the /**conf** subdirectory of Tomcat's installation folder.

Tomcat Port

By default, Tomcat is configured to run on port **8080**. That's why all the deployed web applications are accessible though URLs like http://localhost:8080/yourapp

How to Change port

To make this port change, open **server.xml** and find below entry:

```
<Connector
        port="8080"          !! Change this line
        protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />
```

Servlets API & Ways to create a Servlet

Table of Contents

What is a Servlet?

Servlets are an overwhelming technology; it is not one but many things. It can be defined in many ways as below depending on the context:

- Servlet is a technology which is used to create a web application.
- Servlet is an **API** that provides many **interfaces** and **classes** to receive request and send response.
- Servlet is **an interface** that must be implemented for creating any Servlet.
- Servlet **extends the capabilities** of the servers and responds to the incoming requests.
- Servlet is a **web component** that is deployed on the server to create a dynamic web page.
- Servlets are ***server-side programs*** that run inside a *Java-capable* HTTP server (ex: Apache Tomcat) that handle clients' requests and return a ***customized*** or ***dynamic response*** for each request.

This dynamic response could be based on user's input such as

- Search
- Online shopping
- Online transaction
- Chat, Like, Share
- Submit a form etc.

Servlet API

The **Apache Tomcat software** is an **open source implementation** of the

1. Java Servlet
2. Java Server Pages
3. Java Expression Language
4. Java WebSocket
5. Java Annotations, and

Important Points:

- These specifications are part of the **Java EE platform**.
- The Java EE platform is the **evolution of** the **Java EE** platform.
- **Tomcat 10 and later** implement specifications developed as part of **Jakarta EE.**
- **Tomcat 9 and earlier** implement specifications developed as part of **Java EE.**

Servlet API

1. javax.servlet
2. javax.servlet.annotation
3. javax.servlet.descriptor
4. javax.servlet.http

We will discuss the most important packages here: **java.servlet** and **java.servlet.http**

java.servlet

It is the core package of the Servlet API, here is a list of all its important interfaces, classes and methods:

S. No	Interface/ Class	Description
1.	Servlet (I)	Defines methods that all servlets must implement.
2.	ServletConfig (I)	A servlet configuration object is used by a servlet container to pass information to a servlet during initialization.
3.	ServletContext (I)	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
4.	ServletRequest (I)	Defines an object to provide client request information to a servlet. The servlet container creates a ServletRequest object and passes it as an argument to the servlet's service method.
5.	ServletResponse (I)	Defines an object to assist a servlet in sending a response to the client. The servlet container creates a ServletResponse object and passes it as an argument to the servlet's service method.
6.	RequestDispatcher (I)	Defines an object that receives requests from the client and sends them to any resource (such as a Servlet, HTML file, or JSP file) on the server
7.	GenericServlet (C)	Defines a generic, protocol-independent servlet.

We will be learning about every one of them in detail later in the tutorial

java.servlet.http

It contains the following classes and interfaces that hold the responsibility of handling HTTP requests and sending out HTTP responses:

S. No	Interface/Class	Description
1.	HttpServlet (C)	Provides an abstract class to be sub-classed to create an HTTP servlet suitable for a Web site.
2.	HttpServletRequest (I)	Extends the ServletRequest interface to provide request information for HTTP servlets
3.	HttpServletResponse (I)	Extends the ServletResponse interface to provide HTTP-specific functionality in sending a response.
4.	Cookie (C)	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.
5.	HttpSession (I)	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

We will be learning about every one of them in detail later in the tutorial

How to create a Servlet?

As we know that a Servlet is used to exchange requests and response with a client. In order to do so, we need to create a **user defined servlet** and define request and response handling mechanism.

We can create a servlet in the following 3 ways:

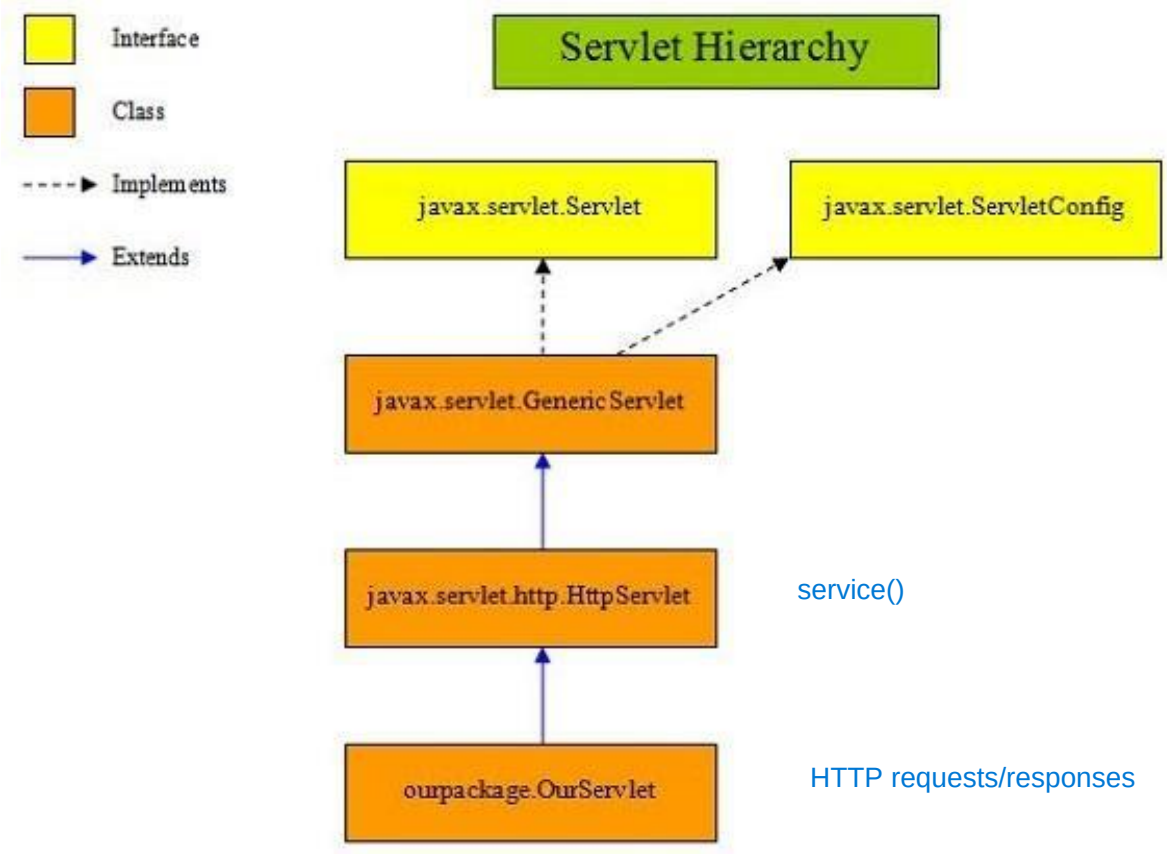
1. By implementing Servlet interface
2. By inheriting GenericServlet class
3. By inheriting HttpServlet class

Before studying each of them in brief, it is important to see the relationship between all the 3 interfaces and classes.

Servlet API Hierarchy

The below diagram depicts:

1. Servlet is an interface. It has **5** abstract methods
2. GenericServlet is a class. It **implements** Servlet Interface and defines 4 of its methods
3. HttpServlet **extends** GenericServlet class. It defines the last remaining method service()
4. Our custom servlet must **extend** HttpServlet in case we need to deal with HTTP requests/responses



Now, as we have got an abstract look at the hierarchy, let's look the 3 ways of creating a servlet one by one.

Implementing Servlet Interface

Servlet Interface

Servlet is the main interface that declares methods that all servlets must implement.

Main Points

The main points about the Servlet are:

- Servlet is an interface
- Fully qualified name: javax.servlet.Servlet
- Signature: public interface Servlet {}
- Extends: Nothing
- Implemented by: GenericServlet class

Abstract Methods in Servlet Interface

1. public abstract void **init**(ServletConfig config) throws ServletException
2. public abstract void **service**(ServletRequest req, ServletResponse res) throws ServletException, IOException
3. public abstract void **destroy**()
4. public abstract ServletConfig **getServletConfig**()
5. public abstract String **getServletInfo**()

Create a Servlet

Step 1: Define a normal class that implements java.servlet.Servlet

```
class FirstServlet implements Servlet {


}
```

Step 2: Override all the abstract methods of Servlet interface

```
public class FirstServlet implements Servlet {

	ServletConfig servletConfig;

	@Override
	public void init(ServletConfig servletConfig) throws ServletException {
		this.servletConfig = servletConfig;
		System.out.println("Servlet is initialized");
	}

	@Override
	public void service(ServletRequest req, ServletResponse res) throws
```

```
    ServletException, IOException {
                System.out.println("Request is being serviced...");
        }


        @Override
        public void destroy() {
                System.out.println("Servlet is destroyed");
        }


        @Override
        public ServletConfig getServletConfig() {
                return this.servletConfig;
        }


        @Override
        public String getServletInfo() {
                return "FirstServlet";
        }


}
```

Note: Put SOPs inside the overridden methods to track the servlet lifecycle

Step 3: Map the servlet in web.xml

```
<servlet>
        <servlet-name>first</servlet-name>
        <servlet-class>com.java.servlets.FirstServlet</servlet-class>
</servlet>


<servlet-mapping>
        <servlet-name>first</servlet-name>
        <url-pattern>/first</url-pattern>
</servlet-mapping>
```

Step 4: Execute the project

Start the Server and run the project, type: **localhost:8080/02ServletCreationTechniques/first** to run the Servlet

Step 5: Track the Output

- init() will run once when the first request is made

- service() will run on every request
- destroy () will be called when we stop the server

Observations

- init (), service() and destroy() are lifecycle methods
- Lifecycle methods are called implicitly by the web container.
- Their sequence and mode of calling depends on the state of the servlet
    - When the 1st request is made, the web container initializes the servlet by calling init ()
    - Every request to the servlet triggers the service()
    - To call destroy , we have to explicitly stop the server
- getServletConfig() and getServletInfo() are non-life cycle methods
- These methods are not called implicitly & have to be called explicitly

Note: You can find the above example in Project: 02ServletCreationTechniques

Inheriting GenericServlet Class

GenericServlet Class

GenericServlet defines a generic, protocol-independent servlet. It must be used to handle any type of request

Main Points

The main points about the GenericServlet are:

- GenericServlet is a class
- Fully qualified name: javax.servlet.GenericServlet
- Signature: public abstract class GenericServlet
- Extends: java.lang.Object
- Implements: Servlet, ServletConfig, Serializable
- Sub-classes: HttpServlet

Creating the Servlet

Step 1: Create a new class and extend it from GenericServlet

```
public class SecondServlet extends GenericServlet {


}
```

Step 2: Override unimplemented service () of GenericServlet class

GenericServlet class provides implementation of 4 methods of Servlet interface, however service() still remains unimplemented.

```
public class SecondServlet extends GenericServlet {


    private static final long serialVersionUID = 1L;
```

```
        @Override
        public void service(ServletRequest req, ServletResponse res) throws
    ServletException,IOException {
                System.out.println("Request is being serviced");
        }
}
```

Step 3: Execute the project & we will notice that service is triggered whenever there is a new request

Note: You can find the above example in Project: 02ServletCreationTechniques

Inheriting HttpServlet class

HttpServlet Class

- The websites communicate with the server through HTTP protocol
- If we make a web application, we extend our Servlet from HttpServlet
- We can say that HttpServlet class is used to create http protocol specific servlet.
- HttpServlet class is abstract although all its methods are concrete. This is done to force its child class to override request specific doXXX() method

Main Points

The main points about the HttpServlet are:

- HttpServlet is a class
- Fully qualified name: java.servlet.http.HttpServlet
- Signature: public abstract class HttpServlet
- Extends: java.servlet.GenericServlet
- Implements: Servlet, ServletConfig, Serializable (**all indirectly through GenericServlet**)
- Predefined Sub-classes: No

HttpServlet class methods

A subclass of HttpServlet must override at least one method, usually one of these:

1. doGet(), for HTTP GET requests
2. doPost(), for HTTP POST requests
3. doPut(), for HTTP PUT requests
4. doDelete(), for HTTP DELETE requests
5. getServletInfo(), which the servlet uses to provide information about itself
6. service(): There's no reason to override service() as it passes HTTP requests to HttpServlet handler methods

Creating the Servlet

Step 1: Create a new class and extend it from HttpServlet

```
public class ThirdServlet extends HttpServlet {


}
```

Step 2: Override any one method that you wish to be handled

```
public class ThirdServlet extends HttpServlet {


    private static final long serialVersionUID = 1L;


        @Override
        protected void doGet(HttpServletRequest req, HttpServletResponse
    resp) throws ServletException, IOException {


                System.out.print("Request is being processed");
        }
}
```

Step 3: Execute the project & we will notice that doGet is triggered for get request

Note: You can find the above example in Project: 02ServletCreationTechniques

Serial Version UID

GenericServlet and HttpServlet classes implement **java.io.Serializable** interface as we can see in class definitions. Whenever we create a servlet by extending these classes, we need to provide a SerialVersionUid. To understand its significance, we must have familiarity with the concept of serialization and deserialization.

Serialization

Serialization is a mechanism of converting the state of an object into a byte stream.

Deserialization

Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

SerialVersionUid

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization.

If there is a mismatch in UIDs of serialized and de-serialized classes, the deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.

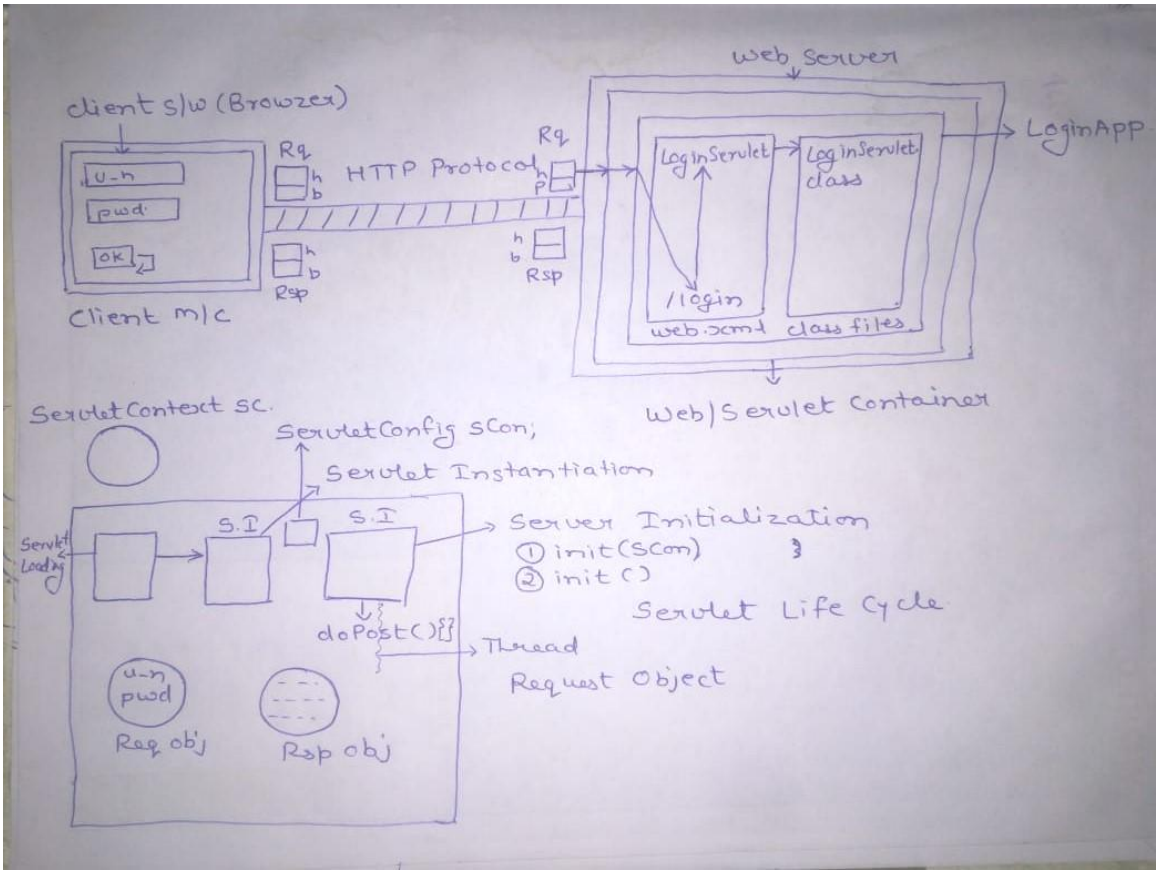
It must be static, final and of type long: i.e. private static final long serialVersionUID=42L;

Note: You can find the above example in Project: 02ServletCreationTechniques

Servlet Work Flow and Life Cycle

Table of Contents

Servlet Workflow



Example Application

To understand Servlet Workflow, consider an example application. This example includes:

- A Web Application named as **LoginApplication**
- This Web Application contains an html page **login.html**, a servlet **LoginServlet** and a **web.xml** file
- This login form will have 2 fields: **user_name** and **password**.
- The form tag contains attributes: method="**post**", action="**login**"
- The servlet LoginServlet has URL pattern **"/login**"

Starting the Server

Task 1 of Web Server

When we start the web server the job of the web container is:

- To recognize all the web applications
- Deploy each web application on the server, and
- Prepare a separate object for each web application.
- This object is known as **ServletContext** Object.

For example, as we start **Tomcat Web Server**, the web container

- Recognizes the Login application
- Deploy it to the server and
- Prepare an object for Login Application called **ServletContext** Object.

Task 2 of Web Server

The web container also

- Recognizes the web.xml file present inside the WEB-INF folder.
- Loads the web.xml file
- Parses the file and
- Reads the content of the file.
- If any application level data is available, the container will store this data inside the ServletContext object.

Request is made

Scenario 1: A static page is requested

The Web Server is responsible to deliver static content to the client. So, as a request is made from the client browser, the requested static page is delivered through the web server to the client browser.

Scenario 2: A dynamic page is requested

Prerequisite

1: A page is displayed in the browser.

2: The page contains a login form with 2 input fields: username and password.

Step 1: The data is entered in the form fields and button is clicked.

HTTP Protocol

- The request will come to protocol (here HTTP).
- Protocol will establish a virtual socket connection between client and server on the basis of server IP address and port number.
- After preparing the socket connection, the protocol will prepare a request format.
- The request format contains 2 parts:
  - Header part
  - Body part.

Header Part

The header will manage **request headers.** Request headers contain **details about the client browser** such as:

- The Locale managed by the client browser
- The Zipping formats supported by the client browser
- The Encoding mechanism supported by the client browser.

Body Part

The body part will manage **request parameters** data. The request parameters data is nothing but the data provided by the user through the form.

Request reaches the Web Server

- Whenever the request format is prepared, the protocol will carry this request format to web server.

- This request format is given to main web server.

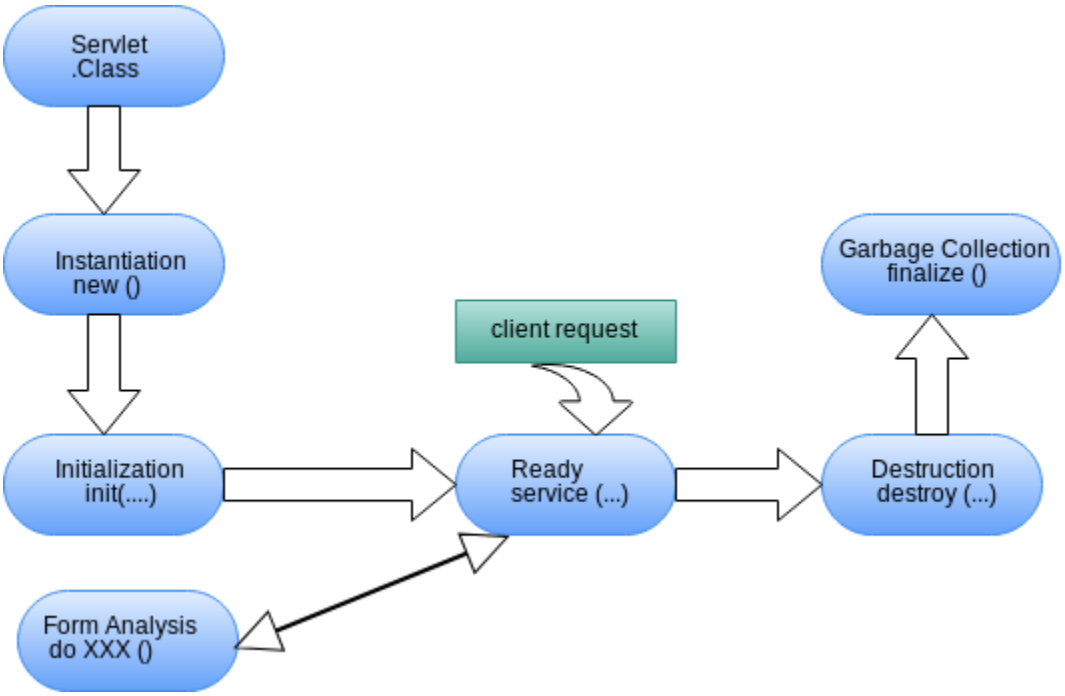
Web Server checks the request

- The web server will first check whether it is a valid request or not.
- If the request is valid, the web server will forward the request to the web container.
- The web container will identify the name of the requested application and of the requested resource.
- The information required by container is mentioned in the form tag in **action** attribute. It contains either the **name** of the **servlet** or the **URL pattern** of the requested servlet.

Web Server locates the resources

- The web container will go to our Login Application Folder and then to web.xml present inside WEB-INF folder.
- In web.xml, mapping details are available i.e. the mapping between the URL pattern (/login) and the servlet name (LoginServlet).
- The container identifies the LoginServlet resource from web.xml.
- Now the container searches for **.class** files in class' folder for the particular servlet.
- As the container finds the LoginServlet.class file, it will begin the **Servlet Life Cycle.**

Servlet Life Cycle



1. Servlet Loading.

Container will load LoginServlet.class file's bytecode into the memory i.e. perform Servlet loading.

```
Class c = Class.forName("LoginServlet.class")
```

Before loading the servlet, the container will first check whether this particular servlet is loaded beforehand or not. If the servlet is not loaded, then only the container will perform the loading of the servlet.

2. Servlet Instantiation

After LoginServlet class loading, the container will prepare an object of LoginServlet (Servlet Instantiation).

Object obj = c.newInstance()

3. Servlet Initialization

After Servlet Instantiation, the web container will perform Servlet Initialization. Here the container will call init() on Servlet and pass ServletConfig object in it.

- init (ServletConfig servletConfig)

Server Initiatlization: init(ServletConfig s)

This method belong to **GenericServlet class.**

- The servlet container calls the init method exactly once after instantiating the servlet.
- The init method must complete successfully before the servlet can receive any requests.
- The servlet container will call service () after init is completed

4. Thread Creation and Data Processing

service(ServletRequest, ServlestResponse) in **GenericServlet** class
service(HttpServletRequest, HttpServletResponse) in **HttpServlet** class
doXXX(HttpServletRequest, HttpServletResponse)in **LoginServlet** class

- As soon as the servlet initialization is complete, the web container will spawn a new thread and call the service () method.
- The service () method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.
- As the form contains get method, we must implement the doGet() in LoginServlet class.
- The container prepares HttpServletRequest and HttpServletResponse objects.
- The HTTPServletRequest object will contain 3 types of data
  - Request **Headers** Data: the client browser data
  - Request **Parameters** Data: the data that we entered in form: u_n, pwd
  - Request **Attributes** Data: the dynamic data the servlet may include in request attributes

5. Response is prepared

- As container executes the doPost(), some data is created in Response object.

- As the response is created and the thread reaches the end point, it service() and doPost() ends
- As the thread is near destroy, the servlet sends the response object to the main server.
- Main server will format the response and give it to protocol and this protocol prepares the header and body part of response.
    - Header will contain type of response, length of response etc.
    - The body will contain dynamic data.

6. Response is sent back

- The protocol will carry this data to client machine.
- Browser will get response from body part and parse and display the result.
- As the information is displayed, the protocol will destroy the connection.
- As soon as the connection is destroyed the server will destroy the request and response objects.
- The container will again come in waiting state.

Brief Life cycle

- init() is called when the request is made for the 1st time
- service() is called on every request
- destroy() is called, when the server stops which in turn destroys all servlets

Servlets API & Ways to create a Servlet

Table of Contents

What is a Servlet?

Servlets are an overwhelming technology; it is not one but many things. It can be defined in many ways as below depending on the context:

- Servlet is a technology which is used to create a web application.
- Servlet is an **API** that provides many **interfaces** and **classes** to receive request and send response.
- Servlet is **an interface** that must be implemented for creating any Servlet.
- Servlet **extends the capabilities** of the servers and responds to the incoming requests.
- Servlet is a **web component** that is deployed on the server to create a dynamic web page.
- Servlets are ***server-side programs*** that run inside a *Java-capable* HTTP server (ex: Apache Tomcat) that handle clients' requests and return a ***customized*** or ***dynamic response*** for each request.

