CS-25 Advanced Java Programming (J2EE)

BCA Semester – 5

Prof. Dr. Malay B. Dave

Unit – 1

The J2EE Platform, JDBC (Java Database Connectivity)

Introduction to J2EE

The Java EE stands for Java Enterprise Edition, which was earlier known as J2EE and is currently known as Jakarta EE. It is a set of specifications wrapping around Java SE (Standard Edition). The Java EE provides a platform for developers with enterprise features such as distributed computing and web services. Java EE applications are usually run on reference run times such as micro servers or application servers. Examples of some contexts where Java EE is used are e-commerce, accounting, banking information systems.

As its name implies, Java EE is targeted at large-scale business systems. Software that functions at this level doesn't run on a single PC—it requires significantly more computing power and throughput than that. For this reason, the software needs to be partitioned into functional pieces and deployed on the appropriate hardware platforms. That is the essence of distributed computing. Java EE provides a collection of standardized components that facilitate software deployment, standard interfaces that define how the various software modules interconnect, and standard services that define how the different software modules communicate.

To reduce costs and fast-track application design and development, the Java™ 2 Platform, Enterprise Edition (J2EE™) provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE platform offers a multitiered distributed application model, reusable components, a unified security model, flexible transaction control, and web services support through integrated data interchange on Extensible Markup Language (XML)-based open standards and protocols. Like Java Platform, Standard Edition, Java EE consists of several packages that contain classes and interfaces that define the framework. You're already familiar with J2SE, and you gained your expertise by taking the J2SE framework one topic at a time.

Enterprise Architecture Styles:

One of the recurring themes that you'll run into with Java EE is the notion of supporting applications that are partitioned into several levels, or tiers. That is an architectural cornerstone of Java EE and merits a little explanation. If you are already familiar with n-tier application architectures, feel free to skip ahead. Otherwise, the overview presented here will be a good introduction or review that will help lay the foundation for understanding the rationale behind much of Java EE's design and the services it provides. If you think about a software application composition, you can break it down into three fundamental concerns, or logical layers:

- The first area of concern is displaying stuff to the user and collecting data from the user. That user interface layer is often called the presentation layer, since its job is to present stuff to the user and provide a means for the user to present stuff to the software system. The presentation layer includes the part of the software that creates and controls the user interface and validates the user's actions.
- Underlying the presentation layer is the logic that makes the application work and handles the important processing. The process in a payroll application to multiply the hours worked by the salary to determine how much to pay someone is one example of this kind of logic. This logical layer is called the business rules layer, or more informally the middle tier.
- All nontrivial business applications need to read and store data, and the part of the software that is responsible for reading and writing data—from whatever source that might be—forms the data access layer.

Single-Tier Systems

Simple software applications are written to run on a single computer. All of the services provided by the application—the user interface, the persistent data access, and the logic that processes the data input by the user and reads from storage—all exist on the same physical machine and are often lumped together into the application. That monolithic architecture is called single tier, because all of the logical application services—the presentation, the business rules, and the data access layers—exist in a single computing layer.

Single-tier systems are relatively easy to manage, and data consistency is simple because data is stored in only one single location. However, they also have some disadvantages. Single tier systems do not scale to handle multiple users, and they do not provide an easy means of sharing data across an enterprise. Think of the word

processor on your personal computer: It does an excellent job of helping you to create documents, but the application can be used by only a single person. Also, while you can share documents with other people, only one person can work on the document at a time.

Two-Tier (Client/Server) Architecture:
More significant applications may take advantage of a database server and access persistent data by sending SQL commands to a database server to save and retrieve data. In this case, the database runs as a separate process from the application, or even on a different machine than the machine that runs the rest of the program. The components for data access are segregated from the rest of the application logic. The rationale for this approach is to centralize data to allow multiple users to simultaneously work with a common database, and to provide the ability for a central database server to share some of the load associated with running the application. This architecture is usually referred to as client/server and includes any architecture where a client communicates with a server, whether that server provides data access or some other service.

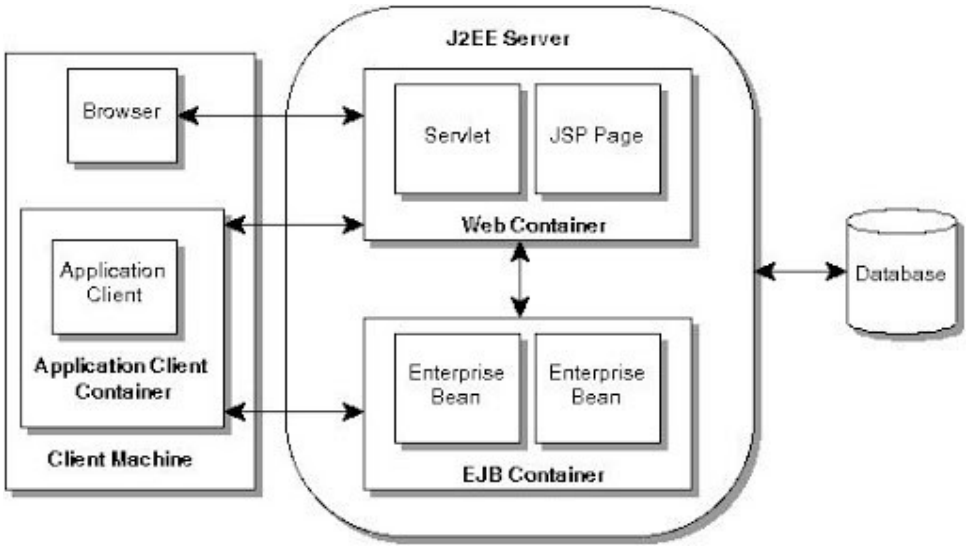
One of the disadvantages of two-tier architecture is that the logic that manipulates the data and applies specific application rules concerning the data is lumped into the application itself. This poses a problem when multiple applications use a shared database. Consider, for example, a database that contains customer information that is used for order fulfilment, invoicing, promotions, and general customer resource management. Each one of those applications would need to be built with all of the logic and rules to manipulate and access customer data. For example, there might be a standard policy within a company that any customer whose account is more than 90 days overdue will be subject to a credit hold. It seems simple enough to build that rule into every application that's accessing customer data, but when the policy changes to reflect a credit hold at 60 days, updating each application becomes a real mess.