This dynamic response could be based on user's input such as

- Search
- Online shopping
- Online transaction
- Chat, Like, Share
- Submit a form etc.

Servlet API

The **Apache Tomcat software** is an **open source implementation** of the

1. Java Servlet
2. Java Server Pages
3. Java Expression Language
4. Java WebSocket
5. Java Annotations, and

Important Points:

- These specifications are part of the **Java EE platform**.

- The Java EE platform is the **evolution of** the **Java EE** platform.
- **Tomcat 10 and later** implement specifications developed as part of **Jakarta EE.**
- **Tomcat 9 and earlier** implement specifications developed as part of **Java EE.**

Servlet API

1. javax.servlet
2. javax.servlet.annotation
3. javax.servlet.descriptor
4. javax.servlet.http

We will discuss the most important packages here: **java.servlet** and **java.servlet.http**

java.servlet

It is the core package of the Servlet API, here is a list of all its important interfaces, classes and methods:

S. No	Interface/ Class	Description
1.	Servlet (I)	Defines methods that all servlets must implement.
2.	ServletConfig (I)	A servlet configuration object is used by a servlet container to pass information to a servlet during initialization.
3.	ServletContext (I)	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
4.	ServletRequest (I)	Defines an object to provide client request information to a servlet. The servlet container creates a ServletRequest object and passes it as an argument to the servlet's service method.
5.	ServletResponse (I)	Defines an object to assist a servlet in sending a response to the client. The servlet container creates a ServletResponse object and passes it as an argument to the servlet's service method.
6.	RequestDispatcher (I)	Defines an object that receives requests from the client and sends them to any resource (such as a Servlet, HTML file, or JSP file) on the server
7.	GenericServlet (C)	Defines a generic, protocol-independent servlet.

We will be learning about every one of them in detail later in the tutorial

java.servlet.http

It contains the following classes and interfaces that hold the responsibility of handling HTTP requests and sending out HTTP responses:

S. No	Interface/Class	Description
1.	HttpServlet (C)	Provides an abstract class to be sub-classed to create an HTTP servlet suitable for a Web site.
2.	HttpServletRequest (I)	Extends the ServletRequest interface to provide request information for HTTP servlets
3.	HttpServletResponse (I)	Extends the ServletResponse interface to provide HTTP-specific functionality in sending a response.
4.	Cookie (C)	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.
5.	HttpSession (I)	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

We will be learning about every one of them in detail later in the tutorial

How to create a Servlet?

As we know that a Servlet is used to exchange requests and response with a client. In order to do so, we need to create a **user defined servlet** and define request and response handling mechanism.

We can create a servlet in the following 3 ways:

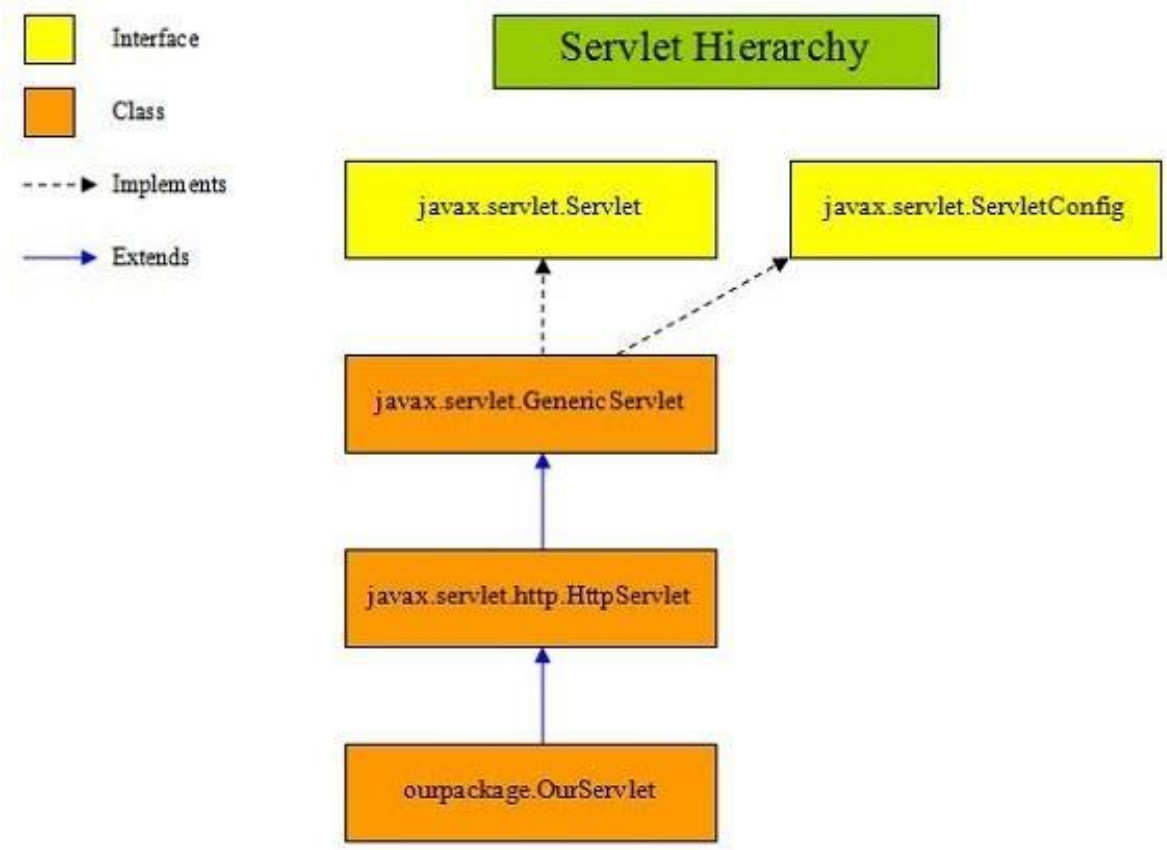
1. By implementing Servlet interface
2. By inheriting GenericServlet class
3. By inheriting HttpServlet class

Before studying each of them in brief, it is important to see the relationship between all the 3 interfaces and classes.

Servlet API Hierarchy

The below diagram depicts:

1. Servlet is an interface. It has **5** abstract methods
2. GenericServlet is a class. It **implements** Servlet Interface and defines 4 of its methods
3. HttpServlet **extends** GenericServlet class. It defines the last remaining method service()
4. Our custom servlet must **extend** HttpServlet in case we need to deal with HTTP requests/responses



Now, as we have got an abstract look at the hierarchy, let's look the 3 ways of creating a servlet one by one.

Implementing Servlet Interface

Servlet Interface

Servlet is the main interface that declares methods that all servlets must implement.

Main Points

The main points about the Servlet are:

- Servlet is an interface
- Fully qualified name: javax.servlet.Servlet
- Signature: public interface Servlet {}
- Extends: Nothing
- Implemented by: GenericServlet class

Abstract Methods in Servlet Interface

1. public abstract void **init**(ServletConfig config) throws ServletException
2. public abstract void **service**(ServletRequest req, ServletResponse res) throws ServletException, IOException
3. public abstract void **destroy**()
4. public abstract ServletConfig **getServletConfig**()
5. public abstract String **getServletInfo**()

Create a Servlet

Step 1: Define a normal class that implements java.servlet.Servlet

```
class FirstServlet implements Servlet {


}
```

Step 2: Override all the abstract methods of Servlet interface

```
public class FirstServlet implements Servlet {

	ServletConfig servletConfig;

	@Override
	public void init(ServletConfig servletConfig) throws ServletException {
		this.servletConfig = servletConfig;
		System.out.println("Servlet is initialized");
	}

	@Override
	public void service(ServletRequest req, ServletResponse res) throws
    ServletException, IOException {
		System.out.println("Request is being serviced...");
	}

	@Override
	public void destroy() {
		System.out.println("Servlet is destroyed");
	}

	@Override
	public ServletConfig getServletConfig() {
		return this.servletConfig;
	}
```

```
	@Override
	public String getServletInfo() {
		return "FirstServlet";
	}


}
```

Note: Put SOPs inside the overridden methods to track the servlet lifecycle

Step 3: Map the servlet in web.xml

```
<servlet>
	<servlet-name>first</servlet-name>
	<servlet-class>com.java.servlets.FirstServlet</servlet-class>
</servlet>


<servlet-mapping>
	<servlet-name>first</servlet-name>
	<url-pattern>/first</url-pattern>
</servlet-mapping>
```

Step 4: Execute the project

Start the Server and run the project, type: **localhost:8080/02ServletCreationTechniques/first** to run the Servlet

Step 5: Track the Output

- init() will run once when the first request is made
- service() will run on every request
- destroy () will be called when we stop the server

Observations

- init (), service() and destroy() are lifecycle methods
- Lifecycle methods are called implicitly by the web container.
- Their sequence and mode of calling depends on the state of the servlet
    - When the 1st request is made, the web container initializes the servlet by calling init ()
    - Every request to the servlet triggers the service()
    - To call destroy , we have to explicitly stop the server
- getServletConfig() and getServletInfo() are non-life cycle methods

- These methods are not called implicitly & have to be called explicitly

Note: You can find the above example in Project: 02ServletCreationTechniques

Inheriting GenericServlet Class

GenericServlet Class

GenericServlet defines a generic, protocol-independent servlet. It must be used to handle any type of request

Main Points

The main points about the GenericServlet are:

- GenericServlet is a class
- Fully qualified name: javax.servlet.GenericServlet
- Signature: public abstract class GenericServlet
- Extends: java.lang.Object
- Implements: Servlet, ServletConfig, Serializable
- Sub-classes: HttpServlet

Creating the Servlet

Step 1: Create a new class and extend it from GenericServlet

```
public class SecondServlet extends GenericServlet {


}
```

Step 2: Override unimplemented service () of GenericServlet class

GenericServlet class provides implementation of 4 methods of Servlet interface, however service() still remains unimplemented.

```
public class SecondServlet extends GenericServlet {


    private static final long serialVersionUID = 1L;


        @Override
        public void service(ServletRequest req, ServletResponse res) throws
    ServletException,IOException {
                System.out.println("Request is being serviced");
        }
}
```

Step 3: Execute the project & we will notice that service is triggered whenever there is a new request

Note: You can find the above example in Project: 02ServletCreationTechniques

Inheriting HttpServlet class

HttpServlet Class

- The websites communicate with the server through HTTP protocol
- If we make a web application, we extend our Servlet from HttpServlet
- We can say that HttpServlet class is used to create http protocol specific servlet.
- HttpServlet class is abstract although all its methods are concrete. This is done to force its child class to override request specific doXXX() method

Main Points

The main points about the HttpServlet are:

- HttpServlet is a class
- Fully qualified name: java.servlet.http.HttpServlet
- Signature: public abstract class HttpServlet
- Extends: java.servlet.GenericServlet
- Implements: Servlet, ServletConfig, Serializable (**all indirectly through GenericServlet**)
- Predefined Sub-classes: No

HttpServlet class methods

A subclass of HttpServlet must override at least one method, usually one of these:

1. doGet(), for HTTP GET requests
2. doPost(), for HTTP POST requests
3. doPut(), for HTTP PUT requests
4. doDelete(), for HTTP DELETE requests
5. getServletInfo(), which the servlet uses to provide information about itself
6. service(): There's no reason to override service() as it passes HTTP requests to HttpServlet handler methods

Creating the Servlet

Step 1: Create a new class and extend it from HttpServlet

```
public class ThirdServlet extends HttpServlet {


}
```

Step 2: Override any one method that you wish to be handled

```
public class ThirdServlet extends HttpServlet {


    private static final long serialVersionUID = 1L;


        @Override
        protected void doGet(HttpServletRequest req, HttpServletResponse
    resp) throws ServletException, IOException {


                System.out.print("Request is being processed");
        }
}
```

Step 3: Execute the project & we will notice that doGet is triggered for get request

Note: You can find the above example in Project: 02ServletCreationTechniques

Serial Version UID

GenericServlet and HttpServlet classes implement **java.io.Serializable** interface as we can see in class definitions. Whenever we create a servlet by extending these classes, we need to provide a SerialVersionUid. To understand its significance, we must have familiarity with the concept of serialization and deserialization.

Serialization

Serialization is a mechanism of converting the state of an object into a byte stream.

Deserialization

Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

SerialVersionUid

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization.

If there is a mismatch in UIDs of serialized and de-serialized classes, the deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.

It must be static, final and of type long: i.e. private static final long serialVersionUID=42L;

Note: You can find the above example in Project: 02ServletCreationTechniques

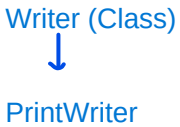
Handling Form Data & Generating Dynamic Response

Table of Contents

Generating Dynamic Response

PrintWriter

- Java PrintWriter class is the implementation of **Writer** class.
- It is used to print the **formatted** representation of objects to the **text-output stream**.
- It is present in java.io package.
- It enables us to write formatted data to an underlying Writer.
- For instance, writing int, long and other primitive data **formatted as text**, rather than as their byte values.
- It creates a character based output streams
- It is useful where we have to mix text and numbers.
- The PrintWriter class has all the same methods as the **PrintStream**
- Being a Writer subclass the PrintWriter is intended to **write text**.



Handling Data

The major source of request from a client includes:

1. Link click
2. Form submission

Link click

The following code shows the creation of a link:

Requesting a Static Page

Code Snippet

```
<a href="home.html">Home</a>
```

When "Home" link is clicked, a **get** request will be generated for the static page "home.html"

Requesting a Dynamic Page

Code Snippet

```
<a href="home.jsp">Home</a>
```

When "Home" link is clicked, a **get** request will be generated for the dynamic page "home.jsp"

Requesting a Dynamic Page with parameters

Code Snippet

```
<a href="home.jsp?greeting=good-morning&name=kaustubh">Home</a>
```

When "Home" link is clicked, the following data is sent to home.jsp in a get request:

a. "greeting" parameter containing the value "good-morning"

b. "name" parameter containing the value "kaustubh"

Requesting a Servlet

Code Snippet

```
<a href="register">Register</a>
```

When "Register" is clicked, a **get** request will be generated for the Servlet mapped by **"/register"**

Requesting a Servlet with data

Code Snippet

```
<%
```

```
    String name=”kaustubh”;
    int age = 25;
%>
<a href=“register?name=<%=name%>&age=<%=age%>”>Register</a>
```

When “Register” link is clicked, the following data is sent to the servlet with mapping “register” and following data:

a. “name” parameter containing the value “kaustubh”

b. “age” parameter containing the value “25”

Form Submission

A form contains many elements that take user input. These user inputs can be:

1. Text: name, password, mobile no, email id, gender, hobbies, address etc.
2. Date & Time: date of birth, time of an event, timestamp of a picture
3. Files: Images, PDF file, MS document file, audios and videos etc.

Let us take a look at the different user input elements and how they must be fetched in a servlet

Text Input

1. Input type text

```
<input type="text" name="full-name"/>

String fullName = request.getParameter("full-name");
```

2. Input type password

```
<input type="password" name="user-password"/>

String userPassword = request.getParameter("user-password ");
```

3. Input type number

```
<input type="number" name="mobile-number"/>
```

```
String mobileNumber = request.getParameter("mobile-number");
```

4. Input type email

```
<input type="email" name="email-id"/>

String emailId = request.getParameter("email-id");
```

5. Input type radio

```
<input type="radio" name="gender" value="male" />Male

<input type="radio" name="gender" value="female" />Female

String gender = request.getParameter("gender");
```

6. Text Area

```
<textarea name="address" rows="8" cols="40"></textarea>

String address = request.getParameter("address");
```

7. Dropdown Select

```
<select name="courses">
<option value="java">java</option>
<option value="python">python</option>
```

```
<option value="javascript">javascript</option>
</select>

String courses = request.getParameter("courses");  // In case of 1 selection for a dropdown

String[] courses = request.getParameterValues("courses"); // In case of multiple selections
```

8. Input type checkbox

```
<input type="checkbox" name="hobby" value="drawing" /> Drawing
<input type="checkbox" name="hobby" value="singing" />Singing
<input type="checkbox" name="hobby" value="dancing" />Dancing
<input type="checkbox" name="hobby" value="cooking" />Cooking

String[] hobbies = request.getParameterValues("hobby");
```

Note: Find the practical implementation in the project: **04RequestResponseWithServlets**

File Input

Before Java EE 6, applications usually have to use an external library like **Apache's Common File Upload** to handle file upload functionality. Fortunately, developers do no longer have to depend on any external library, since Java EE 6 provides **built-in file upload API**.

File Upload API in Servlet 3.0

The Servlet 3.0 API provides some new APIs for working with upload data:

1. **Annotation @MultipartConfig:** A servlet can be annotated with this annotation in order to handle multipart/form-data requests which contain file upload data. The MultipartConfig annotation has the following options:
   - fileSizeThreshold: file's size that is greater than this threshold will be directly written to disk, instead of saving in memory.
   - location: directory where file will be stored via Part.write() method.
   - maxFileSize: maximum size for a single upload file.
   - maxRequestSize: maximum size for a request.
   - All sizes are measured in bytes.

2. **Interface Part:** represents a part in a multipart/form-data request. This interface defines some methods for working with upload data (to name a few):

- getInputStream(): returns an InputStream object which can be used to read content of the part.
- getSize(): returns the size of upload data, in bytes.
- write(String filename): this is the convenience method to save upload data to file on disk. The file is created relative to the location specified in the MultipartConfig annotation.

3. **New methods** introduced in HttpServletRequest interface:

- getParts(): returns a collection of Part objects
- getPart(String name): retrieves an individual Part object with a given name.

These new APIs make our life easier, really!

The Code

The code to save upload file is as follows:

```
for (Part part : request.getParts()) {
    String fileName = extractFileName(part);
    part.write(fileName);
}
```

The above code simply iterates over all parts of the request, and save each part to disk using the write() method.

The file name is extracted in the following method:

```
private String extractFileName(Part part) {
    String contentDisp = part.getHeader("content-disposition");
    String[] items = contentDisp.split(";");
    for (String s : items) {
        if (s.trim().startsWith("filename")) {
            return s.substring(s.indexOf("=") + 2, s.length()-1);
        }
    }
    return "";
}
```

Because file name of the upload file is included in **content-disposition** header like this:

form-data; name="dataFile"; filename="PHOTO.JPG"

So the extractFileName() method gets PHOTO.JPG out of the string.

Header content-disposition

In a multipart/form-data body, the HTTP Content-Disposition general header is a header that must be used on **each subpart** of a multipart body to give information about the field it applies to. The subpart is delimited by the ***boundary*** defined in the **Content-Type** header.

Example:

```
Content-Disposition: form-data; name="fieldName"
Content-Disposition: form-data; name="file"; filename="filename.jpg"
3
4
5
6
.
.
.
```

Java Server Pages (JSP)

Table of Contents

What is JSP?

- JSP stands for Java Server Pages
- JSP is a java technology (just like Servlets) used to create web applications.
- **JSP can be said as an extension to Servlet**
- A JSP page consists of HTML tags and JSP tags.
- Using JSP we can separate view with controller

Difference between JSP and HTML

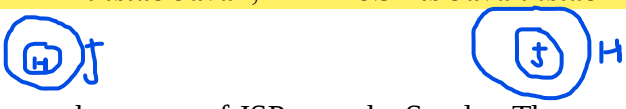
The differences between JSP and HTML are given below:

S. No.	HTML	JSP
1.	HTML is a **Client side technology**	JSP is a **Server side technology.**
2.	HTML generates **static web pages.**	JSP generates **dynamic web pages.**
3.	HTML is given by **W3C** (World Wide Web Consortium).	JSP is given by **Sun Micro System.**
4.	An HTML page can have only HTML tags and content	A JSP page can have both - HTML as well as java code.
5.	Need **HTML Interpreter** to execute this code.	Need **JSP container** to execute JSP code.

6.	It does not allow us to place custom tag or third party tag.	It allows us to place custom tag or third party tag.
7.	The file with all HTML content has **.HTML** or **.htm** extension	The file with even 1 JSP statement must have **.jsp** extension

Advantages of JSP over Servlets

"Servlet is HTML inside Java", while "***JSP is Java inside HTML"***.



There are many advantages of JSP over the Servlet. They are as follows:

1. **Extension to Servlet**

- JSP technology is the extension to Servlet technology.
- We can use all the features of the Servlet in JSP, plus
- JSP provides additional functionalities (such as implicit objects, directives, third party tag libraries, expression language) of its own that makes development faster and easier

2. **Giving flexibility to Designing**

- Both JSP and Servlet can be used to design a web page
- Servlet used out.println() to generate static web content at runtime.
- Designing a web page with "out.println()" is **exponentially cumbersome**
- JSP **eases designing** by embedding dynamic java code inside static HTML code

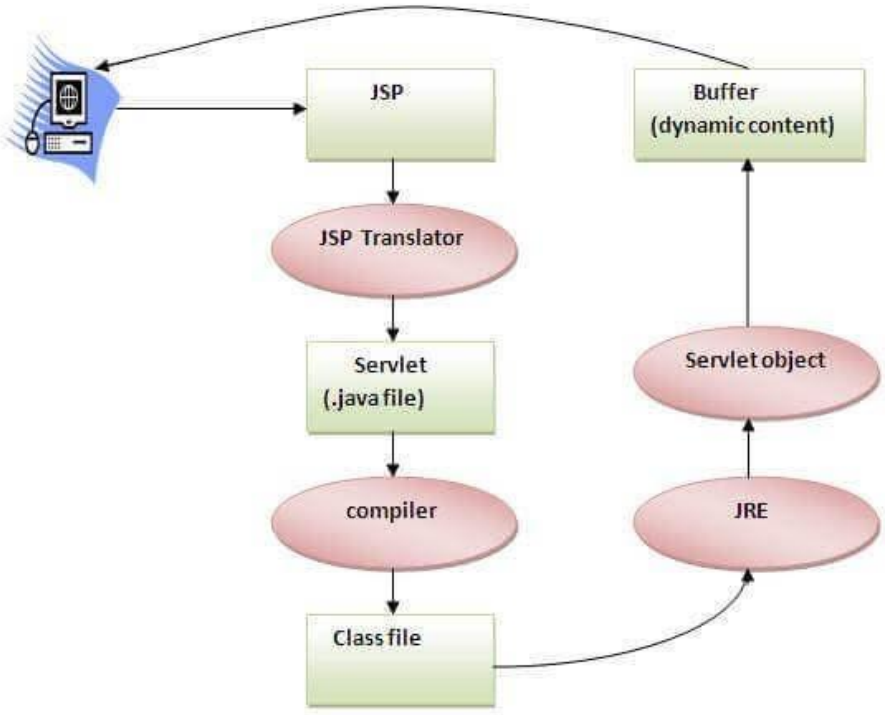
3. **Easy to maintain**

- JSP can be easily managed because we can easily separate our business logic with view logic.
- In Servlet technology, we mix our business logic with the view logic.

Lifecycle of JSP Page

The JSP pages follow these phases:

1. Translation of JSP Page
2. Compilation of JSP Page
3. Class loading (the class-loader loads class file)
4. Instantiation
5. Initialization
6. Request processing
7. Response Generation
8. Destroy



Translation

1. JSP page is translated into Servlet by the help of **JSP engine**, i.e. (XXX.jsp to XXX.java)
2. The JSP engine is a **part of the web server**

From here Servlet Life Cycle begins as discussed earlier in **Servlet Life Cycle**. Here they are for your quick revision:

Compilation

The translated Servlet page is then compiled by the **JSP Engine** into class file. (XXX.java to XXX.class)
You can find the .class files of .JSP's at the path:

[<eclipse-workspace> \ .metadata \ .plugins \ org.eclipse.wst.server.core \ tmp0 \ work \ Catalina \ localhost \ <project-context> \ org \ apache \ jsp]

Class Loading

The Servlet class is than loaded into the memory

Instantiation

Object of the Generated Servlet is instantiated

Initialization

The container invokes **_jspInit()** method

Thread Creation & Request Processing

The container invokes **_jspService()** method, we get request and response objects to get data and output the response

Destroy is called

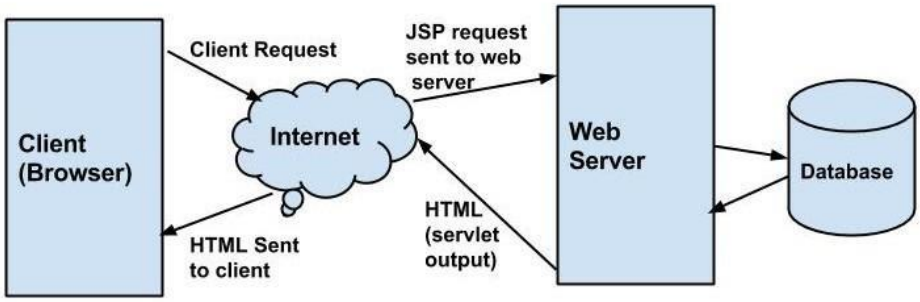
The container invokes **_jspDestroy()** method when Server is closed

Important Points

- The methods **_jspInit()**, **_jspDestroy()** and **_jspService()** of the translated Servlet corresponds to init(), destroy() and service() (doGet(), doPost()) of a regular servlet.
- Similar to servlet's service(), **_jspService()** takes two parameters, request and response, encapsulating the HTTP request and response messages.
- Similar to **PrintWriter** object, a JSP translated Servlet provides **JspWriter** object to write the response message over the network to the client.
- The HTML statements in the JSP script are written out as part of the response via out.write(...), as it-is without modification.
- The Java code in the JSP scripts are translated according to their type of tag used:
  - The code written inside JSP **Declaration tag** is placed in translated Servlet class as it **data members** and **member methods**
  - The code written inside JSP **Scriptlet** is placed inside the **_jspService()** method of the translated servlet as "it is". Scriptlets form the program logic.
  - The code written inside JSP **Expression** is placed inside an “out.print (...)” inside _jspService()

JSP Architecture

Java Server Pages are part of a **3-tier architecture**. A server (generally referred to as application or web server) hosts and supports the Java Server Pages. This server will act as a mediator between the client browser and a database. The following diagram shows the **JSP architecture**.



JSP Architecture Flow

1. The user requests a JSP page from the client browser
2. The browser sends an HTTP request to the web server.

3. The web server recognizes that the HTTP request is for a JSP page and forwards it to a **JSP engine**.
4. The JSP engine **translates the JSP page** into an equivalent Servlet source code.
5. The JSP engine **compiles** the servlet into an executable class and forwards the original request to a **servlet engine**.
6. A part of the web server called the **servlet engine** loads the Servlet class and executes it.
7. During execution, the servlet produces an output in HTML format.
8. The output is then passed on to the web server by the servlet engine inside an HTTP response.
9. The web server forwards the HTTP response to the user's browser in terms of static HTML content.
10. Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

First JSP program in Eclipse

Step 1. Start Eclipse

Step 2. Create a new "Dynamic Web Project"

Step 3. Create a JSP page inside the Web Content folder in the new project.

Step 4. Write some JSP code in the <body></body> tag.

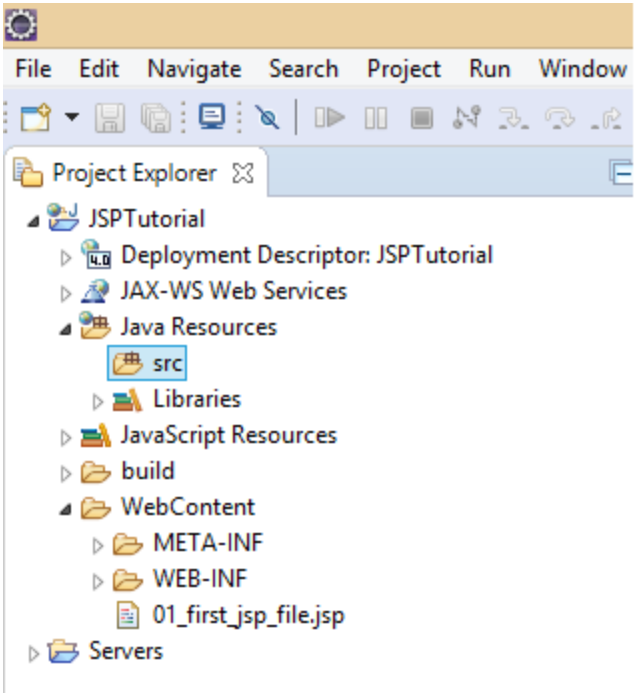
Step 5. Run the file on server.

Step 6. See the output on the browser.

Step 7. Locate the translated servlet and class files at below path:

[**<workspace-path>**\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work\Catalina\localhost**<project-name>**\org\apache\jsp]

Directory structure of JSP



The JSP API

The JSP API consists of following package:

1. javax.servlet.jsp

Package javax.servlet.jsp

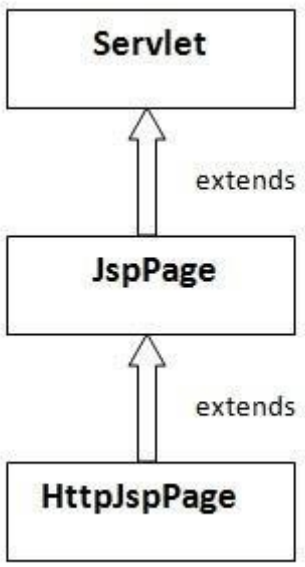
1. JspPage Interface
2. HttpJspPage class

Classes

1. JspWriter
2. PageContext
3. JspFactory
4. JspEngineInfo
5. JspException
6. JspError
7. HttpJspBase

JSP Page (I)

According to the JSP specification, all the generated servlet classes must implement the JspPage interface. It extends the Servlet interface. It provides two life cycle methods.



Methods of JspPage interface

1. **public void jspInit():** It is invoked only once during the life cycle of the JSP when JSP page is requested for the first time. It is used to perform initialization. It is same as the init() method of Servlet interface.

2. **public void jspDestroy():** It is invoked only once during the life cycle of the JSP before the JSP page is destroyed. It can be used to perform some cleanup operations.

HttpJspPage (I)

The HttpJspPage interface provides one life cycle method of JSP. It extends the JspPage interface.

Method of HttpJspPage interface

1. **public void _jspService():** It is invoked each time when request for the JSP page comes to the container. It is used to process the request. The underscore _ signifies that you cannot override this method.

JSP Scripting Elements

Table of Contents

JSP Scripting Elements

In JSP, java code is written inside the JSP page using the scripting element (also called scripting tags).

These scripting elements provide the ability to insert java code inside a JSP page.

There are three types of scripting elements:

- Declaration tag
- Scriptlet tag
- Expression tag

JSP Declaration Tag

- The JSP declaration tag is used to declare **fields** and **methods**
- The implicit objects are **not** available in a declaration tag
- The content of declaration tag is member of generated Servlet. For ex:
  - The variables are converted to **instance variables**, and
  - The methods are converted to **member functions**

Syntax

```
<%!
   field1 declaration;
   .
   .
   fieldN declarationl;
   method1 declaration;
   .
   .
   methodN declaration;
%>
```

Example

```
<html>
<body>
<%!
    int age = 50;
    public int returnAge(){
      return age;
    }
%>
<%= "Value of the variable is : " + returnAge() %>
</body>
</html>
```

JSP Scriptlet Tag

- A Scriptlet tag is used to execute java source code in JSP
- We **can** declare variables in a Scriptlet tag
- We **cannot** declare and define methods in a Scriptlet tag
- The content of the Scriptlet tag goes inside **_jspService()** method of the converted Servlet

- All the implicit objects can be used inside a Scriptlet tag

Syntax

```
<%
   java source code statement 1;
   java source code statement 2;
   .
   .
   java source code statement N;
%>
```

Example

```
<html>
<body>
<%
  String name = request.getParameter("name");


  out.print("Welcome : " + name);
%>
</body>
</html>
```

JSP Expression Tag

- The code placed within JSP expression tag is written to the output stream of the response.
- We use **out.print()** to write data, but in expression tag we can replace it with **(=)**
- It is mainly used to print the values of variable or method.
- **Only 1 statement** can be written in an expression tag
- We do not end the statement in an expression tag with semi-colon **(;)**
- We can't declare or define a variable/method inside expression tag.
- The content of the expression tag goes inside **_jspService()** method of the converted Servlet
- All the implicit objects are available inside expression tag

Syntax

```
<%= statement %>
```

Example

```
<html>
```

```
<body>
<%= "welcome to jsp" %>
</body>
</html>
```

Similarities and Differences

S. No	Property	Declaration Tag	Scriptlet Tag	Expression Tag
1.	Declare Variable	Yes	Yes	No
2.	Declare Method	Yes	No	No
3.	Servlet class members	Yes	No	No
4.	Inside _jspService()	No	Yes	Yes
5.	Implicit Objects	No	Yes	Yes
6.	Statement Terminator	Yes	Yes	No

Implicit Objects

Table of Contents

JSP Implicit Objects

In order to give programmer flexibility, JSP provides **some implicit objects.**

- These implicit objects are created during the **translation phase** of JSP to the servlet.
- These objects can be directly used in **Scriptlet** & **Expression** tag that goes in the _jspService method.
- They are created by the **container** automatically, and they are readily available for use.

JSP provides **9 implicit objects** for our disposal:

1. out
2. request
3. response
4. config
5. application
6. session
7. pageContext
8. page
9. exception

1. out

- JSP implicit object out is an instance of **javax.servlet.jsp.JspWriter**
- It is used to write the data to the buffer and send output to the client in response
- Out object allows us to access the servlet's output stream
- It is one of the most used JSP implicit object, and
- That is why we have JSP Expression to easily invoke *out.print()* method.
- Ex: **<%= new Date()%>** converts to **out.print(new Date());**

2. request

- JSP implicit object request is an instance of **javax.servlet.http.HttpServletRequest**
- It is one of the argument of _jspService() method.
- It is created by container for every request.
- We can use request object to get the request parameters, cookies, request attributes, session, header information and other details about client request.
- Ex: **<%= request.getParameter("username")%>**

3. response

- JSP implicit object response is an instance of **javax.servlet.http.HttpServletResponse**
- It is one of the arguments of _jspService() method.
- Response object will be created by the container for each request.
- We can use response object to set content type, character encoding, header information in response, adding cookies to response and redirecting the request to other resource

- Ex: **<% response.addCookie(new Cookie("Test","Value")); %>**

4. config

- JSP implicit object config is an instance of **javax.servlet.ServletConfig**
- It is created by the container for each JSP page
- It is used to get the JSP initialization parameters configured in deployment descriptor **web.xml**.
- Ex: **<%= config.getInitParameter("Course") %>**

5. application

- JSP implicit object application is instance of **javax.servlet.ServletContext**
- It is created by container **one per application**, when the application gets deployed.
- It is used to get the context information and attributes in JSP.
- We can use it to get the RequestDispatcher object in JSP to forward the request to another resource or to include the response from another resource in the JSP
- It contains a set of methods which are used to interact with the servlet container
- Ex: **<%= application.getInitParameter("User") %>**

6. session

- JSP session implicit object is instance of **javax.servlet.http.HttpSession**
- Whenever we request a JSP page, container automatically creates a session for the JSP in the service method.
- Ex: **<%=session.getId() %>**

7. pageContext

- JSP pageContext implicit object is an instance of **javax.servlet.jsp.PageContext**
- pageContext object also hold reference to other implicit object.
- It can be used to set, get or remove attributes from any of the following scopes –
    - JSP Page – Scope: PAGE_SCOPE
    - HTTP Request – Scope: REQUEST_SCOPE
    - HTTP Session – Scope: SESSION_ SCOPE
    - Application Level – Scope: APPLICATION_ SCOPE
- It has 4 main methods
    - **Object findAttribute (String attributeName):** This method searches for the specified attribute in all four levels in the following order – Page, Request, Session and Application. It returns NULL when no attribute found at any of the level.
    - **Object getAttribute (String attributeName, int scope):** It looks for an attribute in the specified scope. This method is similar to findAttribute method; the only difference is that

findAttribute looks in all the four levels in a sequential order while getAttribute looks in a specified scope.

- **void removeAttribute(String attributeName, int scope):** This method is used to remove an attribute from a given scope
- **void setAttribute(String attributeName, Object attributeValue, int scope):** It writes an attribute in a given scope.

- Ex:

In 1st page:

<% pageContext.setAttribute("Test", "Test Value"); %>

In 2nd Page

<%= pageContext.getAttribute("Test")%>

8. page

- JSP implicit object **page** is an instance of **java.lang.Object** class
- It can be thought of as an object that represents the entire current JSP page
- Or it can be thought of as a reference to the current Servlet instance (Translated from JSP).
- This object is assigned to the reference of auto generated servlet class.
- This object is an actual reference to the instance of the page. (**Object page = this**;)
- Ex: **<%= page.getClass().getName() %>**

Difference between page and pageContext

- The **page** object represents the generated servlet instance itself and is used as a scope with in one **jsp**.
- **The pageContext** is used to initialize all implicit objects for example :- **page** attributes, access to the request, response and session objects, as well as the JspWriter referenced by out.

9. exception

- JSP exception implicit object is instance of **java.lang.Throwable** class
- It is used to provide exception details in **JSP error pages.**
- We can't use this object in normal JSP pages and it's available only in JSP error pages

JSP Scopes

Table of Contents

JSP Scopes

- Every program relies heavily on **variables**.
- In Java till now we have learnt about **instance**, **static** and **local** variables.
- We also know that each variable has a **scope** that decides its accessibility.
- Similarly in JSP too we need variables to carry and manipulate **data**.
- JSP provides the **mechanism of scope** for its created objects.
- The availability of a JSP object for use from a particular place of the application is defined as the scope of that JSP object.
- Every object created in a JSP page will have a scope

Types of Scopes in JSP

Object scope in JSP is segregated into four parts and they are:

1. Page Scope
2. Request Scope
3. Session Scope
4. Application Scope

Page Scope

- Page scope means, the JSP object can be accessed only from within the same page where it was created.
- JSP implicit objects **out**, **response**, **config**, **page, pageContext** and **exception** have 'page' scope.

Request Scope

- A JSP object created using the 'request' scope can be accessed from all the pages that serve that request.
- More than one page can serve a single request.
- The JSP object will be bound to the request object.
- Implicit object request has the 'request' scope.

Session Scope

- Session scope means, the JSP object is accessible from pages that belong to the same session from where it was created.
- The JSP object that is created using the session scope is bound to the session object.
- Implicit object session has the 'session' scope.

Application Scope

- A JSP object created using the 'application' scope can be accessed from any pages across the application.
- The JSP object is bound to the application object.
- Implicit object application has the 'application' scope.

Note: We will practice these scopes while learning about **pageContext** implicit object

Parameters and Attributes

Parameters

- Parameters may come into our application
  - From a client request, or
  - May be configured through deployment descriptor (web.xml) elements or
  - Their corresponding annotations.
- When we submit a form, form values are sent as request parameters to a web application.
  - In case of a **GET** request, these parameters are exposed in the URL as name value pairs and
  - In case of **POST**, parameters are sent within the body of the request.
- Servlet init parameters and context init parameters are set through the deployment descriptor (web.xml) or their corresponding annotations.
- All parameters except application context are **read-only** from the application code.
- We have methods in the Servlet API to retrieve various parameters.

Servlet API methods to access parameters

S. No	Object	Method
1	ServletRequest	String[] getParameterValues (String paramName)
2	ServletRequest	String getParameter (String parmName)
3	ServletRequest	Enumeration<String> getParameterNames ()
4	ServletRequest	Map <String, String[]>getParameterMap ()
5	ServletConfig	Enumeration<String> getInitParameterNames ()
6	ServletConfig	String getInitParameter (String paramName)
7	ServletContext	Enumeration<String> getInitParameterNames ()
8	ServletContext	String getInitParameter (String paramName)

Attributes

- **Attributes** are objects that are attached to various scopes and can be modified, retrieved or removed.
- They can be read, created, updated and deleted by the web container as well as our application code.
- When an object is added to an attribute in any scope, it is called **binding** as the object is bound into a scoped attribute with a given name.
- We have methods in the Servlet API to **add**, **modify**, **retrieve** and **remove** attributes.

Servlet API methods to manipulate attributes

No	**ethod**
	public void setAttribute(String name, Object value)
	public Object getAttribute(String name)
	public Enumeration<String> getAttributeNames()
	public void removeAttribute(String name)

Differences between Parameters and Attributes

1. Parameters are **read-only**; attributes are **read/write** objects.
2. Parameters are **String objects**, attributes can be objects of **any type**.

JSP Page Directives

Table of Contents

JSP Directives

What are JSP Directives?

- JSP directives are the elements of a JSP source code that guide the web container on how to translate the JSP page into its respective servlet.
- They provide global information about an entire JSP page

Syntax of a Directive:

- In JSP, directives are described in <%@ %> tags
- Directives can have many space separated attributes as key-value pairs.

Syntax 1: Directive with a single attribute

```
<%@ directive attribute="" %>
```

Syntax 2: Directive with multiple attributes

```
<%@ directive attribute1 = " " attribute2 = " " attribute3 = " " %>
```

Types of Directives

There are three types of directives:

1. Page directive
2. Include directive
3. Taglib directive

Each one of them is described in detail below with examples.

Page Directive

- It provides attributes that get applied to entire JSP page.
- It defines page dependent attributes, such as imported classes, scripting language, error page etc.
- It is used to provide instructions to the container for current JSP page.

Syntax of Page directive

```
<%@page attribute1 = “” attribute2 = “”%>
```

Attributes of Page Directive

Following is the list of attributes associated with page directive:

1. language
2. contentType
3. pageEncoding
4. info
5. import
6. errorPage
7. isErrorPage
8. isELIgonored
9. session

More details about each attribute are given below

1. Attribute: language

It specifies the scripting language (underlying language) being used in the page.

Syntax:

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

Here in our case value will be **java** as it is the underlying programming language. The code in the tags would be compiled using java compiler.

2. Attribute: contentType

You can use contentType to set

1. The character encoding (charset) of the page source (during translation)
2. The character encoding (charset) of the response (during runtime)

Syntax:

```
<%@ page contentType="charset=character_set" %>
```

- The default value is "text/html; charset=ISO-8859-1"

3. Attribute: pageEncoding

- You can use pageEncoding to **set** the **character encoding** of the **page source**.
- Its main purpose is to specify a different page source character encoding than that of the response.
- The default pageEncoding is specified as "ISO-8859-1".

Syntax:

```
<%@ page pageEncoding = "character_set" %>
```

3. Attribute: info

It provides a description to a JSP page which can be accessed by getServletInfo() method.

Syntax:

```
<%@ page info="Directive Training JSP" %>
```

This attribute is used to set the servlet description.

Example:

```

```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1 "pageEncoding="value"
info="Directive Training JSP" %>
```

4. Attribute: import

- This attribute is most used attribute in page directive attributes.
- It is used to tell the container to import other java classes, interfaces, enums, etc. in the code
- It is similar to import statements in java.

Syntax:

```
<%@ page import="value1, value2, value3" %>
```

Here value indicates the classes which have to be imported.

All classes of the following **four** packages are imported by default into JSP pages:

1. java.lang
2. javax.servlet
3. javax.servlet.http
4. javax.servlet.jsp

No import statement is required in JSP files for classes in the above packages.

Example:

```
<%@ page import="java.util.Random" %>
<%@ page import="java.util.Random, java.util.ArrayList, java.io.*" %>
```

5. Attribute: errorPage

This attribute is used to set the error page for the JSP page if JSP throws an exception and then it redirects to the exception page.

Syntax:

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

Here value represents the error JSP page value

Example:

```
<%@ page errorPage = "errorHandler.jsp" %>
```

In the above code, to handle exceptions we have errroHandler.jsp

6. Attribute: isErrorPage

- It indicates that JSP Page that has an errorPage will be checked in another JSP page
- Any JSP file declared with "isErrorPage" attribute is then capable to receive exceptions from other JSP pages which have error pages.
- Exceptions are available to these pages only.
- The default value is false

Syntax:

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

Here in this case session attribute can be set to true or false

7. Attribute: isELIgnored

- EL stands for **Expression Language**
- IsELIgnored is a flag attribute where we have to decide whether to ignore EL tags or not.
- Its datatype is java enum, and the default value is false hence EL is **enabled by default**.
- We can activate Expression language in the page by setting it to false
- We can de-activate Expression language in the page by setting it to true

Syntax:

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

Here, true/false represents the value of EL whether it should be ignored or not.

8. Attribute: session

- Every time a JSP is requested, JSP creates an HttpSession object to maintain state
- This session data is accessible in the JSP as the implicit **session** object.
- In JSPs, sessions are enabled by default. By default <%@ page session="**true**" %>
- Sometimes when we don't need a session to be created in JSP, we can set this attribute to **false**.
- When it is set to false, it indicates to the compiler to **not create** the session by default.
- Disabling the session in some pages will **improve the performance** of your JSP container.
- Session object **uses** the server **resources**.
- Each session object uses up a small amount of system resources as it is stored on the server side.
- This also **increases** the **traffic** as the session ID is sent from server to client.
- Client also sends the same session ID along with each request.

- If some of the JSP pages on your web site are getting millions of hits from internet browser and there is no need to identify the user, it's **better to disable** the session in that JSP page.
- No session object will be created in translated servlet

Syntax:

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

Here in this case session attribute can be set to true or false

Note: Learn more about JSP tuning here and extends keyword here

Include Directive

- JSP "include directive" is used to include one file to the another file
- This included file can be HTML, JSP, text files, etc.
- It is also useful in creating templates with the user views
- It helps to break the pages into header, footer and sidebar actions.
- It includes file during translation phase

Syntax

```
<%@ include file = "<filename>" %>
```

Here <filename> represents the name of the file to be included

Example

File to be included: header.jsp

```
<h1>This is header</h1>
```

File where header.jsp is to be included: index.jsp

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
```

```
<%@include file=”header.jsp” %>

<h1>This is a Heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

JSP Taglib Directive & JSTL

Table of Contents

JSP Taglib Directive

The Taglib directive is used to define the tag libraries that the current JSP page uses. A JSP page might include several tag libraries. One such popular tag library is **JSTL**, i.e. JSP Standard Tag Library

Syntax:

```
<%@ taglib prefix = "prefixOfTag" uri = “uriOfTagLibrary” %>
```

Example:

Defining prefix as c when using core part of JSTL

<%@ taglib prefix = "c" uri="http://java.sun.com/jsp/jstl/core" %>

Using JSTL in code

```
<c:set var = "salary" value = "${2000}"/>
<p>My salary is:  <c:out value = "${salary}"/><p>
```

Prefix

- The prefix is used to distinguish the custom tag from other library custom tags used in a JSP page.
- Prefix is prepended to the custom tag name. Every custom tag must have a prefix.
- In the example See how prefix: **"c"** is pre-pended to the custom tag name.
- If there were 2 libraries used in the same page, then this prefix helps the programmer to distinguish between the different tag libraries.
- Hence for each library used, the prefix must be unique

URI

- Every tag library has a **Tag Library Descriptor** (TLD).
- It is an **XML document** that contains information about the library as a whole and about each tag contained in the library.
- The extension of a TLD file is **.tld**
- When we want to use any tag library (here JSTL) in our JSP page, we have to provide the URI of this tag library in our "**uri**" attribute.
- This URI is a **unique** identifier defined in the JSTL Tag Library Descriptor (TLD)
- Following is a list of Tag Libraries and their URI:
    1. Core Tags - URI → http://java.sun.com/jsp/jstl/core
    2. Formatting Tags - URI → http://java.sun.com/jsp/jstl/fmt
    3. SQL Tags - URI → http://java.sun.com/jsp/jstl/sql
    4. XML Tags - URI → http://java.sun.com/jsp/jstl/xml
    5. JSTL Functions - URI → http://java.sun.com/jsp/jstl/functions

JSTL

- JSTL stands for JSP Standard Tag Library

- The JSP Standard Tag Library (JSTL) is a collection of predefined tags to simplify the JSP development

Advantage of JSTL

1. **Fast Development:** JSTL provides many tags that simplify the JSP.
2. **Code Reusability:** We can use the JSTL tags on various pages.
3. **No need to use Scriptlet tag:** It avoids the use of Scriptlet tag.

JSTL Tag Libraries

JSTL mainly provides five types of tags:

1. Core Tags

The JSTL core tags provide variable support, URL management, flow control, etc. The URL for the core tag is **http://java.sun.com/jsp/jstl/core**. The prefix of core tag is **c**.

2. Function Tags

The functions tags provide support for string manipulation and string length. The URL for the functions tags is **http://java.sun.com/jsp/jstl/functions** and prefix is **fn**.

3. Formatting Tags

The Formatting tags provide support for message formatting, number and date formatting, etc. The URL for the Formatting tags is **http://java.sun.com/jsp/jstl/fmt** and prefix is **fmt**.

4. XML Tags

The XML tags provide flow control, transformation, etc. The URL for the XML tags is **http://java.sun.com/jsp/jstl/xml** and prefix is **x**.

5. SQL Tags

The JSTL SQL tags provide SQL support. The URL for the SQL tags is **http://java.sun.com/jsp/jstl/sql** and prefix is **sql**.

Configure JSTL jar file

1. Go to: http://tomcat.apache.org/download-taglibs.cgi
2. Download:

a. Impl: taglibs-standard-impl-1.2.5.jar

b. Spec: taglibs-standard-spec-1.2.5.jar

3. Put the .jar files in **WEB-INF/lib** folder
4. Add these to build Path: Right click on .jar file and add it to build path

JSTL Core Tags

The core tags in JSTL defines core functionalities in java such as variable assignment, conditional statements, loop statements, printing statements etc.

Include the library at the top of the jsp pages with the syntax:

```
                Tag prefix      Tag Location
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Below is the list **JSTL core tags** with their explanations

S. No.	Tags	Description
1.	<c:out>	It is used for displaying the content on client after escaping XML and HTML markup tags. Main attributes are default and escapeXML.
2.	<c:set>	This tag is useful for setting up a variable value in a specified scope. It basically evaluates an expression and sets the result in given variable.
3.	<c:remove>	It is used for removing an attribute from a specified scope or from all scopes (page, request, session and application). By default removes from all.
4.	<c:if>	This JSTL core tag is used for testing conditions. There are two other optional attributes for this tag, which are var and scope, test is mandatory.
5.	<c:choose>	It's like switch statement in Java.
6.	<c:when>	It's like case statement in Java.
7.	<c:otherwise>	It works like default attribute in switch-case statements.
8.	<c:catch>	This tag is used in exception handling. In this post we have discussed exception handling using <c:catch> core tag.
9.	<c:import>	This JSTL core tag is used for importing the content from another file/page to the current JSP page. Attributes – var, URL and scope.
10.	<c:forEach>	This tag in JSTL is used for executing the same set of statements for a finite number of times.
11.	<c:forTokens>	It is used for iteration but it only works with delimiter.
12.	<c:param>	This JSTL tag is mostly used with <c:url> and <c:redirect> tags. It adds parameter and their values to the output of these tags
13.	<c:url>	It is used for url formatting or url encoding. It converts a relative url into a application context's url. Optional attributes var, context and scope.
14.	<c:redirect>	It is used for redirecting the current page to another URL, provide the relative address in the URL attribute of this tag and the page will be redirected to the url.