N-Tier (Three-Tier) Architecture
A third tier added to the two-tier client/server model. In this model, all of the business logic is extracted out of the application running at the desktop. The application at the desktop is responsible for presenting the user interface to the end user and for communicating to the business logic tier. It is no longer responsible for enforcing business rules or accessing databases. Its job is solely as the presentation layer.

Typically, in a deployed application, the business logic tier executes on a server apart from the workstation (you'll see shortly that this isn't absolutely required, though). The business logic tier provides the logical glue to bind the presentation to the database. Since it's running on a server, it's accessible to any number of users on the network running applications that take advantage of its business rules. As the number of users demanding those services increases, and the business logic becomes increasingly complex and processor-intensive, the server can be scaled up or more servers can be added. Scaling a single server is a lot easier and cheaper than upgrading everyone's workstations.

One of the really great things that this architecture makes possible is the ability to start to build application models where the classes defined in the business logic tier are taken directly from the application domain. The code in the business logic layer can work with classes that model things in the real world (like a Customers class) rather than working with complex SQL statements. By pushing implementation details into the appropriate layer, and designing applications that work with classes modelled from the real world, applications become much easier to understand and extend.

Enterprise Architecture
Normally, thin-client multitier applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components and the J2EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.



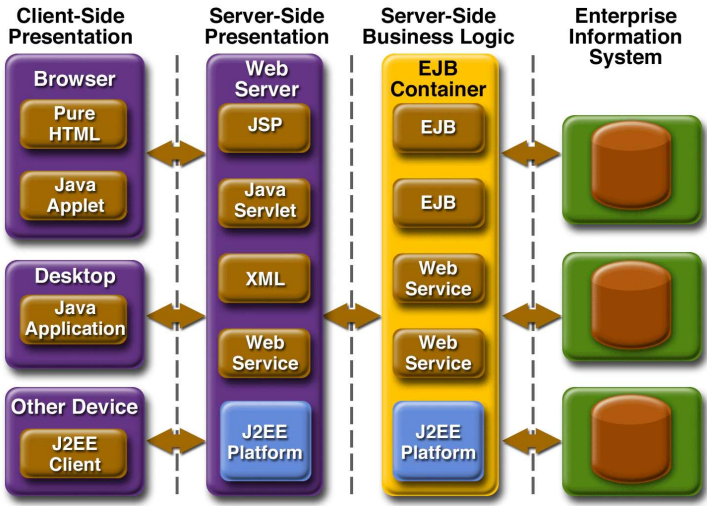
Component are installed in their containers during deployment and are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application and deployed into its container. The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE Server, which include services such as security, transaction management, Java Naming and Directory Interface TM (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a web component or enterprise bean so system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so application components can access naming and directory services.
- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

The fact that the J2EE architecture provides configurable services means that application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The J2EE Platform

The Application Server implements Java 2 Enterprise Edition (J2EE) technology. The J2EE platform is a set of standard specifications that describe application components, APIs, and the runtime containers and services of an application server. J2EE applications are made up of components such as JavaServer Pages (JSP), Java servlets, and Enterprise JavaBeans (EJB) modules. These components enable software developers to build large-scale, distributed applications. Developers package J2EE applications in Java Archive (JAR) files (similar to zip files), which can be distributed to production sites. Administrators install J2EE applications onto the Application Server by deploying J2EE JAR files onto one or more server instances (or clusters of instances).



Introduction to J2EE APIs (Servlet, JSP, EJB, JMS, JavaMail, JSF, JNDI)

The J2EE platform APIs in each J2EE container type. The following sections give a brief summary of the technologies required by the J2EE platform and the J2SE enterprise APIs that would be used in J2EE applications.

Enterprise JavaBeans Technology (EJB)

An Enterprise JavaBeans™ (EJB™) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server. As mentioned earlier, there are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. Enterprise beans often interact with databases. One of the benefits of entity beans is that you do not have to write any SQL code or use the JDBC™ API (see JDBC API, page 22) directly to perform database access operations; the EJB container handles this for you. However, if you override the default container-managed persistence for any reason, you will need to use the JDBC API. Also, if you choose to have a session bean access the database, you must use the JDBC API.

Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer Pages Technology (JSP)

JavaServer Pages™ (JSP™) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows J2EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API (JTA)

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The J2EE architecture provides a default auto commit to handle transaction commits and rollbacks. An auto commit means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend

on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

J2EE applications use the JavaMail™ API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP) supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation. JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: http://www.w3.org/XML/Schema.

Java API for XML-Based RPC

The Java API for XML-based RPC (JAX-RPC) uses the SOAP standard and HTTP, so client programs can make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL, so you can import and export WSDL documents. With JAX-RPC and a WSDL, you can easily interoperate with clients and services running on Java-based or non-Java-based platforms such as .NET. For example, based on the WSDL document, a Visual Basic .NET client can be configured to use a web service implemented in Java technology, or a web service can be configured to recognize a Visual Basic .NET client.

JavaServer Faces (JSF)

JSF is a relatively new technology that attempts to provide a robust, rich user interface for web applications. JSF is used in conjunction with Servlets and JSPs. When using just JSPs or Servlets to generate the presentation, your user interface is limited to what can be implemented in HTML. HTML does provide a good set of user interface components, such as lists, check boxes, radio buttons, fields, labels, and buttons. Alternatively, the client might be implemented as an applet. Applets can provide a rich user interface, but they do require the client to download and execute code in the browser. The main drawback with both Servlet-generated HTML and applets is that the user interface components still must be connected to the business logic. When using this solution, much of your time as a developer will be spent retrieving and validating request parameters, and passing those parameters to business logic components.

Java Database Connectivity (JDBC)

A database management system (DBMS) provides facilities for storing, organizing, and retrieving data. Most business applications store data in relational databases, which applications access via JDBC. The Application Server includes the Point Base DBMS for use sample applications and application development and prototyping, though it is not suitable for deployment. The Application Server provides certified JDBC drivers for connecting to major relational databases. These drivers are suitable for deployment.

Introduction to Containers

Containers are the interface between a component and the low-level platform specific functionality that supports the component. Before a web component, enterprise bean, or application client component can be executed, it must be assembled into a J2EE module and deployed into its container. The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application

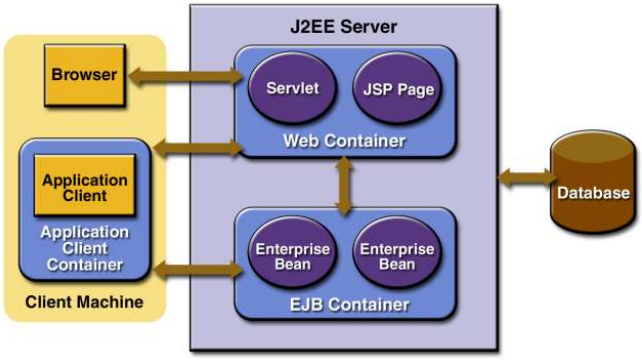
itself. Container settings customize the underlying support provided by the J2EE server, including services such as security, transaction management, Java Naming and Directory Interface™ (JNDI) lookups, and remote connectivity.

Because the J2EE architecture provides configurable services, application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages no configurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs described in section J2EE APIs. Although data persistence is a no configurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache.

Container Types

The deployment process installs J2EE application components in the J2EE containers.



- **J2EE server:** The runtime portion of a J2EE product. A J2EE server provides EJB and web containers.
- **Enterprise JavaBeans (EJB) container:** Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.
- **Web container:** Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.
- **Application client container:** Manages the execution of application client components. Application clients and their container run on the client.
- **Applet container:** Manages the execution of applets. Consists of a web browser and Java Plugin running on the client together.

Tomcat as a Web Container

The Apache Tomcat® software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket specifications are developed under the Java Community Process. The Apache Tomcat software is developed in an open and participatory environment and released under the Apache License version 2. The Apache Tomcat project is intended to be a collaboration of the best-of-breed developers from around the world. We invite you to participate in this open development project. Apache Tomcat software powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations.

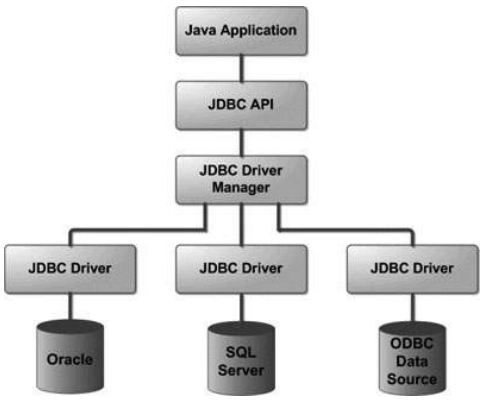
Introduction of JDBC

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases. The JDBC library includes APIs for each of the tasks that are commonly associated with database usage for: Making a connection to a database, Creating SQL or MySQL statements, Executing SQL or MySQL queries in the database, Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as: Java Applications, Java Applets, Java Servlets, Java ServerPages (JSPs), Enterprise JavaBeans (EJBs). All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers: JDBC-ODBC Bridge Driver, Native Driver, Network Protocol Driver, and Thin Driver.

JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers: **JDBC API,** this provides the application-to-JDBC Manager connection. **JDBC Driver API**, this supports the JDBC Manager-to-Driver Connection. The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.



The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases. Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –

Common JDBC Components

The JDBC API provides the following interfaces and classes −

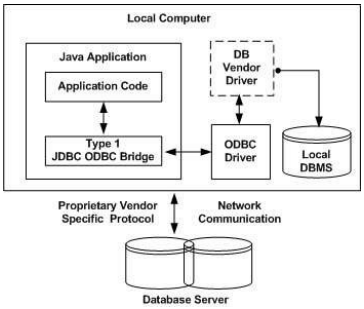
- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub-protocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Types of JDBC Drivers

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java. The Java.sql package that ships with JDK, contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver.