JSTL Functions

The function tags in JSTL defines the functions in java that are used to manipulate Strings.

Include the library at the top of the jsp pages with the syntax:

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

Below is the list **JSTL functional tags** with their explanations

S. No.	Tags	Description
1.	fn:contains()	This function checks whether the given string is present in the input as sub-string. It does a case sensitive check.
2.	fn:containsIgnoreCase()	It does a case insensitive check to see whether the provided string is a sub-string of input.
3.	fn:indexOf()	It is used for finding out the start position of a string in the provided string. Function returns -1 when string is not found in the input.
4.	fn:escapeXML()	It is used for HTML/XML character escaping which means it treats html/xml tags as a string. Similar to the escapeXml attribute of <c:out> tag.
5.	fn:join()	It concatenates the strings with a given separator and returns the output string.
6.	fn:split()	It splits a given string into an array of substrings.
7.	fn:length()	It is used for computing the length of a string or to find out the number of elements in a collection. It returns the length of the object.
8.	fn:startsWith()	It checks the specified string is a prefix of given string.
9.	fn:endsWith()	It is used for checking the suffix of a string. It checks whether the given string ends with a particular string.
10.	fn:substring()	This JSTL function is used for getting a substring from the provided string.
11.	fn:substringAfter()	It is used for getting a substring which is present in the input string before a specified string.
12.	fn:substringBefore()	It gets a substring from input which comes after a specified string.
13.	fn:trim()	It removes spaces from beginning and end of a string and function.
14.	fn:toUpperCase()	It is just opposite of fn:toLowerCase() function. It converts input string to a uppercase string.

15.	fn:toLowerCase()	This function is used for converting an input string to a lower case string.
16.	fn:replace()	It searches for a string in the input and replace it with the provided string. It does case sensitive processing.

JSTL Formatting

The formatting tags provide support for message formatting, number and date formatting etc.

Include the library at the top of the jsp pages with the syntax:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Below is the list **JSTL functional tags** with their explanations

S. No.	**Tags**	**Description**
1.	fmt:parseNumber	It is used to Parses the string representation of a currency, percentage or number.
2.	fmt:timeZone	It specifies a parsing action nested in its body or the time zone for any time formatting.
3.	fmt:formatNumber	It is used to format the numerical value with specific format or precision.
4.	fmt:parseDate	It parses the string representation of a time and date.
5.	fmt:bundle	It is used for creating the ResourceBundle objects which will be used by their tag body.
6.	fmt:setTimeZone	It stores the time zone inside a time zone configuration variable.
7.	fmt:setBundle	It loads the resource bundle and stores it in a bundle configuration variable or the named scoped variable.
8.	fmt:message	It display an internationalized message.
9.	fmt:formatDate	It formats the time and/or date using the supplied pattern and styles.

Session Management in Java

Table of Contents

Session Management in Java

Why Session Management?

HTTP protocol and Web Servers are **stateless**, what it means is that for web server every request is a new request to process and they can't identify if it's coming from client that has been sending request previously.

But sometimes in web applications, we should know who the client is and process the request accordingly. For example, a shopping cart application should know who is sending the request to add an item and in which cart the item has to be added or who is sending checkout request so that it can charge the amount to correct client

What is a Session?

- **Session** is a conversional state between client & server where both are aware about each other
- A session indicates a **period of time** during which **a single user uses or visits a website**.
- A session starts when the user requests a page for the first time.
- During a session, the user can view as many pages as he wants, this browsing build his history
- The session ends when
  - The user hasn't requested any pages for a given amount of time (timeout).
  - The user has logged out of the website
- The session timeout varies, depending on server configuration – typically from 15 to 30 minutes.

Session Management Mechanism

Since HTTP and Web Server both are stateless, the only way to **maintain a session** is when some **unique information** about the session (**session id**) is passed between server and client in every request and response.

Session Management Techniques

There are several ways through which we can provide unique identifier in request and response:

1. Cookies
2. Session Management API

Let us go through each one of them one by one

Cookies

Simply put, a cookie is a small piece of data stored on the client-side which servers use when communicating with clients.
They're used to identify a client when sending a subsequent request. They can also be used for passing some data from one servlet to another.

- Cookies are small pieces of information that are sent by web server in response header.
- It gets stored in the browser cookies.
- When client make further request, it adds the cookie to the request header
- We can utilize cookies to keep track of the session.
- If the client disables the cookies, then it won't work.

Important points

- Cookie is a class
- Cookie is defined in javax.servlet.http package
- It implements Cloneable and Serializable interface This are the markup interface
- Contains only 1 constructor: Cookie (String key, String value)

Signature

```
public class Cookie
extends Object
implements Cloneable, Serializable
```

Create a Cookie

To send a cookie to the client, we need to create one and add it to the response:

```
Cookie myCookie = new Cookie("userEmail", "kaustubhchoudhary@yahoo.com");
response.addCookie(myCookie);
```

Set the Cookie Expiration Date

We can set the max age which defines how many seconds a given cookie should be valid for:

```
myCookie.setMaxAge(60*60);
```

We set a max age to one hour. After this time, the cookie cannot be used by a client (browser) when sending a request and it also should be removed from the browser cache.

Advantages of cookies

1. The cookies are simple to use & implement.
2. They do not require any server resources.
3. They are stored on the user's computer, so no extra burden on the server & they can lighten the load on the server's memory.
4. They are light in size, so they occupy less memory and you do not need to send back the data to the server.
5. You can configure the cookies to expire when the browser session ends (session cookies) or
6. They can exist for a specified length of time on the client's computer (persistent cookies) and
7. One of the most advantages of the cookies is their persistence , When the cookie is set on the client's browser, it can persist for days, months or years, this makes it easy to save user preferences & visit information.
8. The cookies are stored on the client's hard disk, so if the server crashes, the cookies are still available.
9. The cookies do not only remember which websites you have been to, they remember the information about forms, and they can fill out the address forms quick & efficient.
10. Most online shopping websites allow the cookies for the address & email information but they make you fill out your credit card information each time.
11. Many companies collect the data from the cookies to run the marketing campaigns aimed at a very specific market segment including the product group, geo-location, search term & the demographics.
12. You can manage your cookies easily , if you know how, Most browsers make it easy for you to clear your browsing history , Just go to the tools , clear the history and select the cookies, the cookies are stored on your hard drive in the text file under cookie.txt , You can view or edit & delete them .

13. The cookies make browsing the Internet faster & easier, the cookies allow the website to know who you are, they can tailor your browsing experience based on the previous visits, certain websites customize site information based on your location (city) & you do not have to enter the same information every time you visit the site.

But

- Although the cookies make browsing the Internet a bit easier, they are seen by many as an invasion of privacy, since most websites will not allow their site to be accessed unless cookies are enabled, so the **browsers are set to accept the cookies by default**.
- So, the cookies are being stored "invisibly" on your hard drive every time you browse the internet, since your IP address is collected, your browsing history and online activities become public knowledge.
- The browsers such as Mozilla Firefox and Internet Explorer have the options to clear the cache and delete the cookies either manually or automatically when you exit the browser.

Disadvantages of cookies

- The cookies are not secure as they are stored in a clear text & no sensitive information should be stored in cookies, they may pose to a possible security risk because anyone can open & tamper with the cookies.
- You can manually encrypt & decrypt the cookies, but it requires extra coding, you can affect the application performance because of the time that is required for encryption & decryption.
- The user has the option of disabling the cookies on his computer from the browser's setting in response to the security or the privacy worries which will cause the problem for the web applications that require them and the cookies will not work if the security level is set to high in the browser.
- The cookies can't store complex information as they are limited to simple string information , Many limitations exist on the size of the cookie text , The individual cookie can contain a very limited amount of information (not more than 4 kb)
- A lot of security holes have been found in different browsers, Some of these holes are very dangerous that they allow malicious webmasters to gain access to the users' email, different passwords & credit card information.

Technical Disadvantages

- We cannot store space separated text in cookies
- We cannot use cookies in different browsers

HttpSession

JEE provides us with HttpSession API to manage session across different clients and servers in a standard way. This way we solve the disadvantages presented by above methods, some of which are:

a. Most of the times we don't want to only track the session; we have to store some data into the session that we can use in future requests. This will require a lot of effort if we try to implement this.
b. All the above methods are not complete in themselves; all of them won't work in a particular scenario.

So we need a solution that can utilize these methods of session tracking to provide session management in all cases.

How Http Session works?

1. On client's **first request**, the Web Container generates a **unique session** and gives it back to the client with response. This is a temporary session created by web container.
2. The client sends back the session id with each request making it easier for the web container to identify where the request is coming from.
3. The Web Container uses this session, finds the matching session and associates the session with the request

- HttpSession object is used to store entire session with a specific client.
- We can store, retrieve and remove attribute from HttpSession object.
- Any servlet can have access to HttpSession object throughout the getSession() method of the HttpServletRequest object. If we try to print httpSession Object, it will give the output as:

org.apache.catalina.session.StandardSessionFacade@1bb6a9ba

- StandardSessionFacade is the class that implements HttpSession Interface
- It is present in: org.apache.catalina.session package

Signature

public class StandardSessionFacade

extends java.lang.Object

implements HttpSession

Important Points

- HttpSession is an interface
- It is defined in **javax.servlet.http** package
- It is implemented by StandardSessionFacade class in org.apache.catalina.session package
- An object of HttpSession can be used to perform two tasks:
  - Bind objects
  - View & manipulate session information, such as session identifier, creation/last accessed time.
- The main methods in HttpSession interface are:

  - public long getCreationTime();
  - public long getLastAccessedTime();
  - public ServletContext getServletContext();
  - public void setMaxInactiveInterval(int interval);
  - public int getMaxInactiveInterval();
  - public Object getAttribute(String name);

- public Enumeration<String> getAttributeNames();
- public void setAttribute(String name, Object value);
- public void removeAttribute(String name);
- public void invalidate();
- public boolean isNew();

Preventing Caching of JSP Pages

- A browser can cache web pages so that it doesn't have to get them from the server every time the user asks for them.
- Proxy servers can also cache pages that are frequently requested by all users going through the proxy.
- Caching helps cut down the network traffic and server load, and provides the user with faster responses.
- But caching can also cause problems in a web application in which you really want the user to see the latest version of a dynamically generated page.

Solution

Both browsers and proxy servers can be told not to cache a page by setting response headers.

We can use a Scriptlet tag like this in your JSP pages to set these headers:

```
<%
  response.addHeader("Pragma", "no-cache");
  response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");
%>
```

Introduction to Maven

Table of Contents

Why do we need Maven?

To understand Maven, let us first make ourselves familiar with some terminologies of a Java Project

Executing a Java Project

We have developed many java applications until now such as:

- Simple Java Application containing a single .java file
- A bit more complex Java Program containing multiple .java files
- Java Project that contains not only .java files but also html, css, js, jsp, .jpeg, mp4, xml and jar files

Whatever be the size of the application/project, we have to execute the application.

So, how did we do it?

For single java file applications

When we were developing single .java files application, we did it by

- Compiling the source code : javac MyFirstProgram.java
- Executing the generated .class file: java MyFirstProgram 12 45
- And then we get the result

For multiple java files applications

- We have to compile all the .java file that the main program needed
- We can only then run the .class file that contained main method

For multiple types of files Java Project

If we do it manually

This task will be exhausting and virtually impossible as it will include

- Compiling all the .java files that does not depend upon any other .java file
- After the above step we will have to compile .java files that depend upon other smaller classes
- Converting the other resources and attaching them to the java application
- Including the third party .jar files such as connector.jar, servlet-api.jar, jstl.jar, hibernate.jar in the project
- Attach xml files and resources with the executable code

If we used IDE (Eclipse)

- Click on the green arrow and the IDE will take care of compiling all the .java files into .class files, attaching the resources, parsing xml files, generate compressed .jar, .war files etc.
- In the same flow after compilation, the IDE will execute the code

Build System

So, as you can see from the above discussion that writing code and executing code requires a lot of effort. This process of bringing all the resources together to a state where we can successfully execute our Java Application is known as **Building the Project**.

And the system that we follow to build the project is known as a Build System.

In other words we can say that:

Software Build is an activity to translate the human-readable source code into the efficient executable program.

There are a lot of build systems, softwares, plugins available in the market, famous amongst which are:

1. Jenkins
2. Apache Ant
3. Gradle
4. Maven
5. Some IDEs use their custom build process
6. Some IDEs use third party plugins to build their projects

So, we can say that building a software project typically includes one or more of these activities:

- Generating source code (if auto-generated code is used in the project).
- Generating documentation from the source code.
- Compiling source code.
- Packaging compiled code into JAR files or ZIP files.
- Installing the packaged code on a server, in a repository or somewhere else.

Dependencies

Suppose we write a simple java application which contains two classes: Box and BoxDemo. The main method of BoxDemo class contains the code to instantiate Box class. Now in order to execute the code we will require .class files of both classes: Box.class and BoxDemo.class. The programs will not execute if Box.class is not available.

That is what we call a dependency. Our program depends upon Box.class to execute.

Now take this knowledge into our JEE projects.

- We use Tomcat server that provides servlet-api.jar package that contains .class files for implementing Servlet logic
- We use connector.jar files to connect our Java application to database by using interfaces such as Connection, PreparedStatement, ResultSet etc

All these .jar files that we use are known as dependencies that have to be included in our Project

The problem without Maven

The good thing is that our IDE takes care of building the whole Java Project by taking proper care of all the dependencies in just a click. Still we need to search, paste and include the jars in our project manually. To automate this redundant task, Maven is used

Why Study Maven?

In the modern world, we need faster, cost effective and efficient development cycles. So, to achieve this there had been continuous efforts in the field of software development to

- Ease the life of developers,
- Reduce the time of development,
- To reduce the cost of development
- Reduce the bugs

Maven is the result of one such effort

We know that a Java Application needs a lot of dependencies, so it becomes hard for the java programmer to maintain them at a certain point of time. Maven simplifies this process

What is Maven?

Building a software project typically consists of such tasks as downloading dependencies, putting additional jars on a class-path, compiling source code into binary code, running tests, packaging compiled code into deployable artifacts such as JAR, WAR, and ZIP files, and deploying these artifacts to an application server or repository.
Apache Maven automates these tasks, minimizing the risk of humans making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.

Maven Features

The key features of Maven are:

- **Simple project setup that follows best practices:** Maven tries to avoid as much configuration as possible, by supplying project templates (named *archetypes*)
- **Dependency management:** It includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies)

- **Isolation between project dependencies and plugins:** with Maven, project dependencies are retrieved from the *dependency repositories* while any plugin's dependencies are retrieved from the *plugin repositories*, resulting in fewer conflicts when plugins start to download additional dependencies
- **Central repository system:** project dependencies can be loaded from the local file system or public repositories, such as Maven Central

Installing Maven

To install Maven on your own system (computer), go to the Maven download page and follow the instructions there. In summary, what you need to do is:

1. Download Maven from: https://maven.apache.org/download.cgi
2. Set the JAVA_HOME environment variable to point to a valid Java SDK
3. Example: (JAVA_HOME = C:\Program Files\Java\jdk-11.0.2\).
4. Download and unzip Maven.
5. Set the M2_HOME environment variable to point to the directory you unzipped

Example: (**M2_HOME** = C:\apache-maven-3.6.3)

6. Set the MAVEN_HOME environment variable to point to the directory you unzipped

Example: (**MAVEN_HOME** = C:\apache-maven-3.6.3)

7. Set the M2 environment variable to point to:

Example: (**M2** = M2_HOME/bin)

8. Add M2 to the PATH environment variable (%M2% on Windows)
9. Open a command prompt and type '**mvn -version**' (without quotes) and press enter.
10. After typing in the mvn -version command you should be able to see Maven execute, and the version number of Maven written out to the command prompt

Maven Overview - Core Concepts

Project Object Model (pom.xml)

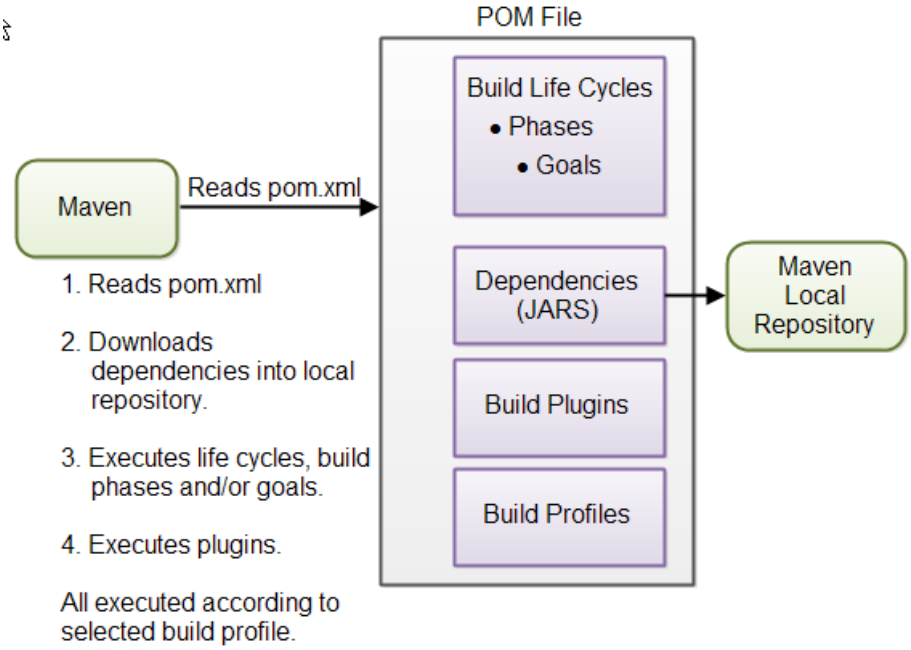
Maven is centered on the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

Let's look at the basic structure of a typical *POM* file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.baeldung</groupId>
  <artifactId>org.baeldung</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>org.baeldung</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
```

```
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
        //...
        </plugin>
      </plugins>
    </build>
</project>
```

Here is a diagram illustrating how Maven uses the POM file, and what the POM file primarily contains:



Project Identifiers

Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artifact should be packaged:

- *groupId* – a unique base name of the company or group that created the project
- *artifactId* – a unique name of the project
- *version* – a version of the project
- *packaging* – a packaging method (e.g. *WAR*/*JAR*/*ZIP*)

The first three of these (*groupId:artifactId:version*) combine to form the unique identifier and are the mechanism by which you specify which versions of external libraries (e.g. JARs) your project will use.

Dependencies

These external libraries that a project uses are called dependencies. The dependency management feature in Maven ensures automatic download of those libraries from a central repository, so you don't have to store them locally.
This is a key feature of Maven and provides the following benefits:

- uses less storage by significantly reducing the number of downloads off remote repositories
- makes checking out a project quicker

In order to declare a dependency on an external library, you need to provide the *groupId*, *artifactId*, and the *version* of the library. Let's take a look at an example:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>4.3.5.RELEASE</version>
</dependency>
```

Repositories

A repository in Maven is used to hold build artifacts and dependencies of varying types. The default local repository is located in the *.m2/repository* folder under the home directory of the user.
If an artifact or a plug-in is available in the local repository, Maven uses it. Otherwise, it is downloaded from a central repository and stored in the local repository. The default central repository is Maven Central.
Some libraries, such as JBoss server, are not available at the central repository but are available at an alternate repository. For those libraries, you need to provide the URL to the alternate repository inside *pom.xml* file:

```
<repositories>
  <repository>
    <id>JBoss repository</id>
    <url>http://repository.jboss.org/nexus/content/groups/public/</url>
    </repository>
</repositories>
```

Note: Please note that you can use multiple repositories in your projects

Introduction to Hibernate

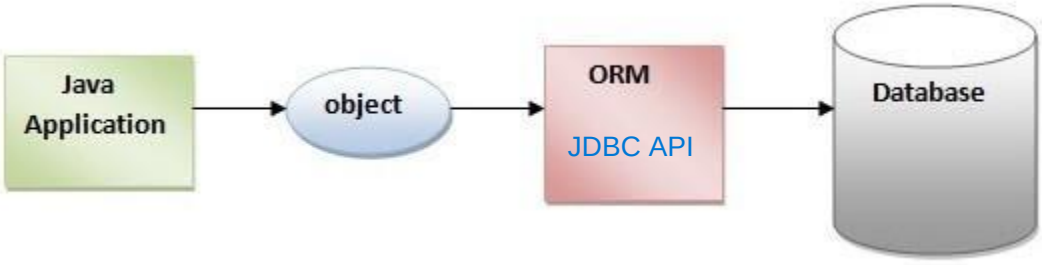
Table of Contents

What is Hibernate?

- Hibernate is a **Java framework**
- It simplifies the development of Java applications to interact with the database.
- Hibernate is an **ORM** (Object Relational Mapping) tool.
- Hibernate implements the specifications of JPA (**Java Persistence API**) for data persistence.
- Latest JPA specification is **2.2**.
- Other implementations of JPA are **iBatis** and **Toplink**
- Hibernate can be used to develop all types of application be it web, enterprise or desktop application

What is ORM?

An ORM tool simplifies data creation, data manipulation and data access. It is a programming technique that **maps an object** to **the data stored in the database**.



Note: The ORM tool **internally uses** the **JDBC API** to interact with the database.

What is JPA?

Java Persistence API (JPA) is a Java **specification** that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

Hibernate Origins

- Hibernate was developed by **Gavin King** in **2001** at **Cirrus Technologies**
- In early 2003, the Hibernate development team began Hibernate 2 releases
- **JBoss Inc.** (now part of **Red Hat**) later hired the lead Hibernate developers for further development.

Hibernate Version History

- Hibernate Core 3.0 was released in 2005
- Hibernate Core 4.0 was released in 2011
- Hibernate **Core 5.0** was released in 2018

Features of Hibernate

1. Open Source

- Hibernate is **open source**.
- It is distributed under the GNU Lesser General Public License 2.1
- Hence, it is free to download and use

2. Lightweight

A framework is called lightweight when it comes to size and transparency, the term lightweight is sometimes applied to a program, protocol, device, or anything that is **relatively simpler or faster** or **that has fewer parts** than something else.

Hibernate is lightweight because:

- Hibernate is intended for **one and the only task** **i.e.** of saving object data to our database.
- Because Hibernate is focused on just one thing, it is **relatively simple and efficient**.
- Hibernate is implemented with a set of **simple POJO**s and POJOs are **simple classes**.
- **Logic** in the POJOs is directly **executed** in the **same thread** of control as the web layer.

Hence Hibernate is simple and lightweight

3. Non-invasive

Hibernate is a **non-invasive** framework, means it won't force the programmers to extend/implement any class/interface.

4. Fast Performance

Hibernate implements various mechanisms to achieve fast performance, such as:

- **Caching** – mechanism to save number of queries to the database
- **Lazy Loading** – mechanism to make query database when the data is actually needed

5. Database Independent Query

Hibernate Query Language **(HQL)** is an **object-oriented query language**, similar to SQL, but instead of operating on tables and columns, HQL **works with persistent objects** and their properties.

- **HQL** (Hibernate Query Language) is the **object-oriented version** of SQL.
- It generates **database independent queries**. So there is no need to write database specific queries.
- HQL queries are **translated** by Hibernate **into conventional SQL queries**, which in turns perform action on database.
- Before Hibernate, if database was changed for the project, we had to change the SQL query as well to suit to changed database. It lead to the maintenance problem.