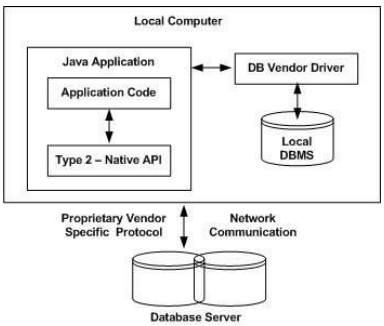
JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below –

Type 1: JDBC-ODBC Bridge Driver: In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.



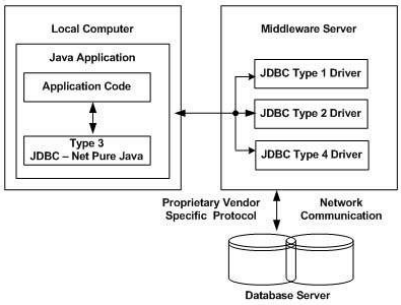
When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API: In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.



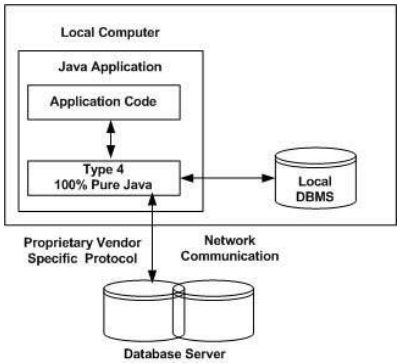
If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java: In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.



This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases. You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java: In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.



This kind of driver is extremely flexible; you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically. MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

JDBC API for Database Connectivity (java.sql package)

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

java.sql package provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language. This API includes a framework whereby different drivers can

be installed dynamically to access different data sources. Although the JDBC API is mainly geared to passing SQL statements to a database, it provides for reading and writing data from any data source with a tabular format. The reader/writer facility, available through the javax.sql.RowSet group of interfaces, can be customized to use and update data from a spread sheet, flat file, or any other tabular data source.

Once a connection is obtained we can interact with the database. The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database. 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.

Statement: Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example –

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object. A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   stmt.close();
}
```

The PreparedStatement Objects: The PreparedStatement interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object. This statement gives you the flexibility of supplying arguments dynamically. Creating PreparedStatement Object:
PreparedStatement pstmt = null;

```
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement. The setXXX() methods bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException. Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0. All of the Statement object's methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object. A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

The CallableStatement Objects: Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure. The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object based on the preceding stored procedure −

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders. Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException. If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding. When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return. Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type. Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

ResultSetMetaData

Java ResultSetMetaData Interface The metadata means data about data i.e. we can get further information from the data. If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object. Commonly used methods of ResultSetMetaData interface:

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.

public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

DatabaseMetaData

This interface is implemented by driver vendors to let users know the capabilities of a Database Management System (DBMS) in combination with the driver based on JDBCTM technology ("JDBC driver") that is used with it. Different relational DBMSs often support different features, implement features in different ways, and use different data types. In addition, a driver may implement a feature on top of what the DBMS offers. Information returned by methods in this interface applies to the capabilities of a particular driver and a particular DBMS working together. Note that as used in this documentation, the term "database" is used generically to refer to both the driver and DBMS.

A user for this interface is commonly a tool that needs to discover how to deal with the underlying DBMS. This is especially true for applications that are intended to be used with more than one DBMS. For example, a tool might use the method getTypeInfo to find out what data types can be used in a CREATE TABLE statement. Or a user might call the method supportsCorrelatedSubqueries to see if it is possible to use a correlated subquery or supportsBatchUpdates to see if it is possible to use batch updates. Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException:** it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException:** it returns the product name of the database.
- **public String getDatabaseProductVersion()throws SQLException:** it returns the product version of the database.
- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException:** it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

Other JDBC APIs

ResultSet: The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The java.sql.ResultSet interface represents the result set of a database query. A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object. The methods of the ResultSet interface can be broken down into three categories:

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created. JDBC provides the following connection methods to create statements with desired ResultSet:

- createStatement(int RSType, int RSConcurrency);
- prepareStatement(String SQL, int RSType, int RSConcurrency);
- prepareCall(String sql, int RSType, int RSConcurrency);

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable. The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object:

```
try {
   Statement stmt = conn.createStatement(
                            ResultSet.TYPE_FORWARD_ONLY,
                            ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
   ....
}
finally {
   ....
}
```

Commonly used methods of ResultSet interface:

public boolean next():	is used to move the cursor to the one row next from the current position.
public boolean previous():	is used to move the cursor to the one row previous from the current position.
public boolean first():	is used to move the cursor to the first row in result set object.
public boolean last():	is used to move the cursor to the last row in result set object.
public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.

public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

SQLException: Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion. When an exception condition occurs, an exception is thrown. The term thrown means that current program execution stops, and the control is redirected to the nearest applicable catch clause. If no applicable catch clause exists, then the program's execution ends. JDBC Exception handling is very similar to the Java Exception handling but for JDBC, the most common exception you'll deal with is java.sql.SQLException. An SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause. The passed SQLException object has the following methods available for retrieving additional information about the exception:

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and it's backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and it's backtrace to the print writer you specify.

RowSet: The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5. It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use. The implementation classes of RowSet interface are as: JdbcRowSet, CachedRowSet, WebRowSet, JoinRowSet, FilteredRowSet.

Connecting with Databases (MySQL, Access, Oracle) Practicals:

Unit – 2

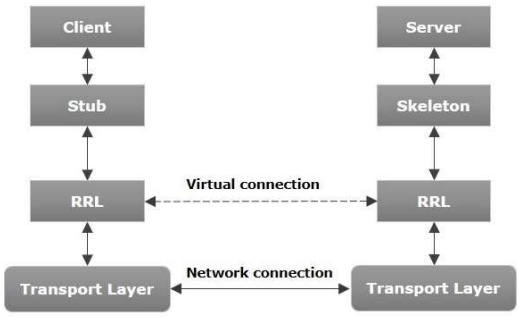
RMI, Servlet

RMI overview

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

RMI architecture

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client). Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry). The client program requests the remote objects on the server and tries to invoke its methods. The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture. Transport Layer − This layer connects the client and the server. It manages the existing connection and also sets up new connections. Stub − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program. Skeleton − This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object. RRL(Remote Reference Layer) − It is the layer which manages the references made by the client to the remote object.

Stub and Skeleton

A remote object is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

Stub: The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the tasks like: It initiates a connection with remote Virtual Machine (JVM), It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM), It waits for the result It reads (unmarshals) the return value or exception, and It finally, returns the value to the caller.

Skeleton: The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the task like: It reads the parameter for the remote method, It invokes the method on the actual remote object, and It writes and transmits (marshals) the result to the caller. In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.

Developing and Executing RMI application (Practical)

Servlet Introduction

What is a web application?

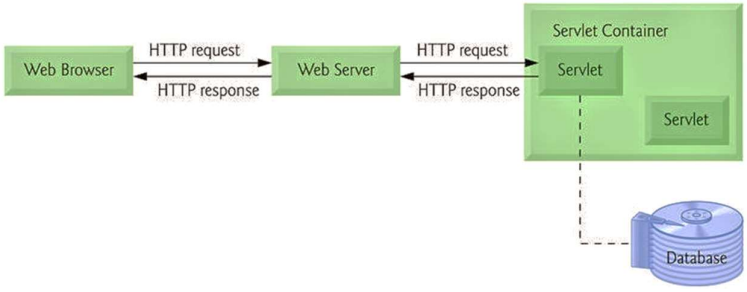
A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter, etc. and other elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request. CGI (Common Gateway Interface) technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

Servlets provide a component-based, platform-independent method for building Web based applications, without the performance limitations of CGI programs. Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. This tutorial will teach you how to use Java Servlets to develop your web based applications in simple and easy steps.

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page). Servlet technology is robust and scalable because of java language. Before Servlet, CGI scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below. There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc. Servlet can be described in many ways, 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 including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.

Architecture of a Servlet



Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server. Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically. Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI. The diagram shows the position of Servlets in a Web Application.

Servlets perform the following major tasks:

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlet API (Javax.servlet and Javax.servlet.http)

Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification. Servlets can be created using the javax.servlet and javax.servlet.http packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.

These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1. Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet API. The javax.servlet package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol. The javax.servlet.http package contains interfaces and classes that are responsible for http requests only. Let's see what are the interfaces of javax.servlet package.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

- Servlet
- ServletRequest
- ServletResponse
- RequestDispatcher
- ServletConfig
- ServletContext
- SingleThreadModel
- Filter
- FilterConfig
- FilterChain
- ServletRequestListener
- ServletRequestAttributeListener
- ServletContextListener
- ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

- GenericServlet
- ServletInputStream
- ServletOutputStream
- ServletRequestWrapper
- ServletResponseWrapper
- ServletRequestEvent
- ServletContextEvent
- ServletRequestAttributeEvent
- ServletContextAttributeEvent
- ServletException
- UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

- HttpServletRequest
- HttpServletResponse
- HttpSession
- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionBindingListener
- HttpSessionActivationListener
- HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

- HttpServlet
- Cookie
- HttpServletRequestWrapper
- HttpServletResponseWrapper

- HttpSessionEvent
- HttpSessionBindingEvent
- HttpUtils (deprecated now)

Servlet Interface: Servlet interface provides common behaviour to all the servlets. Servlet interface defines methods that all servlets must implement. Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods. There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

Method	Description
public void init(ServletConfig config)	initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once.
public void service(ServletRequest request,ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about servlet such as writer, copyright, version etc.

GenericServlet class: GenericServlet class implements Servlet, ServletConfig and Serializable interfaces. It provides the implementation of all the methods of these interfaces except the service method. GenericServlet class can handle any type of request so it is protocol-independent. You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

HttpServlet class: The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

Servlet Life Cycle

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

- The servlet is initialized by calling the init() method.
- The servlet calls service() method to process a client's request.
- The servlet is terminated by calling the destroy() method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in detail.