6. Automatic Table Creation

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

7. Simplifies Complex Join

Fetching data from multiple tables is easy in hibernate framework.

8. Provides Query Statistics and Database Status

Hibernate supports Query cache and provide statistics about query and database status

Why Hibernate?

- Hibernate was developed as an alternative to using **EJB2-style entity beans.**
- The original goal was to offer **better persistence capabilities** than those offered by EJB2
- It **simplified the complexities** and supplemented certain missing features.

Why use Hibernate over JDBC?

JDBC has its place, but Hibernate comes ready with an arsenal of **helpful tools and capabilities** that make connecting and transacting with the database a **much easier** prospect.

Advantages of Hibernate over JDBC

S. No.	JDBC	Hibernate
1.	In case of JDBC we need to learn SQL (Structured Query Language) along with Java to write database code.	Hibernate is set of objects, so we don't need to learn SQL language. Having only Java knowledge is sufficient.
2.	JDBC is **database dependent** i.e. one needs to write different codes for different database.	Hibernate is **database independent** and same code can work for many different databases such as MySQL, Oracle, SQL Server etc. with minor changes

3.	JDBC enables developers to create queries and update data to a relational database using the Structured Query Language (SQL).	Hibernate uses HQL which is similar to SQL but understands object-oriented concepts like inheritance, association etc.
4.	JDBC code needs to be written in a try catch block as it throws checked exception (SQLException).	Whereas Hibernate manages the exceptions itself by marking them as **unchecked**.
5.	In JDBC, the developer needs to write code to map the object model's data representation to the schema of the relational model.	Hibernate maps the object model's data to the schema of the database itself with the help of annotations.
6.	Creating associations between relations is quite hard in JDBC.	Associations like one-to-one, one-to-many, many-to-one, and many-to-many can be acquired easily with the help of annotations.

First Hibernate Application

Table of Contents

First Hibernate Application

Setup Development Environment

Download and Install

1. Java Development Kit (Current version: **11.0.2**)
2. Eclipse IDE (Current version: Eclipse IDE **2020-12 R**)
3. MySQL Server and MySQL Workbench (Current version: **8.0.23**)
4. Apache Maven (Current version: **3.6.3**)

Create Application

Step 1: Create a Maven Project

- Choose **Internal** Catalogue
- Choose **Quicktype** archetype
- Add other project details such as
    1. group id: com.hibernate.demo,
    2. artifact id: FirstHibernateProject and
    3. version: 0.0.1-SNAPSHOT (Do not change – Keep it as it is)
    4. package: com.hibernate.demo

Step 2: Add Dependencies

In order to use Java Persistent API, Hibernate Framework and Database API we have to include the certain dependencies in our project.

- Search for **Hibernate** Maven and **MySQL Connector** Maven on the internet
- Among the search results, open the link that specifies: **https://mvnrepository.com/**.
- Copy paste the dependencies in **pom.xml** file inside **<dependencies>** tag
- The project will **automatically** update its Maven Dependencies
- If there is no automatic update, right click on the project, choose **Maven>Update** manually

Step 3: Create a POJO class

Hibernate Framework maps a class-object with RDBMS. In order to do so, we must create a POJO class.

- Create a POJO class (say **Student**) in **src/main/java** folder in a package (say **com.hibernate.pojo**)
- Create 3 fields: **private int id**, **private String name** and **private int section**.
- Annotate class Student with **@Entity** and id with **@Id.**
- Be sure to check the above annotations belong to **javax.persistence** package.

Step 4: Create Hibernate Configuration File

Hibernate will take our object and use it to transact with database, as such we need to provide some **connection details** as well as some **hibernate specific configurations**

We will use **XML** to do the configurations

- Step 1: Make an xml file in src/main/java folder
- Step 2: Its name should be **hibernate.cfg.xml**
- Step 3: Write **Hibernate DTD 3.0** (from http://hibernate.org/dtd/) as its first tag

```
DOCTYPE hibernate-configuration SYSTEM
ttp://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

- DTD stands for Document Type Definition

Note:

1. Please keep in mind that DTD versions do NOT match the Hibernate ORM version.
2. Just use the **highest** DTD version that is **lower than** or **equal to** your Hibernate ORM version.

- Step 4: Put the parent tags

```
<hibernate-configuration>
ession-factory>
ession-factory>
</hibernate-configuration>
```

- Step 5: Write down the following properties inside **<session-factory>** tag

```
<property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/hibernate_demo</property>
<property name="connection.username">root</property>
<property name="connection.password">1234</property>
<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
<property name="hbm2ddl.auto">update</property>
<property name="show_sql">true</property>
<property name="format_sql">true</property>
```

- Step 6: Tell Hibernate about Entity classes

Copy the fully qualified name of the Entity class (Student class in our example) and mention it in **<mapping>** tag inside <session-factory> tag after all the <property> tags

```
<mapping class="com.hibernate.pojo.Student"/>
```

The configuration file is complete

Step 5: Create an Execution Class

Write the following code

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


public class App {
  public static void main(String[] args) {
    // Load Configuration file
    Configuration configuration = new Configuration();
    configuration.configure();
```

```
    // Create a session factory
    SessionFactory sessionFactory = configuration.buildSessionFactory();

    // Create a session
    Session session = sessionFactory.openSession();


    // Start Transaction
    Transaction transaction = session.beginTransaction();

    // Create Student object to save
    Student student = new Student(101, "Kaustubh", 5);


    // Save the student object
    session.save(student);


    // Commit the transaction
    transaction.commit();


    // Close Resources
    session.close();
    sessionFactory.close();
  }
}
```

Step 6: Execute the code

- Make sure the MySQL Server is running with the config data mentioned in config file
- Make sure there is a schema: hibernate_demo in place
- Run the application as a Java Application

Note: We will learn about the above code in later classes

Hibernate Architecture

Table of Contents

Hibernate Architecture

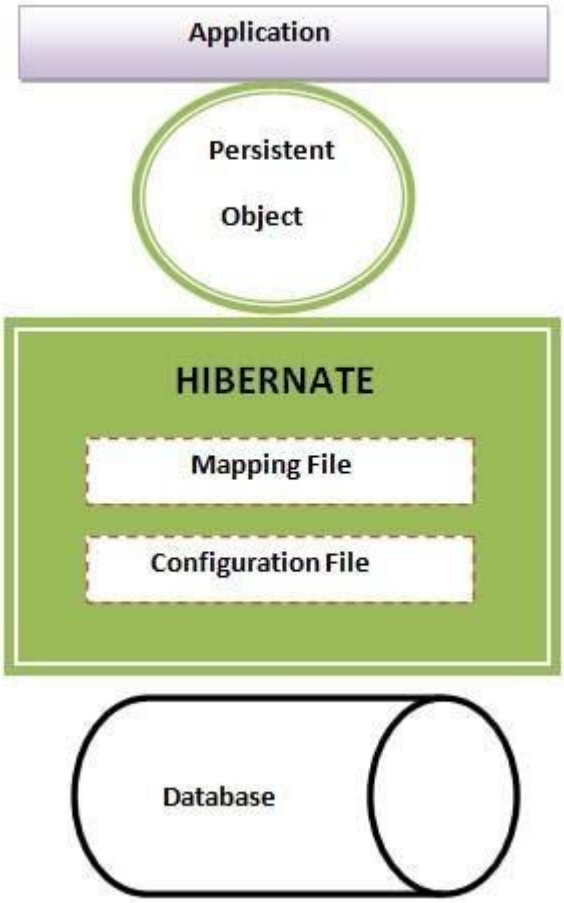
The Hibernate **architecture** includes many objects such as **persistent object**, **session factory**, **transaction factory**, **connection factory**, **session**, **transaction** etc.

The Hibernate architecture is categorized in four layers.

1. Java application layer
2. Hibernate framework layer
3. Backend API layer
4. Database layer

Let's see the diagram of hibernate architecture:

High Level Architecture



Java Application Layer: The layer where we write Java Code

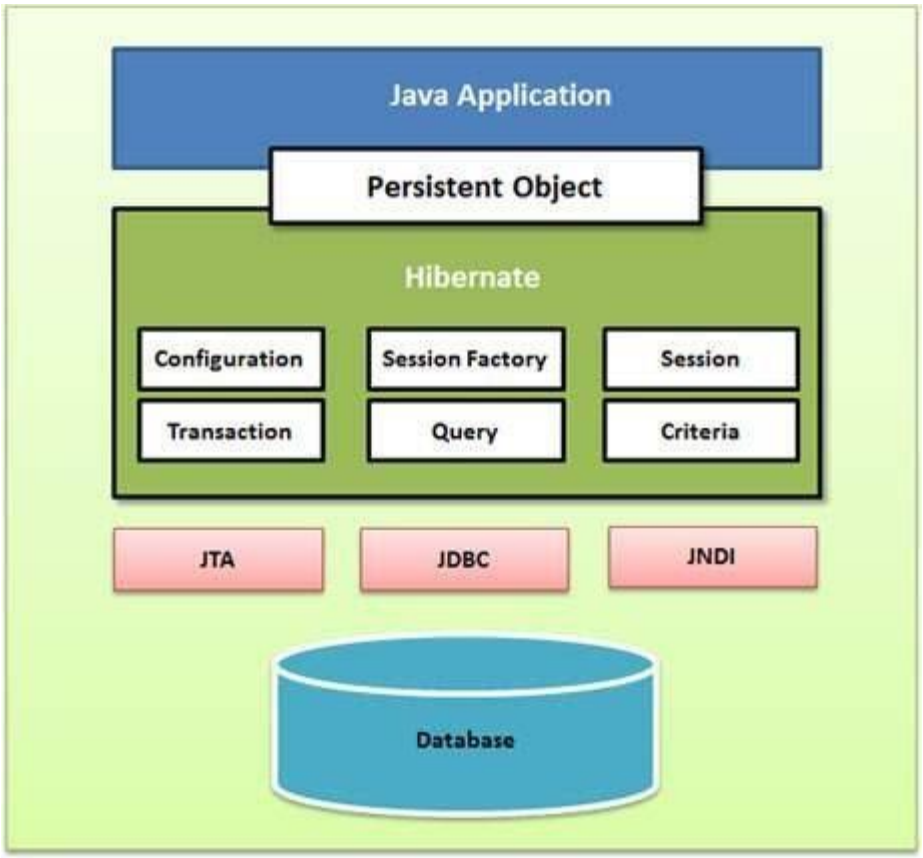
Hibernate Framework Layer: It consists the Persistent Object that is mapped with the database

Backend API Layer: It consists of the mapping file and configuration file

Database layer: The targeted database

Block Diagram

The following is the block diagram of Hibernate



Core Objects of Hibernate API

The following are the core objects of the Hibernate API:

1. Configuration
2. Session Factory
3. Session
4. Transaction
5. Query
6. Criteria

Internal APIs used by Hibernate

1. **JDBC** (Java Database Connectivity): JDBC is the standard API that Java applications use to interact with a database. Hibernate uses JDBC internally to insert a Java object into database
2. **JTA** (Java Transaction API): It allows applications to perform distributed transactions, that is, transactions that access and update data on two or more networked computer resources.
3. **JNDI** (Java Naming and Directory Interface)

Elements of Hibernate Architecture

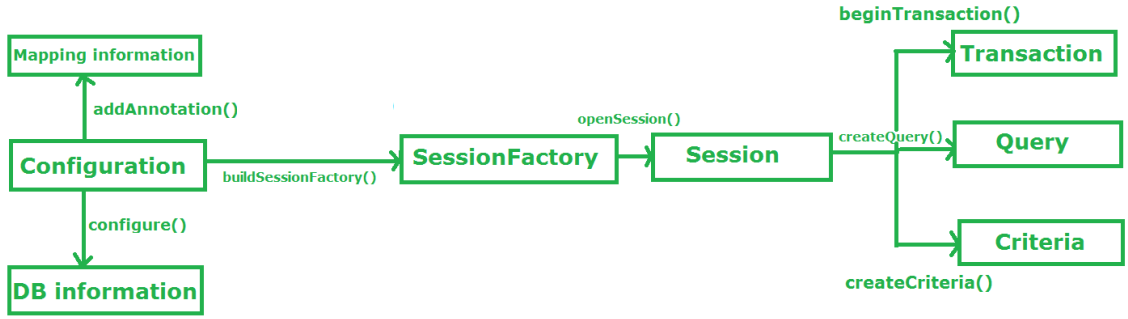


Fig: Hibernate Architecture

While creating our first hibernate application, we made use of the various classes and interfaces. Let's look at them one by one briefly.

Configuration

- Configuration is a class
- It is present in **org.hibernate.cfg** package.
- The below statement activate Hibernate Framework

```
Configuration configuration = new Configuration();
```

- The below statement **reads** both, the configuration file (hibernate.cfg.xml) & mapping files (Student entity)

```
configuration.configure();
```

- The **configure()** checks for the configuration file "**hibernate.cfg.xml**" at "**src/main/java**" location, if it can't find one it will throw a **ConfigurationException: Could not locate cfg.xml resource**
- In case our configuration file has a **different name** or is at a **different location**, we can use the **overloaded versions** of configure()
- It has got 5 overloaded methods:

No	ethod Signature (All return Configuration)	sk – Loads configurations in memory
	configure()	cates hibernate.cfg.xml in java folder
	configure(String resource)	cates the resource from specified path
	configure(URL url)	cates the resource from specified URL
	configure(File configFile)	cates the resource from specified File
	configure(org.w3c.dom.Document document)	precated

- If the resource is valid then config() **creates** a **meta-data** in **memory** and returns the meta-data to **configuration** object to **represent the config file**.

- Configuration class invokes **buildSessionFactory** to provide meta data to SessionFactory object

Significance

- It is the first object we create in a Hibernate Application
- It is created only once during application initialization.
- The configuration object provides 2 key components:
  1. **Database Connection**: This is handled through one or more configuration files supported by Hibernate: These files are **hibernate.properties** or **hibernate.cfg.xml**
  2. **Class mapping Setup**: This component creates the connection between the **Java Classes** and **database tables**

SessionFactory

- SessionFactory is an **Interface**
- It is present in **org.hibernate** package
- To get an object of SessionFactory, we call **buildSessionFactory()** on Configuration object.
- This method creates a SessionFactory using the **properties** and **mappings** in the configuration file.
- buildSessionFactory() takes JDBC information from the configuration object & **creates a JDBC Connection**.

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

- The SessionFactory is **immutable** and **thread-safe** in nature.
- So changes made to this Configuration after building the SessionFactory will not affect it.
- SessionFactory holds **second level cache** of data.
- Throws exception on invalid **configuration** or invalid **mapping information**
- The SessionFactory interface provides **openSession()** method to get the object of Session.

Immutability and Thread Safety

Most problems with concurrency occur due to sharing of objects with mutable state. Once the object is immutable, its internal state is settled on creation and cannot be changed. So many threads can access it concurrently and request for sessions

However, Session is a **non-thread safe object**; you cannot share it between threads.

Significance

- It **configures Hibernate** for the **application**
- We will need **one** SessionFactory object **per database** using a **separate configuration file**
- So, SessionFactory must be maintained in the application as a **Singleton Object**
- If our application **connects** to **more than 1 database**, we can have **multiple** config files and so, **multiple and related** Configuration and SessionFactory objects

Session

- Session is an **interface**

- It is present in **org.hibernate** package.
- Session object is created when SessionFactory object calls **openSession()** method
- It **opens the Session** with Database software through Hibernate Framework.
- The session object provides an interface between the application and data stored in the database.
- It is a **short-lived** object and wraps the **JDBC connection**.
- It holds a **first-level cache** of data.
- The Session interface is used to perform **CRUD** operations through insert, update, delete methods
- It also provides **factory methods** for Transaction, Query and Criteria.
- It is a **light-weight** object and it is **not thread-safe**.

```
Session session = sessionFactory.openSession();
```

Short Lived

- Having **long running transactions is a bad** thing as it may cause **concurrency problems**
- The session objects **should not be kept open for a long time** because they are not usually thread safe and they should be created and destroyed as needed.
- The main function of the Session is to offer, create, update, read, and delete operations for instances of mapped entity classes.

Light Weight

- The Session object is **lightweight** and designed to be instantiated each time an interaction is needed with the database.
- Persistent objects are saved and retrieved through a Session object.

Thread Safety

- Session is not Thread Safe
- Session represents a single threaded unit of work.
- The Hibernate session is a complex object that is highly stateful (it caches objects; synchronize its internal representation with the DB, etc.).
- This is just a warning that if you share a session across different threads, 2 threads could be calling methods at the same time in a way that messes up the internal state of the session and cause bugs.
- Hibernate has no way to "detect" that 2 threads are accessing it and may not throw exceptions. It's just not designed for it.
- **Program running on a single thread is simple:** everything runs sequentially, so behaviors are very predictable.

JDBC Connection v/s Session

JDBC Connection

A JDBC connection is the physical communication channel between the Database Server and the application: the TCP socket, the named pipe, the shared memory region etc.

Session

Session is a state of information exchange. A **session** encapsulates **user interaction with the database**, from the moment user was **authenticated** until the moment the user **disconnects**.

It could be thought of as an HTTP Session.

Note: A Connection may have multiple sessions

Transaction

- It is an **interface**.
- It is present in **org.hibernate** package
- The transaction object specifies an **atomic** unit of work.
- It is optional (used only when database **modifying** query is executed)
- The Transaction **interface** provides methods for transaction management.
- It is used during the queries that changes the content of the database
- It uses **commit**() to make permanent changes to database

```
Transaction transaction = session.beginTransaction();

// Create Update Delete Operation

transaction.commit();
```

Methods of Transaction interface

S. No	Method	Description
1.	void begin()	starts a new transaction.
2.	void commit()	ends the unit of work unless we are in FlushMode.NEVER.
3.	void rollback()	forces this transaction to rollback.
4.	void setTimeout(int seconds)	It sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5.	boolean isAlive()	Checks if the transaction is still alive.
6.	void registerSynchronization (Synchronization s)	Registers a user synchronization callback for this transaction.
7.	boolean wasCommited()	boolean wasCommited()
8.	boolean wasRolledBack()	boolean wasRolledBack()

Query

- Query is an **interface**

- It is present in **org.hibernate** package.
- A Query instance is obtained by calling **session.createQuery()**.
- This interface exposes some extra functionality beyond that provided by **Session.iterate()** and **Session.find()**
- A particular page of the result set may be selected by calling setMaxResults(), setFirstResult().
- Named query parameters may be used.
- Query query = session.createQuery();

Criteria

- Criteria is an **interface**
- It is present in **org.hibernate** package
- Criteria is a simplified API for retrieving entities by composing Criterion objects.
- The Session is a factory for Criteria.
- Criterion instances are usually obtained via the factory methods on Restrictions.
- Criteria criteria=session.createCriteria()

Note: In order to learn for above classes and interfaces, you should refer to JBoss docs.

Hibernate Annotations

Table of Contents

Hibernate Annotations

When we start learning and using Hibernate and JPA, the number of annotations might be **overwhelming**. But as long as we rely on the defaults, we can implement our persistence layer using only a small subset of them.

All Hibernate Annotations

@AccessType	@Generated	@NotFound
@Any	**@GeneratedValue**	**@OneToOne**
@AnyMetaDef	@GeneratorType	**@OneToMany**
@AnyMetaDefs	@GenericGenerator	@OrderBy
@AttributeAccessor	@GenericGenerators	@ParamDef
@BatchSize	**@Id**	@Parameter
@Cache	@Immutable	@Sort
@Cascade	@Index	@SortComparator
@Check	@IndexColumn	@SortNatural
@CollectionId	**@JoinColumn**	@Source
@CollectionType	@JoinColumnOrFormula	@SQLDelete
@ColumnDefault	@JoinColumnsOrFormulas	@SQLDeleteAll
@Column	@ListIndexBase	@SqlFragmentAlias
@Columns	@Loader	@SQLInsert
@ColumnTransformer	**@Lob**	@SQLUpdate
@ColumnTransformers	@ManyToAny	@Subselect
@CreationTimestamp	**@ManyToOne**	@Synchronize
@DiscriminatorFormula	**@ManyToMany**	**@Table**

@DiscriminatorOptions	@MapKeyType	@Tables
@DynamicInsert	@MetaValue	@Target
@DynamicUpdate	@NamedNativeQueries	**@Temporal**
@Entity	@NamedQueries	**@Transient**
@Embed	@NamedQuery	@Tuplizer
@Embeddable	@Nationalized	
@Fetch	@NaturalId	
@FetchProfile	@NaturalIdCache	

Key Annotations

@Entity

- The JPA specification **requires** the @Entity annotation.
- It identifies a class as an entity class.
- It is used to mark any class as an **entity**. With this we tell Hibernate that a **table has to be made** with this Entity.

Syntax

```
@Entity
public class Author { ... }
```

@Table

By default, each entity class maps a database table with the **same name** in the default schema of your database. We can customize this mapping using the *name* attribute of the *@Table* annotation.

Syntax

```
@Entity
@Table(name = "AUTHORS")
public class Author {…}
```

Key Attribute

1. Name: It enables us to change the name of the database table which our entity maps.

@Column

The @Column annotation:

1. Is an optional annotation
2. Enables us to customize the mapping between the entity **attribute** and the database **column**.
3. For example, we can set
    1. A different column name
    2. Nullable property

3. Length property

Syntax

```
@Entity
public class Book {

   @Column(name = "title", length="50", nullable="false")
   private String title;

   ...
}
```

Key Attributes

1. **Name**: The name of the column. Defaults to the property or field name. (Optional)
2. **Nullable**: Whether the database column is nullable.(Optional)
3. **length**: The column length. (Applies only if a string-valued column is used.)(Optional)

@Id

JPA and Hibernate requires us to specify **at least one primary key** attribute for each entity. We can do that by annotating an attribute with the @Id annotation.

Syntax

```
@Entity
public class Author {

   @Id
   private Long id;

   ...
}
```

Attributes: No attributes

@GeneratedValue

Hibernate will automatically generate values for that using an internal sequence. Therefore we don't have to set them manually

Syntax

```
@Entity
public class Address {
```

```
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column()
        private int addressId;


}
```

Attributes

1. strategy: specifies the key generation strategy

@Transient

It may happen sometime that we want a data member but we don't want its column to be created, in this case we use @Transient on the instance variable. This indicates to hibernate **not to save these fields**.

@Temporal

- If we use classes Date or Calendar as your Entity property then it prints a lot of information.
- Sometime all of this information is not necessary when we want to save it to database.
- @Temporal solves this problem and tells hibernate the format in which the date needs to be saved
    - @Temporal (TemporalType.TIMESTAMP): To store timestamp to your database
    - @Temporal(TemporalType.DATE): To store date
    - @Temporal(TemporalType.TIME): To store time

Note:

- Since **Hibernate 5** you don't need and **should not use @Temporal** in new code.
- It was for annotating Date and Calendar fields which are now **deprecated**.
- Instead use classes from java.time, the modern Java Date and time API
- They have their expected precision built in, and you don't use that annotation with them

@Lob

This annotation specifies that a column be created to hold a large object such as an image or a document

@Embeddable

It tells hibernate to embed an object in another object

Note: We will learn about key annotations topic wise as the course progresses

Entity Life Cycle

Table of Contents

Hibernate – Entity Life Cycle

- We know that Hibernate works with Plain Java objects **(POJO)**.
- In raw form (without hibernate specific annotations), hibernate won't be able to identify these java classes.
- But when these POJOs are properly annotated with required annotations they are said to be **mapped with hibernate**
- Only then hibernate will be able to identify them and work with them e.g. store in the database, update them, etc.

Hibernate Session

- The Session interface is the main tool used to communicate with Hibernate.
- It provides an API enabling us to **create**, **read**, **update**, and **delete** persistent objects.
- The *session* has a simple lifecycle. We open it, perform some operations, and then close it.
- When we operate on the objects during the *session*, they get attached to that *session*.
- The changes we make are detected and saved upon closing.
- After closing, Hibernate breaks the connections between the objects and the session.