The init() Method: The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet. The init method definition looks like:

```
public void init() throws ServletException {
```

```
    // Initialization code...
}
```

The service() Method: The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client. Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate. Here is the signature of this method:

```
public void service(ServletRequest request, ServletResponse response)
   throws ServletException, IOException {
}
```

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client. The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method: A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The doPost() Method: A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The destroy() Method: The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such clean-up activities. After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy() {
   // Finalization code...
}
```

Developing and Deploying Servlets (Practical)

Servlet Requests

When a browser requests for a web page, it sends lot of information to the web server which cannot be read directly because this information travel as a part of header of HTTP request. You can check HTTP Protocol for more information on this. An object of ServletRequest is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header information, attributes etc. There are many methods defined in the ServletRequest interface. Some of them are as follows:

Method	Description
public String getParameter(String name)	is used to obtain the value of a parameter by name.
public String[] getParameterValues(String name)	returns an array of String containing all values of given parameter name. It is mainly used to obtain values of a Multi select list box.
java.util.Enumeration getParameterNames()	returns an enumeration of all of the request parameter names.

public int getContentLength()	Returns the size of the request entity data, or -1 if not known.
public String getCharacterEncoding()	Returns the character set encoding for the input of this request.
public String getContentType()	Returns the Internet Media Type of the request entity data, or null if not known.
public ServletInputStream getInputStream() throws IOException	Returns an input stream for reading binary data in the request body.
public abstract String getServerName()	Returns the host name of the server that received the request.
public int getServerPort()	Returns the port number on which this request was received.

Servlet Responses

When a Web server responds to an HTTP request, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
   (Blank Line)
<!doctype ...>
<html>
   <head>...</head>
   <body>
      ...
   </body>
</html>
```

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example). Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side. There are following methods which can be used to set HTTP response header in your servlet program. These methods are available with HttpServletResponse object:

Method & Description
String encodeRedirectURL(String url): Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
String encodeURL(String url): Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
boolean containsHeader(String name): Returns a Boolean indicating whether the named response header has already been set.
boolean isCommitted(): Returns a Boolean indicating if the response has been committed.
void addCookie(Cookie cookie): Adds the specified cookie to the response.

void addDateHeader(String name, long date): Adds a response header with the given name and date-value.
void addHeader(String name, String value): Adds a response header with the given name and value.
void addIntHeader(String name, int value): Adds a response header with the given name and integer value.
void flushBuffer(): Forces any content in the buffer to be written to the client.
void reset(): Clears any data that exists in the buffer as well as the status code and headers.
void resetBuffer(): Clears the content of the underlying buffer in the response without clearing headers or status code.
void sendError(int sc): Sends an error response to the client using the specified status code and clearing the buffer.
void sendError(int sc, String msg): Sends an error response to the client using the specified status.
void sendRedirect(String location) Sends a temporary redirect response to the client using the specified redirect location URL.
void setBufferSize(int size): Sets the preferred buffer size for the body of the response.
void setCharacterEncoding(String charset): Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
void setContentLength(int len): Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
void setContentType(String type): Sets the content type of the response being sent to the client, if the response has not been committed yet.
void setDateHeader(String name, long date): Sets a response header with the given name and date-value.
void setHeader(String name, String value): Sets a response header with the given name and value.
void setIntHeader(String name, int value): Sets a response header with the given name and integer value
void setLocale(Locale loc): Sets the locale of the response, if the response has not been committed yet.
void setStatus(int sc): Sets the status code for this response

Reading Initialization Parameters

Session Tracking Approaches

Session simply means a particular interval of time. Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet. Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request. Still there are following three ways to maintain session between web client and web server:

URL Rewriting: You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. For example, with http://tutorialspoint.com/file.htm;sessionid = 12345, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and it works even when browsers don't support cookies. The drawback of URL re-writing is that you would have to generate every URL dynamically to assign a session ID, even in case of a simple static HTML page.

Advantage of URL Rewriting

- It will always work whether cookie is disabled or not (browser independent).
- Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

- It will work only with links.
- It can send only textual information.

Hidden Form Fields: A web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers. This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

Advantage of Hidden Form Field

- It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

- It is maintained at server side.
- Extra form submission is required on each pages.
- Only textual information can be used.

Cookies: A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie. This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Advantage of Cookies

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

Disadvantage of Cookies

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

HttpSession: Apart from the above mentioned three ways, servlet provides HttpSession Interface which 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. The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. You would get HttpSession object by calling the public method getSession() of HttpServletRequest, as below:

```
HttpSession session = request.getSession();
```

You need to call request.getSession() before you send any document content to the client.

Servlet Collaboration

RequestDispatcher in Servlet: The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration. There are two methods defined in the RequestDispatcher interface, they are:

- **public void forward(ServletRequest request,ServletResponse response) throws ServletException, java.io.IOException:** Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
- **public void include(ServletRequest request,ServletResponse response) throws ServletException, java.io.IOException:** Includes the content of a resource (servlet, JSP page, or HTML file) in the response.

SendRedirect in servlet: The sendRedirect() method of HttpServletResponse interface can be used to redirect response to another resource, it may be servlet, jsp or html file. It accepts relative as well as absolute URL. It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server. There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:

forward() method	sendRedirect() method
The forward() method works at server side.	The sendRedirect() method works at client side.
It sends the same request and response objects to another servlet.	It always sends a new request.
It can work within the server only.	It can be used within and outside the server.
Example: request.getRequestDispacher("servlet2") .forward(request,response);	Example: response.sendRedirect("servlet2");

Servlet with JDBC (Practical)

Unit-3

JSP, Java Beans

Introduction to JSP and JSP Basics

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc. A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>. A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands. Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically. JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

Advantages of JSP vs. Servlet

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

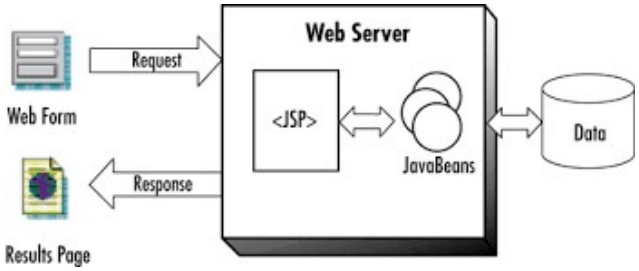
Extension to Servlet: JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP that makes JSP development easy.