Entity Lifecycle States

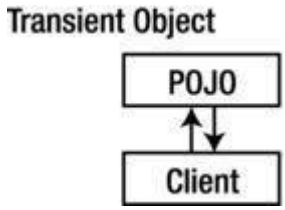
In the context of Hibernate's *Session*, instance of a class that is **mapped to Hibernate** can be in any one of the following three possible persistence states, also known as **hibernate entity lifecycle states**

1. Transient
2. Persistent
3. Detached

Transient

- An object we haven't attached to any ***session*** is in the transient state.
- Transient entities exist in heap memory as normal Java objects.
- Hibernate does not manage transient entities or persist changes done on them.

```
Session session = openSession();
UserEntity userEntity = new UserEntity("Kaustubh");
```

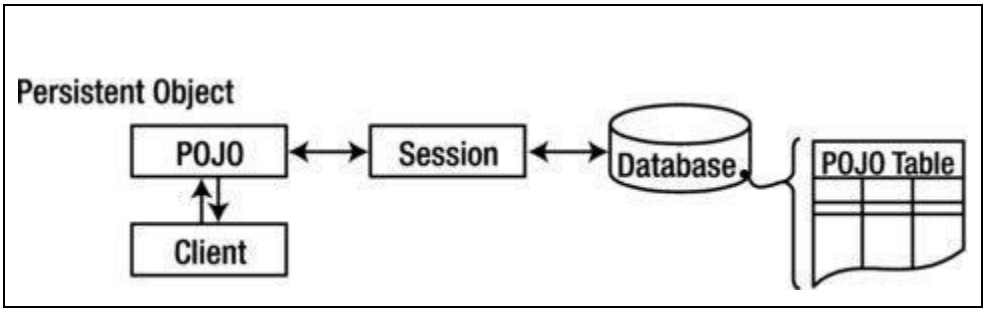


To persist the changes to a transient entity, we would have to ask the hibernate session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

Persistent

- An object that we've associated with a ***session*** is in the persistent state.
- We either **saved it** or **read it from** a persistence context, so it **represents some row** in the database.
- Persistent entities exist in the database, and Hibernate manages the persistence for persistent objects.

```
Session session = openSession();
UserEntity userEntity = new UserEntity("John");
session.persist(userEntity);
```



If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

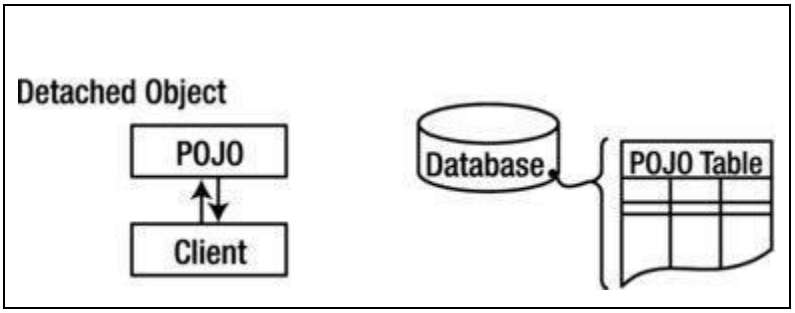
Detached

- A detached entity can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session's evict() method.
- When we close the *session*, all objects inside it become detached.

```
session.persist(userEntity);


session.evict(userEntity);
OR
session.close();
```

- Although they still represent rows in the database, they're no longer managed by any session
- That is, changes to the entity will not be reflected in the database, and vice-versa.
- This temporary separation of the entity and the database is shown in the image below.



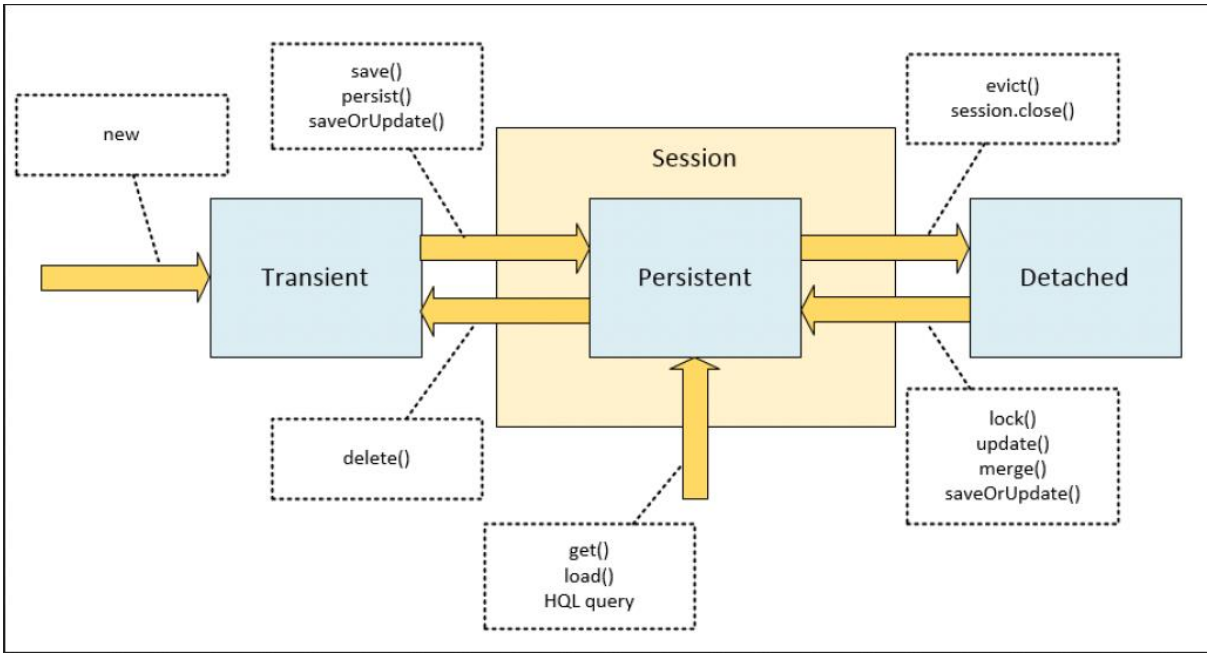
- In order to persist changes made to a detached object, the application must re-attach it to a valid Hibernate session.
- A detached instance can be associated with a new Hibernate session when your application calls one of the load(), refresh(), merge(), update(), or save() methods on the new session with a reference to the detached object.
- After the method call, the detached entity would be a persistent entity managed by the new Hibernate session.

Conclusion

1. Newly created POJO object will be in the transient state. Transient entity doesn't represent any row of the database i.e. not associated with any session object. It's plain simple java object.
2. Persistent entity represents one row of the database and always associated with some unique hibernate session. Changes to persistent objects are tracked by hibernate and are saved into database when commit call happen.
3. Detached entities are those who were once persistent in the past, and now they are no longer persistent. To persist changes done in detached objects, you must re-attach them to hibernate session.

CRUD with Hibernate

Here is a simplified state diagram with comments on *Session* methods that make the state transitions happen.



Caching Mechanism in Hibernate

Table of Contents

Cache Mechanism in Hibernate

1. Caching is a facility provided by Hibernate:

a. To helps users to get fast running web application,

b. To help framework itself to reduce number of queries made to database in a single transaction.

2. Caching is a mechanism to enhance the performance of an Application
3. Caching is implemented by Hibernate to reduce the hits in the database

How Caching works?

When our application queries the database for an object using its primary key for the first time, hibernate executes the query, fetches data, prepares an object and saves it in cache.

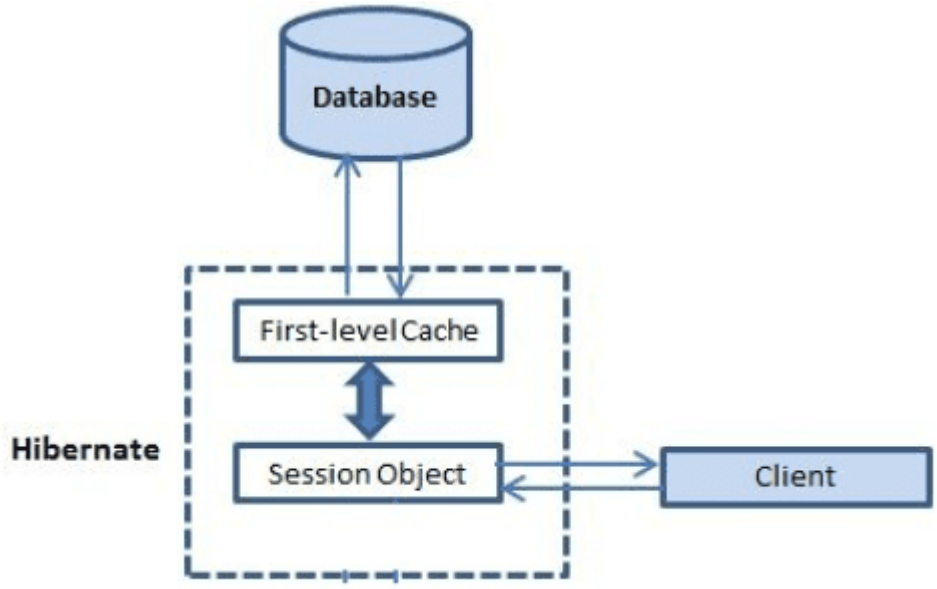
Now, if we fire a 2nd query demanding the same object, hibernate instead of querying the database fetches the stored object in the cache and returns it to the application, thereby saving 1 extra query

Types of Cache

There are 2 types of caching in Hibernate:

1. First Level cache (also known as **L1 cache**)
2. Second Level cache (also known as **L2 cache**)

First Level Cache



Lifecycle

The scope of L1 cache objects is of **session**.

1. First level cache is associated with “session” object.
2. The first level cache comes into existence as soon as a session object is created from session factory.
3. This L1 cache is not available to other session objects in the application but only to its own
4. The first level cache associated with its session object is available only till its session’s object is live.
5. Once session is closed, cached objects are gone forever.

Accessibility

1. First level cache is enabled by default and we cannot disable it, even forcefully.

2. We don’t need to do anything special to get this functionality working.

Working

1. Hibernate stores an entity in a session’s first level cache when:
    1. An object is inserted in database for the 1st time
    2. An object is updated in database
    3. An object is retrieved from database for the 1st time
2. If we query same object again with **same session object**, it will be loaded from cache and no SQL query will be executed.

Manipulation

1. The loaded entity can be removed from session using **evict (Object object)** method.

Now if we try to retrieve the same object, hibernate will make a call to database

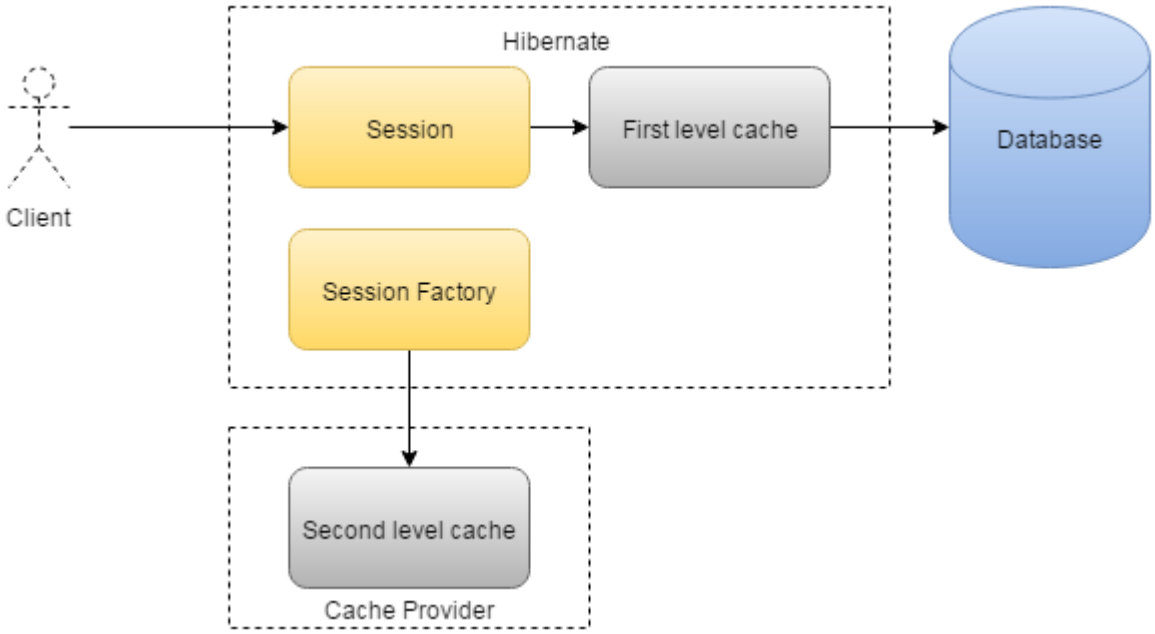
2. The whole session cache can be removed using **clear ()** method.

It will remove all the entities stored in cache.

Second Level Cache

Lifecycle

1. Second Level cache or Level 2 cache is associated with Session factory
2. It is created in session factory scope
3. It is available to be used globally in session factory scope.
4. It is available to be used in all sessions which are created using that particular session factory.
5. It also means that once session factory is closed, all cache associated with it die and cache manager also closed down.
6. It also means that if you have two instances of session factory (**normally no application does that**), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results.



How second level cache works

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. However, Second level cache cannot validate itself for modified entities, if modification has been done in the background without hibernate session APIs.

Accessibility

1. We have to manually enable second level cache
2. We will have to include some jar and do some configurations to use it
3. There are many cache providers. We will use:
    1. EH cache
    2. OS Cache
    3. Hibernate cache

How To?

Step 1: Include some dependencies in pom.xml

Step 2: Make the following changes in configuration file

Step 3: By default our entity is not cacheable; we have make our entity cacheable explicitly:

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
class Department {


}
```

Get v/s Load

Table of Contents

Fetch Data

Hibernate Session class provides two methods to access object: get () and load ()
Both looked quite similar to each other but there are subtle differences between both of them which can affect the performance of the application.

Get V/s Load

Property	**Get()**	**Load()**
Behavior	Get method of hibernate session returns null if object is not found in cache as well as on database	Load method throws ObjectNotFoundException if object is not found on cache as well as on data base but never returns null
Database Hit	get() involves database hit if an object doesn't exist in Session Cache and returns a fully initialized object which may involve several database calls	load method can return proxy in place and only initialize the object or hit the database if any method other than getId() is called on persistent or entity object. This lazy initialization can save a couple of database round-trip which results in better performance.
Proxy	Get method never returns a proxy, it either returns null or fully initialized Object	load() method may return proxy, which is the object with ID but without initializing other properties, which is lazily initialize
Performance	In comparison with load, get() is low on performance	load() is better in performance
Usage	Use if you are not sure that object exists in data base or not	Use if you are sure that object exists

Spring Introduction

Table of Contents

Framework

Why Framework?

Developing software is a complex process. It contains a lot of tasks such as – architecture, coding, designing, testing, deployment, maintenance, scaling etc. For only the coding part, programmers had to take a lot of care in syntax, declarations, garbage collection, statements, error handling, space complexity, time complexity, development cycles and more.

It could be really nice if some readymade library or system was there to simplify these tasks. This is where frameworks come into picture. These frameworks help developers to minimize these headaches.

What is a Software Framework?

Software frameworks make life easier for developers by allowing them to take control of the entire software development process, or most of it, from a single platform.

Advantages of using a software framework

1. Assists in establishing better programming practices and fitting use of design patterns
2. Code is more secure
3. Duplicate and redundant code can be avoided
4. Helps consistent development of code with fewer bugs
5. Makes it easier to work on sophisticated technologies
6. Several code segments and functionalities are pre-built and pre-tested. This makes applications more reliable
7. Testing and debugging the code is a lot easier and can be done even by developers who do not own the code
8. The time required to develop an application is reduced significantly
9. With the reduction in time and labor, softwares are cost efficient

Java Frameworks

Java is one of the oldest languages with a huge community of developers. There are many frameworks of Java that are currently being used for various tasks, such as:

1. Spring

2. Apache Struts
3. Grails
4. Hibernate
5. JSF (Java Server Faces)
6. GWT (Google Web Toolkit)
7. Blade
8. Play
9. Vaadin
10. DropWizard
11. And many more…

Spring

What is Spring?

Spring is a **Java framework** used to develop Enterprise Applications.

Or, in other words:

Spring is a Framework used to develop Enterprise applications using the best practices of J2EE.

So, it is clear from above facts that:

1. Spring is a **Software Development framework**
2. Spring is a **Java based** software development framework
3. Spring is used to develop **Enterprise grade applications**

More on Spring

Spring is **not just one framework** – it is a framework of frameworks.

What it means is:

1. It consists of 20+ modules – all targeted to solve some particular problem of an enterprise grade application
2. It can work well with other Java Frameworks as well with easy integration techniques.

What is Spring?

- Spring is a Dependency Injection Framework used to make Java Applications loosely coupled
- Spring is a powerful, lightweight, open source application development framework used for developing Java Enterprise Edition (JEE) Applications.

Why should we learn Spring Framework?

- Spring is a big framework
- It is the most popular application development framework for enterprise Java.
- Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

Features of Spring Framework

Lightweight

The Spring Framework is very lightweight with respect to its size and functionality.

1. This is due to its POJO implementation, which doesn't force it to inherit any class or implement any interfaces.
2. We don't need all of Spring to use, only part of it. For example, we can use Spring JDBC without Spring MVC.
3. It follows modular approach: Spring provides various modules for different purposes; we can just use certain according to our required module.

Dependency Injection (DI)

What is DI?

Dependency Injection is a design pattern. It is not just specific to Java, it is used in planning software architecture and therefore it can be applied in any language

1. Suppose there are 2 classes: class A and class B.
2. Now class B is a member of class A, i.e. class A has "has-a" relationship with class B.
3. Class A will be known as dependent class and,
4. Class B will be known as a dependency
5. When we inject (supply/assign) a value to B's reference in class A, it is known as injecting a dependency.
   Let us see an example: DIExample

Why worry about DI?

Tight Coupling

As we saw in the example, a change in class B forced us to change the coding of class A. This interdependency of classes is known as coupling and when one change forces us to change other classes' code, it is known as tight coupling. In tight coupling, we have to change the code in both the classes and recompile the source

Loose Coupling

When we build large applications, scalability and maintainability are important aspects to keep in mind. This is where design patterns come into picture. These design patterns help us to develop an application that is easy to maintain and scale. With the help of Spring Framework, we can develop loosely coupled applications where the change in one class does not affect the other class. In loose coupling, we only need to recompile the changed code

Inversion of Control (IOC)

Spring Framework takes control of creating objects and injecting dependencies itself dynamically on run time, i.e. it uses the Dependency Injection design pattern to loosely couple the application code. This is known as Inversion of Control (control is shifted from programmer to Spring Framework).

Spring/IOC/DI Container

Spring Framework provides us with Spring/IOC/DI container:

- To create and manage the life cycle of Java Beans
- To manage configuration of application objects.
- To achieve Dependency Injection
- Spring Container is also known as **IOC Container** or **DI container**
- With the help of either of the following ways, we can specify dependencies in a project:
    - Configuration through XML
    - Configuration through Java Class
    - Configuration through Annotations

So, what Spring does?

Suppose we have following layers & classes in our JEE application:

Layer	Classes
UI Layer	HTML/JSP Page
Business Layer	ProductController class
Service Layer	ProductService class
Data Access Layer	ProductDao, Product Model class
DataBase	Data

- All the above dependencies will be managed by Spring
- Spring will inject the object of:
    - Product object in ProductDao class
    - ProductDao object in ProductService class
    - ProductService object in ProductController class
- We will have to just maintain config data in xml or annotation

What makes Spring Framework special?

Spring is Open Source

- Spring is freely available. We can learn it, develop in it with absolutely no cost up front

- Strong community to support the developers
- Continuous updates
- Wide adaptation in the industry

Framework of Frameworks

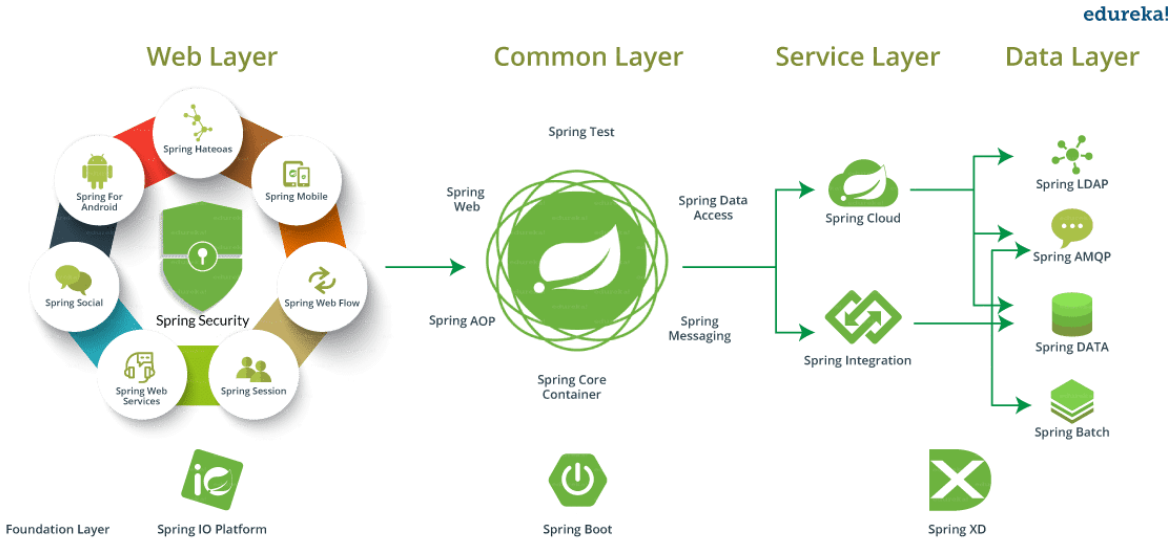
Spring is not just another java framework. It is a framework of frameworks. It provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.

Spring has evolved over time

Since its origin till date, Spring has spread its popularity across various domains. Spring Framework now is the foundation for various other Spring Projects that have come up in the offerings in the last two to three years.

Easy Development & Testing Cycles

Spring Framework follows best practices for coding and testing thereby making the development of Java EE application easy.



History of Spring Framework

Spring was initially developed by Rod Johnson in 2002

1. The first version of the Spring framework was written by **Rod Johnson** along with a book – "*Expert One-on-One J2EE Design a nd Development*" in October 2002.
2. The framework was first released in **June 2003** under the **Apache license version 2.0.**
3. The first milestone release of **Spring framework (1.0)** was released in March **2004.**
4. Spring 2.0, which came in 2006, simplified the XML config files.
5. Spring 2.5, which came in 2007, introduced annotation configurations.
6. Spring 3.2, which came in 2012, introduced Java configuration, had support for Java 7, Hibernate 4, Servlet 3.0, and also required a minimum of Java 1.5.

7. Spring 4.0, which came in 2014, had support for Java 8.
8. Spring Boot also was introduced in 2014.
9. Spring 5.0 came out in 2017. Spring Boot 2.x has support for Spring 5.

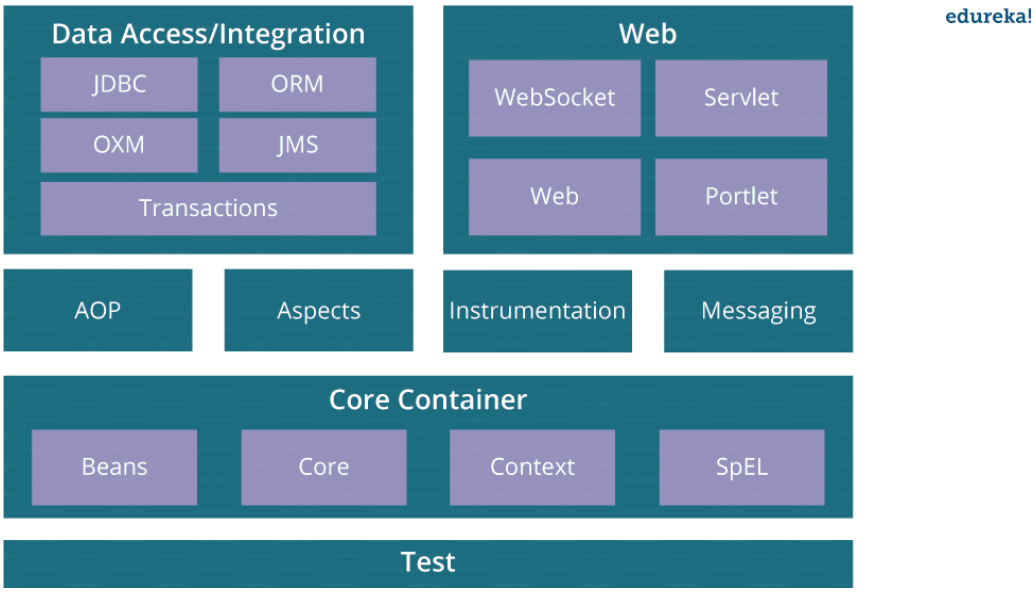
Official Website for Spring Framework: https://spring.io/

Spring Framework Architecture

Spring Framework architecture is an arranged layered architecture that consists of different modules. All the modules have their own functionalities that are utilized to build an application.

There are around **20 modules** that are generalized into **Core Container**, **Data Access/ Integration**, **Web**, **AOP** (Aspect Oriented Programming), **Instrumentation**, and **Test**.

Here, the developer is free to choose the required module. Its modular architecture enables integration with other frameworks without much hassle.



Spring Core Container

This container has the following **four modules**:

Spring Core

This module is the core of the Spring Framework. It provides an implementation for features like IOC (Inversion of Control) and Dependency Injection with a singleton design pattern. The main concepts of Spring core are DI, IOC, Property Injection & Constructor Injection

Spring Bean

This module provides an implementation for the factory design pattern through BeanFactory. It is an important module and is used to inject objects in other class

Spring Context

Context module inherits features from bean module and adds to it: internationalization, event propagation, resource loading, and transparent creation of context.

This module is built on the solid base provided by the Core and the Bean modules and is a medium to access any object defined and configured.

Spring Expression Languages (SpEL)

This module is an extension to expression language supported by Java server pages. It provides a powerful expression language for querying and manipulating an object graph, at runtime.

Spring Data Integration and Data Access

It consists of the following five modules:

Spring JDBC

This module provides JDBC abstraction layer which eliminates the need of repetitive/hideous and unnecessary exception handling overhead JDBC code.

Spring ORM

ORM stands for Object Relational Mapping. This module provides consistency/portability to our code regardless of data access technologies based on object oriented mapping concept.

It provides integration of any other ORM tool in our spring application such as integration of hibernate to our code

Spring OXM

OXM stands for Object XML Mappers. It is used to convert the objects into XML format and vice versa. The Spring OXM provides a uniform API to access any of these OXM frameworks. It provides an abstraction layer to integrate with object xml mapping tools such as Castor, Xstream etc.

Spring JMS

JMS stands for Java Messaging Service. This module contains features for producing and consuming messages among various clients/java applications.

Transaction

This module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs. All the enterprise level transaction implementation concepts can be implemented in Spring by using this module.

Spring Web

Web layer includes the following modules:

Web

This module uses servlet listeners and a web-oriented application context to provide basic web-oriented features to help us create web projects. We can use MVC, Rest APIs, Socket, Portlets, and Http Client etc. in the applications

Web-Servlet

This module contains Model-View-Controller (MVC) based implementation for web applications. It provides all other features of MVC, including UI tags and data validations.

Web-Socket

This module provides support for WebSocket based and two-way communication between the client and the server in web applications.

Web-Portlet

This module is also known as the Spring-MVC-Portlet module. It provides the support for Spring-based Portlets and mirrors the functionality of a Web-Servlet module.

Web Services and Rest API

Aspect-Oriented Programming (AOP)

AOP language is a powerful tool that allows developers to add enterprise functionality to the application such as transaction, security etc. It allows us to write less code and separate the code logic. AOP uses cross-cutting concerns, defines method interceptors, point cuts, code decoupling.

When we want to do something before or after function calls, this is known as method interceptor.

Instrumentation

This module provides class instrumentation support and class loader implementations that are used in certain application servers.

Messaging

It serves as a foundation to build a message based application. It uses annotations to do so. It could be used whenever we need messaging functionality in applications

Test

This module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring **ApplicationContexts** and caching of those contexts. It also provides mock objects that we can use to unit test our code in isolation.

What are Spring Framework, Spring JDBC, Spring MVC and Spring Boot?

Spring Framework

Spring Framework is a collection of modules. Spring JDBC and Spring MVC are modules of Spring Framework.

Spring JDBC

Spring JDBC Framework takes care of all the low-level details starting from opening the connection, preparing and executing the SQL statement, processing exceptions, handling transactions, and finally closing the connection.

Spring MVC

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

Spring Boot

Over the past few years, due to added functionalities, the Spring framework has become increasingly **complex**. It requires going through a lengthy procedure in order to start a new Spring project. To avoid starting from scratch and save time, Spring Boot has been introduced. This uses the Spring framework as a foundation.

While the Spring framework focuses on providing flexibility to you, **Spring Boot aims to shorten the code length** and provide you with the easiest way to develop a web application. With annotation configuration and default codes, Spring Boot shortens the time involved in developing an application. It helps create a stand-alone application with less or almost zero-configuration.

Prerequisites

To learn Spring Framework, one must have knowledge of following technologies:

- Core Java - OOP concepts: class, objects, constructor, method, overloading, overriding
- JDBC: Data integration layer/Access layer for Spring JDBC
- Hibernate to learn Spring ORM: Spring Hibernate Integration
- Servlet and JSP: Spring Web MVC
- Important Web and Database related terms: HTML, CSS, JS, Bootstrap, MySQL/PostGres/any database => working knowledge

Spring Core

Table of Contents

Dependency Injection

The main concept of spring core is dependency injection (DI) and inversion of control (IOC). DI is done with the help of IOC container

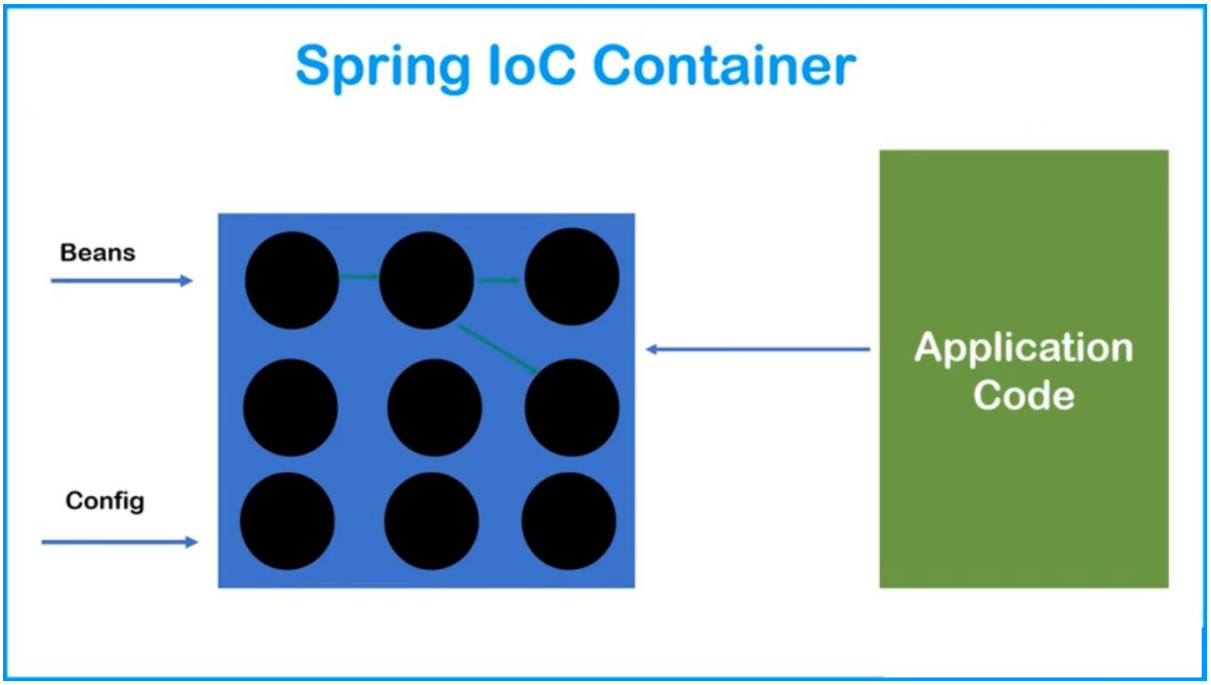
Spring IOC container

One of the main features of the Spring framework is the IoC (Inversion of Control) **container**. The Spring IoC container is responsible for managing the objects of an application. It uses dependency injection to achieve inversion of control.

We can say that Spring IOC container is a program/container in Spring framework whose main tasks are:

- Objects creation
- Holding objects in memory
- Inject an object in another object, i.e. DI
- Maintain Object lifecycle: creation to destruction

We need to tell IOC about dependencies and this we do with the help of beans & configurations. As soon as the dependencies are done, these are made available to the application code on demand



IOC API

The interfaces BeanFactory and ApplicationContext represent the Spring IoC container.

BeanFactory Interface

- BeanFactory is the root interface for accessing the Spring container.
- It provides basic functionalities for managing beans.

ApplicationContext Interface

- ApplicationContext is the sub-interface of the BeanFactory.
- Hence, it offers all the functionalities of BeanFactory.
- ApplicationContext provides more enterprise-specific functionalities.
- The important features of ApplicationContext are:
    - Resolving messages,
    - Supporting internationalization,
    - Publishing events, and
    - Application-layer specific contexts.

    This is why we use ApplicationContext as the default Spring container.

Bean

Before we dive deeper into the *ApplicationContext* container, it's important to know about Spring beans.

- The objects that form the backbone of a Spring application and that are managed by the Spring IOC container are called **beans**.
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IOC container.

- These beans are created with the configuration metadata that is supplied to the container, such as:
    - Fully qualified name of the class
    - Name of the object that can be used as a reference
    - Property setters
    - Constructor setters
- The Spring configuration XML file has <beans></beans> tag as the root tag. All the other beans must be nested inside the root tag inside individual <bean></bean> tag

So, should we configure all the objects of our application as Spring beans? Well, as a best practice, we should not.

As per Spring documentation, in general, we should define beans for:

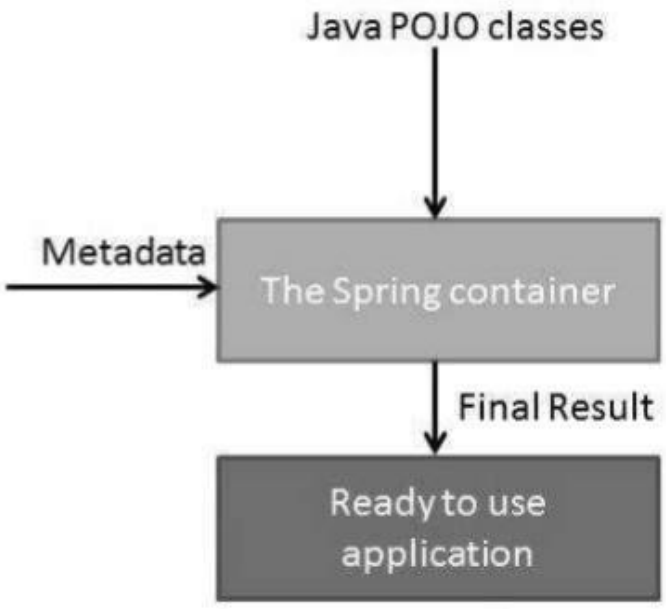
1. Service layer objects,
2. Data access objects (DAOs),
3. Presentation objects,
4. Infrastructure objects such as Hibernate *SessionFactories*,
5. JMS Queues, and so forth.

Spring Configuration Metadata

The application needs to provide bean configuration to the IoC container (ApplicationContext Container) about what beans/classes are to be injected in other classes. Therefore, a Spring bean configuration consists of one or more bean definitions.

Bean definition contains the information called as **configuration metadata**, which is needed for the container to know the following −

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies



Also, Spring supports different ways of configuring beans.

Types of Configurations

Following are the three important methods to provide configuration metadata to the Spring Container −

- XML based configuration file.
- Annotation-based configuration
- Java-based configuration

Types of Dependency Injections

There are 2 types of dependency injections:

1. Property Injection (also known as Setter Injection)
2. Constructor Injection

What to Inject?

1. Primitive Data Types: Setter Injection/Constructor Injection
2. Reference Type: Setter Injection/Constructor Injection
3. Collections: Setter Injection/Constructor Injection

There are two basic types of containers in Spring – the Bean Factory and the Application Context. The former provides basic functionalities, which are introduced here; the latter is a superset of the former and is most widely used

The application code can use these beans

Application Context Interface

- ApplicationContext represents IOC. It extends BeanFactory
- ApplicationContext is an interface in the org.springframework.context package and it has several implementations
- We can fetch the objects from an IOC by the above interface by forming an object of its implementing class

Implementing Classes

1. **ClasspathXMLApplicationContext**
    - It searches XML configuration from Java class path
    - ClassPathXmlApplicationContext can load an XML configuration from a class-path and manage its beans
2. **AnnotationConfigApplicationContext**
    - It searches the beans on which we have used annotation.
3. **FileSystemXMLApplicationContext**
    - It searches config from a file.

We have different XML configurations for Development, Staging, QA, Production

Depending upon XML to initialize

==========

Ways of Injecting dependencies | Types of dependencies handled by IOC Container

==========

DI:

```
class Student {
	int id;
	String name;
	Address address; // Dependency
}

class Address {
	String street;
	String city;
	String state;
	String country;
}
```

Spring Framework (IOC) will create an Address object and set the values of variables at runtime. It will also create the Student objects and its varoiables value at runtime.

Then we will ask the students and adreess's objects from IOC container.

IOC-creates the dependencies and inject automatically

OOC can do DI in 2 ways:

1. Using setter/property Injection

2. Using Constructor Injection

1. Using Setter/Property Injection

```
class Student {
        int id;
        String name;
        Address address; // Dependency

        setId(id){}
        setName(name){}
        setAddress(address){}
}

class Address {
        String street;
        String city;
        String state;
        String country;
        setStreet(street){}
        setCity(city){}
        setState(state){}
        setCountry(country){}

}
```

IOC container will use these setter methods at runtime to set values after creating the objects automatically.

We can then ask IOC to provide us the objects

2. Using Constructor Injection

```
class Student {
        int id;
        String name;
        Address address; // Dependency
        Student(id, name, address){}
}

class Address {
        String street;
        String city;
        String state;
        String country;
        Address(street, city, state, country){}
}
```

Here the IOC will use the constructor to set values to thye created objects

We can provide all this information by

1. XML Configuration
2. Java Configuration
3. Annotation Based Configuration

XML Configuration file

If we want to use DI featuer with the help of IOC, We have to provide the details of the class in the configuration file only then IOC will maintain its lifecycle. It is an XML file where we declare beans and their dependencies.

Beans

The classes whose information is passed to an IOC are known as beans. These beans have to follow some rules:

<beans>

<bean>

</bean>

</beans>

We need to specify the type of data that needs to be injected through IOC:

Data Types:

1. Primitive Data Types

byte, short, char, int, long, boolean, float, double

2. Collection Types

List, Set, Map and Properties: The IOC will be able to inject any List/... object in other class

3. Reference Type

Other class object

Topic 00: New Maven Project | Adding Spring Dependencies | Create Config File | Setter Injection

Softwares:

1. Eclipse/Netbeans/IntelliJ

2. Tomcat Server

3. MySQL for DB

4. SQL Yog / workbench / phpmyadmin

Steps to create a project:

1. Create Maven Project: quickstart archetype

2. Adding dependencies=>spring core, spring context

3. Creating beans=> Java POJO

4. Creating configuration file=> config.xml

5. Setting Injection

6. Main class: which can pull the object and use

Topic 01: Property Injection - primitive

Property injection using p Schema and using value as attribute

Project: 01Injection

Package: property.injection.primitive

Topic 02: Property Injection - Reference |

Injecting Reference Type

Project: 01Injection

Package: property.injection.reference

Topic 03: Property Injection Collections |

Injecting Collection Types[List , Set , Map , Properties]

Project: 01Injection

Package: property.injection.collections

Topic 04: Standalone Collections

Spring Standalone Collections[List,Map,Properties] | Util Schema in Spring

Project: 01Injection

Package: standalone.collections

While creating a list/set in a bean, these collections are dependent, i.e they can't be used again.

Second, we dont specify the type of list/set that is injected

To overcome all this, we use standalone collections

Topic 05: Constructor Injection

Project: 01Injection

Package: constructor.injection

Topic 06: Constructor Ambiguity |

Ambiguity Problem and its Solution with Constructor Injection

Project: 01Injection

Packages: constructor.ambiguity.one, constructor.ambiguity.two

Case 1:

If you observe the above output we didn't get expected result. ie. second constructor (float,String) argument was not called, instead first constructor (String,String) argument was called. This is the Constructor injection type ambiguity.

Spring container by default converts every passing value to String value. i.e. In our example 10000 converted to String. That's why second constructor with (String,String) argument was called.

Case 2:

If you observe we didn't get expected output. ie. third constructor with (float,String) argument was not called, instead second constructor with (String,float) argument was called. This is again ambiguity in Constructor injection type.

Solution:

We can resolve this problem by using index attribute of <constructor-arg> tag. So while using constructor injection always specify the exact datatype for constructor-arg value using type attribute and index attribute in beans.xml.

Topic 07: Stereotype Annotations | @Component Annotation | @Value Annotation

Spring provides us with three ways to define our beans and dependencies:

- XML configuration

- Java annotations

- Java Configuration Class

<context:component-scan base-package="bean.scope" />

This tag is needed when we use Annotations in Spring

Annotations:

@Component: to declare bean

@Value: to give value to a property

We have till now studied about declaring a bean in xml file for Spring/IOC container to inject objects.

In this example we will use stereotype annotations to declare a component and let IOC inject it with the help of

@Component and @value annotation

The default name of the object is camel case conversion of class name, however we can also mention a different name in @Component declaration

for ex: class Student will give student object.

If we wish other name, @Component("newName") must be used

Practical:

Inorder to inject a collection:

1. make a standalone collection in xml and give it values and id

2. Use spEL in class #{<id>} to mention collection

Project: 01Injection

Packages: stereotype.annotations

Topic 08: Spring Bean Scope | Singleton | Prototype | how to configure scope

Project: 01Injection

Packages: bean.scope.annotations

Bean Scope:

Spring Bean Scopes allows us to have more granular control of the bean instances creation. Sometimes we want to create bean instance as singleton but in some other cases we might want it to be created on every request or once in a session.

There are five types of spring bean scopes:

1. singleton
2. prototype
3. request
4. session
5. global-session

1. singleton – only one instance of the spring bean will be created for the spring container. This is the default spring bean scope.

whenever we configure a class in xml or through annotations, and ask the IOC to get the class instance, it will provide us with the same instance throughout the application. This is singleton scope. It is the scope of a bean set by default

2. prototype – To get different objects or to create a new instance every time,, we mention "prototype" scope

Spring MVC

Request and session scopes are used in web applications

3. request – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.

4. session – A new bean will be created for each HTTP session by the container.

5. global-session – This is used to create global session beans for Portlet applications.

Topic 09: Java Configuration (Removing Complete XML for Spring Configuration)| @Configuration | @ComponentScan | @Bean Annotation

Project: 01Injection

package: com.java.configuration

There are 3 ways to configure beans:

1. XML

2. Annotations

3. Java Config Class

Java Config Class

@Configuration

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.

@ComponentScan

Spring needs the information to locate and register all the Spring components with the application context when the application starts. Using @ComponentScan Spring can detect Spring-managed components.

@Bean

The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

When we use @Bean on method name, the method name acts as the bean id: for example: "getSubject" & "getStudent"

However we can specify another name(s) for beans by using name array of @Bean annotation: Example @Bean(name={"name1","name2"})

Topic 10: Spring Bean Lifecycle

Life Cycle Methods: Spring provides 2 important methods to every bean

1. public void init(): Initialization code - Loading config, connecting db, webservices etc
2. public void destroy(): Clean up code

Configure Technique: How to call these methods:

1. XML
2. Spring Interface
3. Annotation

Project: 02SpringBeanLifecycle

registerShutdownHook():

In spring non-web based applications, registerShutdownHook() method is used to shut down IoC container. It shuts down IoC container gracefully.

In non web based application like desk top application it is required to call registerShutdownHook().

In our desktop application we need to release all resources used by our spring application. So we need to ensure that after application is finished, destroy method on our beans should be called.

In web-based application, ApplicationContext already implements code to shut down the IoC container properly. But in desktop application we need to call registerShutdownHook() to shutdown IoC container properly.

1. Init and Destroy through XML

Mention the names of init and destroy methods in xml

init-method="init"

destroy-method="destroy"

and define the methods with same names in bean class

This gives us the flexibility to change the name of these methods, however we must keep in mind that the signature must be the same

Implementing bean life cycle using interfaces | IntializingBean | DisposableBean |

2. Init and Destroy using Spring Interfaces

The bean class for which we need init and destroy methods must implement:

1. IntializingBean Interface and define the method afterPropertiesSet()

2. DisposableBean Interface and define the method destroy()

Implementing Bean LifeCyle using Annotations | @PostConstruct | @PreDestroy

3. Init and Destroy using Spring Annotations

Use annotations

1. @PostConstruct for init() method

2. @PreDestroy for destroy() method

We can set any name for init() and destroy(). The important things are the annotations

We have to specify <context:annotation-config></context:annotation-config> in xml

to activate @PostConstruct & @PreDestroy

Topic 11: Autowiring

Project: 03AutoWiring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection. Autowiring can't be used to inject primitive and string values.

Advantage of Autowiring

It requires less code because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

- No control of programmer.

- It can't be used for primitive and string values.

Autowiring can be done in 2 ways:

1. Through XML

2. Through Annotations

Topic 12: Through XML

There are many autowiring modes:

1) no: It is the default autowiring mode. It means no autowiring by default.

2) byName: The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.

```
class Employee {
        Address ad;
}
```

```
<bean
        class="Address"
        name="ad"
```

/>

a. byName matches class Employee {Address instance name} with Address bean name in xml

b. byName is used when there are more than 1 bean of the same type

3) byType: The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.

byType can be used when there is only one bean of a particular type

It is not necessary to give name in dependency bean class and autowiring is done bytype

4) constructor: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

It is not necessary to give name in dependency bean class and autowiring is done by constructor

We can not mention 2 depency beans as IOC will confuse whom to inject

If 2 constructors of the same number of parameters and same type are defined, the IOC container will call the 1st constructor defined

5) autodetect: It is deprecated since Spring 3

Project: 03AutoWiring

packages: auto.wiring.by.name, auto.wiring.by.type, auto.wiring.by.constructor

---

Topic 13: @Autowired Annotation for Autowiring

@Autowired Annotation: byName is called implicitly

Starting with Spring 2.5, the framework introduced annotations-driven Dependency Injection. The main annotation of this feature is @Autowired. It allows Spring to resolve and inject collaborating beans into our bean.

The Spring framework enables automatic dependency injection. In other words, by declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans. This is called Spring bean autowiring.

@Qualifier Annotation

By default, Spring resolves autowired entries by type.

If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException, indicating that more than one bean is available for autowiring.

Let's imagine a situation in which two possible candidates exist for Spring to inject as bean collaborators in a given instance:

```
@Component("fooFormatter")
public class FooFormatter implements Formatter {

    public String format() {
        return "foo";
    }
}

@Component("barFormatter")
public class BarFormatter implements Formatter {

    public String format() {
        return "bar";
    }
}
```

```
@Component
public class FooService {

    @Autowired
    private Formatter formatter;
}
```

If we try to load FooService into our context, the Spring framework will throw a NoUniqueBeanDefinitionException. This is because Spring doesn't know which bean to inject. To avoid this problem, there are several solutions. The @Qualifier annotation is one of them.

3. @Qualifier Annotation

By using the @Qualifier annotation, we can eliminate the issue of which bean needs to be injected

By using the @Qualifier annotation, we can eliminate the issue of which bean needs to be injected.

Let's revisit our previous example and see how we solve the problem by including the @Qualifier annotation to indicate which bean we want to use:

```
public class FooService {

    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

By including the @Qualifier annotation together with the name of the specific implementation we want to use – in this example, Foo – we can avoid ambiguity when Spring finds multiple beans of the same type.

We need to take into consideration that the qualifier name to be used is the one declared in the @Component annotation.

Note that we could've also used the @Qualifier annotation on the Formatter implementing classes, instead of specifying the names in their @Component annotations, to obtain the same effect:

```
@Component
    @Qualifier("fooFormatter")
    public class FooFormatter implements Formatter {
      //...
    }

    @Component
    @Qualifier("barFormatter")
    public class BarFormatter implements Formatter {
      //...
    }
```

Topic 14 - Topic 24: Moved

Video 42: Spring ORM Tutorial: moved

Topic 25 - Topic 30: Moved

Introduction to Spring MVC

Contents

Spring MVC

What is MVC?

MVC stands for Model View Controller. It is a design pattern. A design pattern helps us to develop efficient code. It is a way to organize code in a nice and cleaner way in our application

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern separates the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

1. The **Model** encapsulates the **application data** and in general they will consist of POJOs.
2. The **View** is responsible for rendering the model data and in general it **generates HTML output** that the client's browser can interpret.
3. The **Controller** is responsible for **processing user requests** and building an appropriate model and passing it to the view for rendering.

In simple, layman’s terms we can say:

- Model – Data in an application
- View – Presents Data to User
- Controller – Interface between Model and View, controls the flow of application

We can implement MVC Design Pattern in every programming language and in every type of application. It is not limited just to java or to Spring.

MVC in Servlets and JSP

We did the same thing while developing a web application with Servlets, JSPs, DAOs & POJOs.

- Servlets worked has Controllers, i.e. they accepted the request, processed the data, generated the response and sent it to the JSPs
- Java Server Pages worked as Views, i.e. they presented the processed data to the viewer
- DAOs & POJOs: worked as Models, i.e. by handling the data

What is Spring MVC?

Learning and using Spring MVC is the 1st step for using Spring for Web Development

1. Spring MVC is a module/sub-framework of Spring Framework used to build a **web application**. For example: any website that we can open on any browser.
2. It is built on the top of **Servlet API.**
3. It follows the **Model-View-Controller** design pattern.
4. It implements all the basic features of Spring Core Framework like Inversion of Control, Dependency Injection etc.

Why Spring MVC?