Easy to maintain: JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

Fast Development: No need to recompile and redeploy. If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

Less code than Servlet: In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

JSP Architecture



The web server needs a JSP engine, i.e, a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development. A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs. The diagram shows the position of JSP container and JSP files in a Web application. The following steps explain how the web server creates the Webpage using JSP:

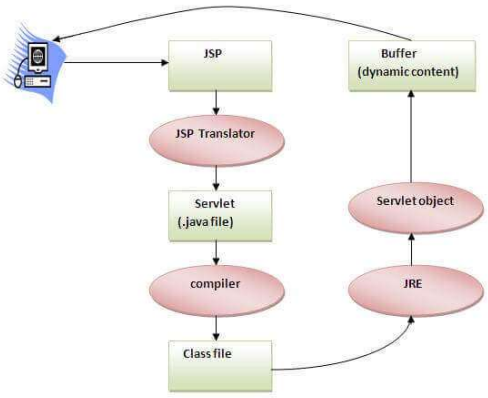
- As with a normal page, your browser sends an HTTP request to the web server.
- The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with .jsp instead of .html.
- The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behaviour of the page.
- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.

- A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is further passed on to the web server by the servlet engine inside an HTTP response.
- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster. So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

Life cycle of JSP

The key to understanding the low-level functionality of JSP is to understand the simple life cycle they follow. A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet. The following are the paths followed by a JSP:



- Translation of JSP Page
- Compilation of JSP Page
- Classloading (the classloader loads class file)
- Instantiation (Object of the Generated Servlet is created).
- Initialization (the container invokes jspInit() method).
- Request processing (the container invokes _jspService() method).
- Destroy (the container invokes jspDestroy() method).

JSP page is translated into Servlet by the help of JSP translator. The JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later like initialization, committing response to the browser and destroy.

JSP Elements: Directive Elements, Scripting Elements, Action Elements

Directives Elements

These directives provide directions and instructions to the container, telling it how to handle certain aspects of the JSP processing. A JSP directive affects the overall structure of the servlet class. It usually has the following form:

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

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas. The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional. There are three types of directive tag:

- **<%@ page ... %>:** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
- **<%@ include ... %>:** Includes a file during the translation phase.
- **<%@ taglib ... %>:** Declares a tag library, containing custom actions, used in the page

JSP - The page Directive: The page directive is used to provide instructions to the container. These instructions pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page. Following is the basic syntax of the page directive:

```
<%@ page attribute = "value" %>
```
You can write the XML equivalent of the above syntax as follows:
```
<jsp:directive.page attribute = "value" />
```
Following table lists out the attributes associated with the page directive:

S.No.	Attribute & Purpose
1	**Buffer:** Specifies a buffering model for the output stream.
2	**autoFlush:** Controls the behavior of the servlet output buffer.
3	**contentType:** Defines the character encoding scheme.
4	**errorPage:** Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	**isErrorPage:** Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	**Extends:** Specifies a superclass that the generated servlet must extend.
7	**Import:** Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
8	**Info:** Defines a string that can be accessed with the servlet's **getServletInfo()** method.
9	**isThreadSafe:** Defines the threading model for the generated servlet.
10	**Language:** Defines the programming language used in the JSP page.
11	**Session:** Specifies whether or not the JSP page participates in HTTP sessions
12	**isELIgnored:** Specifies whether or not the EL expression within the JSP page will be ignored.
13	**isScriptingEnabled:** Determines if the scripting elements are allowed for use.

The include Directive: The include directive is used to include a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code the include directives anywhere in your JSP page. The general usage form of this directive is as follows:
```
<%@ include file = "relative url" >
```
The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP. You can write the XML equivalent of the above syntax as follows:
```
<jsp:directive.include file = "relative url" />
```
For more details related to include directive, check the Include Directive.

The taglib Directive: The JavaServer Pages API allow you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior. The taglib directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in your JSP page. The taglib directive follows the syntax given below:
```
<%@ taglib uri="uri" prefix = "prefixOfTag" >
```

Here, the uri attribute value resolves to a location the container understands and the prefix attribute informs a container what bits of markup are custom actions. You can write the XML equivalent of the above syntax as follows:

```
<jsp:directive.taglib uri = "uri" prefix = "prefixOfTag" />
```

Scripting Elements

In JSP, java code can be written inside the jsp page using the scriptlet tag. Let's see what are the scripting elements first:

Declaration: The JSP declaration tag is used to declare fields and methods. The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet. So it doesn't get memory at each request. The syntax of the declaration tag is as follows:

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

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

```
<html>
      <body>
            <%! int data=50; %>
            <%= "Value of the variable is:"+data %>
      </body>
</html>
```

Scriptlet: A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

```
<%  java source code %>
```

In this example, we are displaying a welcome message.

```
<html>
      <body>
            <% out.print("welcome to jsp"); %>
      </body>
</html>
```

Expression: The code placed within JSP expression tag is written to the output stream of the response. So you need not write out.print() to write data. It is mainly used to print the values of variable or method.