Problem without MVC Design Pattern

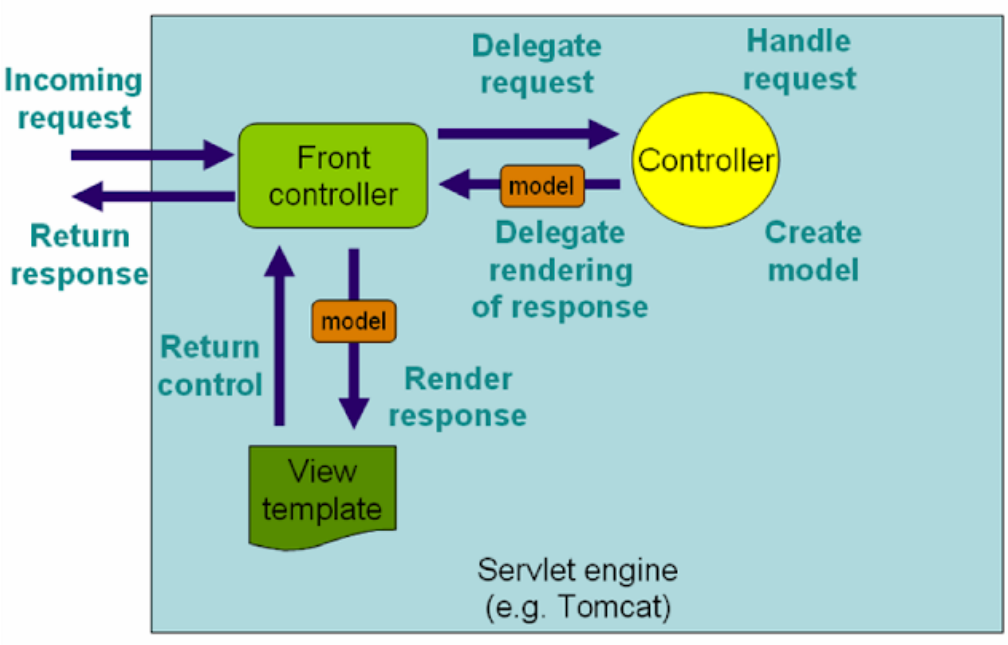
1. The code becomes tedious to maintain
2. The code becomes error prone and
3. The project requires longer development and testing cycles
4. Longer development and testing cycles means higher costs and **loss of business**

What Spring MVC Offers?

1. Separates the roles into: Model, View and Control
2. It provides xml based, annotation based or java based powerful configuration
3. It provides all the basic features of core spring framework such DI, IOC that helps us loosely couple our project
4. It helps in Rapid Application Development
5. Spring MVC is flexible, easy to test and much features. We can also use other frameworks with Spring MVC, for example: Velocity instead of JSP, Hibernate instead of JDBC etc.

Working of Spring MVC

Here is the flow of an HTTP request in Spring MVC Web application



1. The client sends an HTTP request to a specific URL
2. Front Controller (DispatcherServlet) of Spring MVC receives the request
3. It passes the request to a specific controller depending on the URL requested using @Controller and @RequestMapping annotations.
4. Spring MVC Controller then returns a logical view name and model to DispatcherServlet.
5. DispatcherServlet consults view resolvers and uses prefix and suffix properties to fetch the view until actual View is determined to render the output
6. DispatcherServlet contacts the chosen view (like Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data
7. The rendered output is returned to the client as a response

First Spring MVC Project

Contents

First Spring MVC Web Application

Let us create a minimalistic Spring Web MVC application to learn the configurations and the control flow:

Setup Development Environment

Install and configure latest

1. JDK
2. Eclipse for Java EE Developers
3. Apache Tomcat Server
4. Apache Maven
5. MySQL Server and Workbench if you wish to work with database

Create the Project

Create and Setup a Maven Project

1. Create a new Maven project and select – Catalog: “Internal” Filter: “maven-archetype-webapp” and other related parameters for project setup
2. Configure Tomcat Server

3. Configure the build path
4. Include Spring Web MVC dependency in pom.xml

Configure the Project

Step 1: Configure the Front Controller

Configure the front controller (DispatcherServlet) in web.xml (present in WEB-INF folder)

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
     org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Points to remember

- The servlet name can be anything – a name of your choice. However, it must be relatable.
- The servlet class must be a fully qualified name, i.e. the class name along with the package name. We can get the class definition by pressing Ctrl + Shift + T and then searching the class by name.

Step 2: Create Spring Configuration File

1. Create an XML file in **the same folder (WEB-INF**) where web.xml is.
2. The name of the file must be: <DispatcherServlet name> "-" <servlet>.
   For example: In our case **dispatcher-servlet.xml**
3. Include the <beans></beans> tags with proper namespaces

```
<?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:p="http://www.springframework.org/schema/p"
       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">
</beans>
```

Step 3: Create Controller

In the java folder, in a proper package, create a java class with **@Controller** annotation

For a starter, define a method inside this class that will help us in displaying our first JSP page

```
package employee.manager.controller
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/employee")
public class EmployeeController {
   @RequestMapping("/")
   public String redirectToEmpHome() {
      return "emp-home";
   }
   @RequestMapping("/add")
   public String addEmployee() {
      // Add Employee code
      return "add-employee";
   }
}
```

Step 4: Configure Annotation Scanner in Config

Include the following line as the first line inside <beans> tag in dispatcher-servlet.xml file:

```
<context:component-scan base-package=" employee.manager.controller"/>
```

Points to Remember:

- <**context**:**component-scan**> detects the annotations by package scanning. It tells Spring which packages need to be **scanned** to look for the annotated beans or **components**.
- As Controller class has @Controller and @RequestMapping annotations, the controller's package must be mentioned in context:component-scan for it to work properly

Step 5: Create views

1. Create a folder under WEB-INF directory with the name: "views".
2. Create 2 files: "home.jsp" & "add-employee.jsp" inside "views" folder

Step 6: Configure View Resolver

Configure the InternalResourceViewResolver bean inside the configuration file along with prefix and suffix properties

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" name="viewResolver">
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
</bean>
```

Points to remember:

- InternalResourceViewResolver class prepares a full path to the view page
- Set the path to JSP view files in prefix property
- Set the extension of JSP view files in suffix property
- During runtime, the view resolver will prepare the view file name by concatenating prefix & suffix with the file name returned by the controller & present it to the user.
  For example: “/WEB-INF/views/” + emp-home + ".jsp", i.e. “/WEB-INF/views/emp-home.jsp"

Control Flow in Spring MVC Project

We can say that there are following important processes in a Spring Web MVC application:

1. The client triggers the request
2. The front controller delegates the request to the appropriate controller
3. The controller sends the request to the appropriate request handler
4. The request handler:
    1. Receives the request
    2. Collects the request parameters
    3. Processes the data
    4. Sends the processed data and the target view to the front controller

Let us see how Spring MVC achieves the above flow

Step 1: The Client initiates the Request

There are different ways through which a request can be triggered by the client such as:

1. When user clicks on a link
2. When a form is submitted
3. When some control passes from one server component to another
4. Through redirections

Every time any such thing happens, a request is triggered and every request is represented by a URL that specifies the request name

For example:

1. http://localhost:8080/<Project-Name>/employee/
2. http://localhost:8080/<Project-Name>/employee/add

Step 2: The Role of the Front Controller

As you can see in the above section, the requests triggered are “/employee/” and “/employee/add”, in order handle them, we have to create some handler controllers and methods.

```
@Controller
@RequestMapping("/employee")
public class EmployeeController {
   @RequestMapping("/")
   public String redirectToEmpHome() {
      return "emp-home";
   }
   @RequestMapping("/add")
   public String addEmployee() {
      // Add Employee code
      return "add-employee";
   }
}
```

- The request reaches the DispatcherServlet (Front Controller)
- The request will be received by DispatcherServlet (Front Controller)
- DispatcherServlet will take the help of Handler Mapping and will delegate the request to the Controller class (EmployeeController) associated with the given request: “/employee” in the above case.

Step 3: Controller hands the request to handler method

- The Controller (EmployeeController) passes the request to the appropriate handler method:
  1. “/” to “redirectToEmpHome()” and
  2. “/add/” to “addEmployee()” in the above case.

Step 4: The handler method receives the request

- The handler method receives the request
- In case there are request parameters, the handler method extracts the data with the help of:
  1. HttpServletRequest object
  2. @RequestParams annotation
  3. @ModelAttribute annotation

  We will study about these concepts in the later sections

Step 5: The handler method processes the data

The handler method performs following actions on the data

1. Authentication
2. File IO
3. Database transaction
4. etc.

Step 6: The handler method returns data and view

The handler method after processing the data returns the following back to the DispatcherServlet:

1. The processed data
2. The name of the view to render the data, here: ” emp-home” and “add-employee”

The handler can send the data and view name to the front controller with the help of:

1. Model Interface
2. ModelMap class
3. ModelAndView class

We will study about these concepts in the later sections

Step 7: DispatcherServlet sends data and view to the ViewResolver

1. The view resolver will prepare the path of the file by concatenating prefix and suffix
    - “emp-home” becomes “/WEB-INF/views/emp-home.jsp”
    - “add-employee” becomes “/WEB-INF/views/add-employee.jsp”
2. The view resolver will then return the full path to the front controller.

Step 8: Front Controller sets data in View

The Dispatcher Servlet will pass the data/model object to the View page to display the result

Full control flow

After configuring the project as described in the above section, run the application on server

1. When the project will be executed, observe the address bar – it will show http://localhost:8080/<Project-Name>/
2. This request will originate from the client and travel to the front controller (DispatcherServlet)
3. DispatcherServlet will delegate this request to the controller class (that has @Controller annotation), in our case to the EmployeeController class.
4. The request will go to the handler method that is annotated with @RequestMapping annotation and configured to handle the “/” request.

For example: In our case: @RequestMapping (value = "/", method = RequestMethod.GET) redirectToIndex method

5. The method will receive the “/” request and return the name: “emp-home” to the front controller
6. The front controller will call the view resolver declared in spring config file
7. The view resolver will prepare the path of the file by concatenating prefix and suffix
8. The view resolver will then return the full path to the front controller.
9. In case there is some data, the front controller will set it in the view
10. The front controller will send the view back to the client

Introduction to Spring Boot

Table of Contents

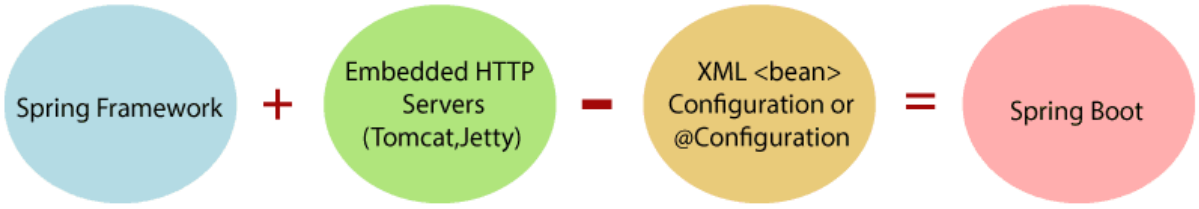
What is Spring Boot?

Spring Boot is a project that is built on top of the Spring Framework.

Funnily said - **Spring Boot is Spring on steroids**. It's a great way to get started very quickly with almost the entire Spring stack.

1. Spring Boot is a module of Spring from which we can speed up development

2. Spring Boot makes it easy to create stand-alone, **production grade** Spring based Applications that we can just run
3. It provides an easier and faster way to set up, configure and run both - simple and web based applications
4. **Spring Boot** = **Spring Framework + Embedded Servers – Configuration**



5. Spring boot follows "**Convention over Configuration**" software design style, i.e. if we follow spring boot's coding and project conventions, Spring boot will take care of the configurations. It decreases the effort of the developer

6. Spring Boot follows "**Opinionated Defaults Configuration**", i.e. if we include "**spring-boot-starter-data-jpa**", Spring Boot will automatically configure - in memory database, a hibernate entity manager, and a simple data-source. Spring Boot scans the class path and find the dependency and then it will automatically configure the things

Pros of Spring Boot

1. It creates stand-alone Spring Applications that can be started using Java –jar, we don't need war file
2. It **embeds** Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
3. Provide opinionated 'starter' dependencies to simplify our build configuration
4. Automatically configure Spring and 3rd party libraries whenever possible
5. Provide production ready features such as metrics, health checks, and external configuration
6. It reduces lots of development time and increases productivity.
7. It **avoids** writing lots of **boilerplate** Code, Annotations and XML Configuration.
8. It is very **easy to integrate** Spring Boot Application with its Spring Ecosystem modules like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
9. It provides CLI (**Command Line Interface**) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly
10. It **provides** lots of **plugins** to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle

Cons of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

Version History of Spring Boot

1. Spring Boot 1.0 was released in April 2014

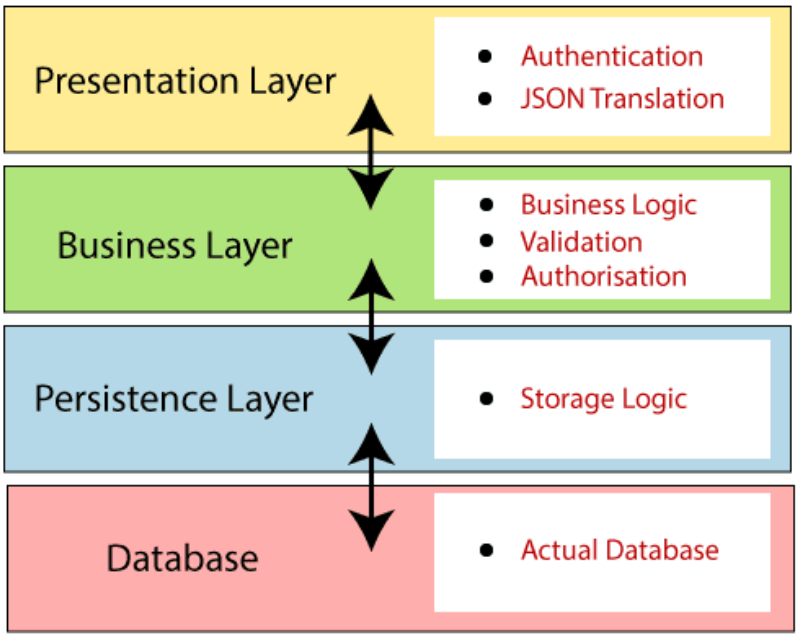
2. Spring Boot 2.0 was released in January 2017
3. Latest current version is **Spring Boot 2.5.3**

Spring Boot Architecture

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:

1. Presentation Layer
2. Business Layer
3. Persistence Layer
4. Database Layer



Presentation Layer

The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

Business Layer

The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

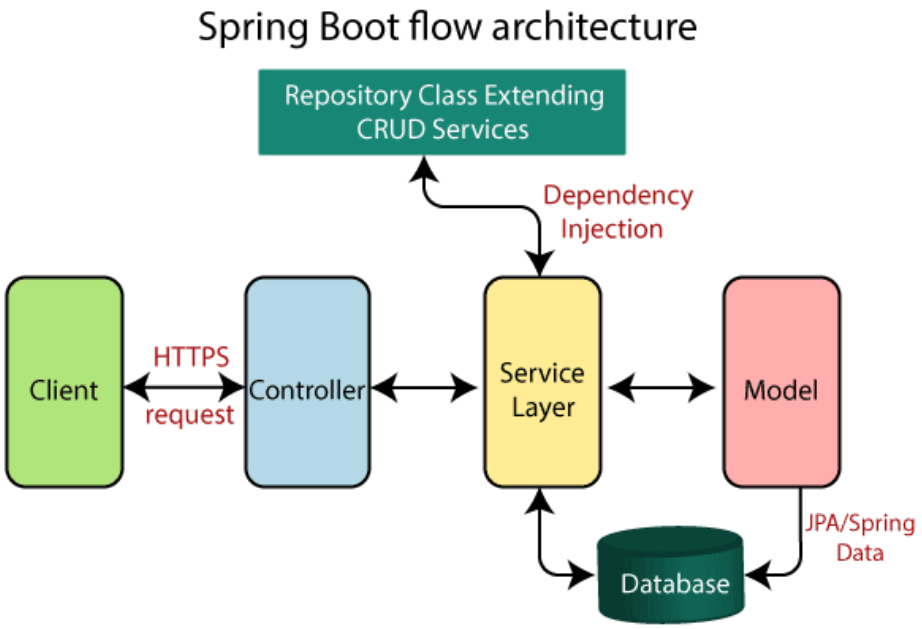
Persistence Layer

The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer

In the database layer, CRUD (create, retrieve, update, delete) operations are performed.

Spring Boot Flow Architecture



Steps:

1. The client makes the HTTP requests
2. The request goes to the controller, and the controller maps that request and handles it.
3. After that, it calls the service logic if required.
4. In the service layer, all the business logic performs.
5. Service Layer performs the logic on the data that is mapped to JPA with model classes.
6. A JSP page is returned to the user if no error occurred.

First Spring Boot Project

Table of Contents

Prerequisites

Following softwares and tools must be installed on the system before creating any Spring Boot Application

1. Any suitable Java version (LTS preferred)
2. Maven 3.0+
3. Eclipse/STS: **Spring Tool Suite** is recommended.

Ways of creating a Spring Boot Application

There are following ways to create a Spring Boot project:

1. With Eclipse (old method)

Create a maven project in **Eclipse** and add starter dependencies as jars

2. Use Spring Initializr and Eclipse/STS
3. **Use Spring Tool Suite IDE**
4. Spring Boot Command Line Interface
5. Using Spring plugin in Eclipse

Eclipse

1. Create a Maven project in Eclipse and add starter dependencies in the form of jar files
2. Use Spring plugin in Eclipse
3. By Spring Initializer: Eclipse is the most used IDE by Java Developers, so Spring Boot provides an initializer to make things easy for eclipse users.

Steps to create Spring Boot Application in Eclipse

1. Step 1: Search for Spring Boot Initializer on the internet
2. Step 2: Go to https://start.spring.io/
3. Step 3: Choose the options for the project:
    1. Project – Maven Project,
    2. Language – Java,
    3. Spring Boot – 2.3.5,
    4. Project Metadata,
    5. Packaging – Jar,
    6. Java – 11
4. Step 4: Add Dependencies
    1. Web
5. Step 5: Click Generate: The project will be downloaded as a zip file. Extract it.
6. Open Eclipse and Import the extracted project as “Existing maven Project”

Spring Tool Suite (STS)

1. Spring Tool Suite is an IDE to develop Spring applications.
2. It is an Eclipse-based development environment.
3. It provides a ready-to-use environment to implement, run, deploy, and debug the application.
4. It validates our application and provides quick fixes for the applications.

Steps to create Spring Boot Application in STS

1. Step 1: Search for STS download
2. Step 2: Go to: https://spring.io/tools
3. Step 3: Download 64 bit STS windows version (here 4.8.1)
4. Step 4: Execute the file by double clicking it
5. Step 5: Go to File>> New >> Spring Starter Project
6. Step 6: Choose the options for the project:
    1. Project – Maven Project,
    2. Language – Java,
    3. Spring Boot – 2.3.5,
    4. Project Metadata, such as project name and package: **spring.boot.demo**
    5. Packaging – Jar,
    6. Java - 11
7. Step 4: Add Dependencies
    1. Web

Note: We will use STS in our course to gain familiarity with it

Project Structure

Java Source Code

1. All the java source code must reside inside **src/main/java** folder.
2. The package we mentioned while making the project (here spring.boot.demo) is important.
3. It will be considered as a base package for all the java classes hereafter, i.e. all the Java files **must** either lie inside this package or **must be** written inside the sub package of this package for Spring Boot to scan them.
4. In this example, the class SpringDemoApplication is the java class from where the execution will begin.
5. SpringDemoApplication is annotated with **@SpringBootApplication** annotation. This annotation is the combination of following annotations:
    1. @Configuration,
    2. @EnableAutoConfiguration, and
    3. @ComponentScan
6. Just run the application as –
    1. Java Application in Eclipse or
    2. Run As >> Spring Boot App in STS.

Resources

1. All the static resources, configurations and themes are kept in this folder
2. **Static**: All the static resources such as HTML, CSS, JavaScript, images and other media files are contained in static folder.
3. **Themes**: It will contain any predefined themes such as for example: Thymeleaf.
4. File **application.properties**: All the properties such database configurations, context path, file path configurations must be done in application.properties file with the help of **key=value** pairs

Dependencies

We add **starter template jars** to our spring boot application such as: **spring-boot-starter-web**, **spring-boot-starter-data-jpa**, etc.

When we add starter jars, then Spring Boot pulls all the related jars. These Jar files contain the file spring.factories in META-INF folder, i.e. META-INF/spring.factories

If we use JPA, JPA configurations become active.

Spring boot scans the class path and if it finds JPA, all the configurations relating to JPA in spring.factories will become active, this will download spring.orm, hibernate, mysql connector etc

This is known as “Opinionated Defaults Configuration”

Rest APIs

Table of Contents

What is REST ?

REST stands for REpresentational State Transfer.

REST is web standards based architecture and uses HTTP Protocol for data communication.

It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources.

Here each resource is identified by URIs

REST uses various representations to represent a resource like text, JSON and XML. Most popular light weight data exchange format used in web services = JSON

HTTP Methods

Following well known HTTP methods are commonly used in REST based architecture.

1. GET - Provides a read only access to a resource.
2. POST - Used to create a new resource.
3. DELETE - Used to remove a resource.
4. PUT - Used to update a existing resource or create a new resource.

RESTFul Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems.

Platform & technology independent solution.

Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer.

This interoperability (e.g., between Java and Python, or Windows and Linux applications or java & .net) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services.

These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Background

Web services have really come a long way since its inception. In 2002, the World Wide Web consortium(W3C) had released the definition of WSDL(web service definition language) and SOAP web services. This formed the standard of how web services are implemented.

In 2004, the web consortium also released the definition of an additional standard called RESTful. Over the past couple of years, this standard has become quite popular. And is being used by many of the popular websites around the world which include Facebook and Twitter.

REST is a way to access resources which lie in a particular environment. For example, you could have a server that could be hosting important documents or pictures or videos. All of these are an example of resources. If a client, say a web browser needs any of these resources, it has to send a request to the server to access these resources. Now REST defines a way on how these resources can be accessed.

eg of a web application which has a requirement to talk to other applications such Facebook, Twitter, and Google.

Now if a client application had to work with sites such as Facebook, Twitter, etc. they would probably have to know what is the language Facebook, Google and Twitter are built on, and also on what platform they are built on.

Based on this, we can write the interfacing code for our web application, but this could prove to be a nightmare.

So instead , Facebook, Twitter, and Google expose their functionality in the form of Restful web services. This allows any client application to call these web services via REST.

What is Restful Web Service?

REST is used to build Web services that are lightweight, maintainable, and scalable in nature. A service which is built on the REST architecture is called a RESTful service. The underlying protocol for REST is HTTP, which is the basic web protocol. REST stands for REpresentational State Transfer

The key elements of a RESTful implementation are as follows:

1. Resources The first key element is the resource itself. Let assume that a web application on a server has records of several employees. Let's assume the URL of the web application is http://www.server.com. Now in order to access an employee record resource via REST, one can issue the command http://www.server.com/employee/1 - This command tells the web server to please provide the details of the employee whose employee number is 1.

2. Request Verbs - These describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE. So in the case of the example http://www.server.com/employee/1 , the web browser is actually issuing a GET Verb because it wants to get the details of the employee record.

3. Request Headers These are additional instructions sent with the request. These might define the type of response required or the authorization details.

4. Request Body - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web service. In a POST call, the client actually tells the web service that it wants to add a resource to the server. Hence, the request body would have the details of the resource which is required to be added to the server.

5. Response Body This is the main body of the response. So in our example, if we were to query the web server via the request http://www.server.com/employee/1 , the web server might return an XML document with all the details of the employee in the Response Body.

6. Response Status codes These codes are the general codes which are returned along with the response from the web server. An example is the code 200 which is normally returned if there is no error when returning a response to the client.

Restful Methods

The below diagram shows mostly all the verbs (POST, GET, PUT, and DELETE) and an example of what they would mean.

Let's assume that we have a RESTful web service is defined at the location. http://www.server.com/employee . When the client makes any request to this web service, it can specify any of the normal HTTP verbs of GET, POST, DELETE and PUT. Below is what would happen If the respective verbs were sent by the client.

POST This would be used to create a new employee using the RESTful web service

GET - This would be used to get a list of all employee using the RESTful web service

PUT - This would be used to update all employee using the RESTful web service

DELETE - This would be used to delete all employee using the RESTful web service

Exercise

Action	Request Method	URL	What it does
Get all Students	Get	/students	Get all students data from server
Get 1 student	Get	/students/2	Get 1 students data with id 2 from server
Add Student	Post	/students Student data	Sever will store this data on server side
Update Student	Put	/students Student data	Server updates the data
Delete Student	Delete	/students studentId	Server will delete the student data