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

In this example of jsp expression tag, we are simply displaying a welcome message.

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

Action Elements

These actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin. There is only one syntax for the Action element, as it conforms to the XML standard:

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

Action elements are basically predefined functions. The following table lists out the available JSP actions:

Syntax & Purpose
jsp:include: Includes a file at the time the page is requested.
jsp:param: sets the parameter value. It is used in forward and include mostly.
jsp:useBean: Finds or instantiates a JavaBean.
jsp:setProperty: Sets the property of a JavaBean.
jsp:getProperty: Inserts the property of a JavaBean into the output.
jsp:forward: Forwards the requester to a new page.

jsp:plugin: Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
jsp:element: Defines XML elements dynamically.
jsp:attribute: Defines dynamically-defined XML element's attribute.
jsp:body: Defines dynamically-defined XML element's body.
jsp:text: Used to write template text in JSP pages and documents.

JSP:param:

JSP:include: The jsp:include action tag is used to include the content of another resource it may be jsp, html or servlet. The jsp include action tag includes the resource at request time so it is better for dynamic pages because there might be changes in future. The jsp:include tag can be used to include static as well as dynamic pages. Syntax of jsp:include action tag without parameter

```
<jsp:include page="relativeURL | <%= expression %>" />
```

Syntax of jsp:include action tag with parameter

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

JSP:Forward: The jsp:forward action tag is used to forward the request to another resource it may be jsp, html or another resource. Syntax of jsp:forward action tag without parameter

```
<jsp:forward page="relativeURL | <%= expression %>" />
```

Syntax of jsp:forward action tag with parameter:

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

JSP:plugin: The plugin action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed. If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean. The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean. Following is the typical syntax of using the plugin action:

```
<jsp:plugin type = "applet" codebase = "dirname" code = "MyApplet.class" width = "60" height = "80">
   <jsp:param name = "fontcolor" value = "red" />
   <jsp:param name = "background" value = "black" />
   <jsp:fallback>
      Unable to initialize Java Plugin
   </jsp:fallback>
</jsp:plugin>
```

JSP Implicit Objects

There are 9 jsp implicit objects. These objects are created by the web container that are available to all the jsp pages. The available implicit objects are out, request, config, session, application etc. These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables. Following table lists out the nine Implicit Objects that JSP supports:

Object & Description
request: This is the **HttpServletRequest** object associated with the request.

response: This is the **HttpServletResponse** object associated with the response to the client.
out: This is the **PrintWriter** object used to send output to the client.
session: This is the **HttpSession** object associated with the request.
application: This is the **ServletContext** object associated with the application context.
config: This is the **ServletConfig** object associated with the page.
pageContext: This encapsulates use of server-specific features like higher performance **JspWriters**.
page: This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
Exception: The **Exception** object allows the exception data to be accessed by designated JSP.

The request Object: The request object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page the JSP engine creates a new object to represent that request. The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc.

The response Object: The response object is an instance of a javax.servlet.http.HttpServletResponse object. Just as the server creates the request object, it also creates an object to represent the response to the client. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

The out Object: The out implicit object is an instance of a javax.servlet.jsp.JspWriter object and is used to send content in a response. The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the buffered = 'false' attribute of the page directive. The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions. Following table lists out the important methods that we will use to write boolean char, int, double, object, String, etc.

Method & Description

- out.print(dataType dt): Print a data type value
- out.println(dataType dt): Print a data type value then terminate the line with new line character.
- out.flush(): Flush the stream.

The session Object: The session object is an instance of javax.servlet.http.HttpSession and behaves exactly the same way that session objects behave under Java Servlets. The session object is used to track client session between client requests.

The application Object: The application object is direct wrapper around the ServletContext object for the generated Servlet and in reality an instance of a javax.servlet.ServletContext object. This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method. By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

The config Object: The config object is an instantiation of javax.servlet.ServletConfig and is a direct wrapper around the ServletConfig object for the generated servlet. This object allows the JSP programmer access to

the Servlet or JSP engine initialization parameters such as the paths or file locations etc. The following config method is the only one you might ever use, and its usage is trivial:

```
config.getServletName();
```

This returns the servlet name, which is the string contained in the <servlet-name> element defined in the WEB-INF\web.xml file.

The pageContext Object: The pageContext object is an instance of a javax.servlet.jsp.PageContext object. The pageContext object is used to represent the entire JSP page. This object is intended as a means to access information about the page while avoiding most of the implementation details. This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object. The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope. The PageContext class defines several fields, including PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, and APPLICATION_SCOPE, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the javax.servlet.jsp.JspContext class. One of the important methods is removeAttribute. This method accepts either one or two arguments. For example, pageContext.removeAttribute ("attrName") removes the attribute from all scopes, while the following code only removes it from the page scope:

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

The page Object: This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The page object is really a direct synonym for the this object.

The exception Object: The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

JSP Scope

JSP provides very useful features. One of them is maintaining the user defined variable. As you all know that in JAVA, each variable has a scope. Scope decides the accessibility of an object or variable. For example, some variables are accessible within for loop, if-else block or within specific method or class or package. Same way, in JSP some variables needs to have different scopes than others. JSP provides the capability to user to define the scope of these variables. JSP provides 4 scopes to a variable. Developer can assign any one of them to a variable.

Page Scope: Page Scope makes variable available to the developer for the current page only. Once the current page is closed by user or forwarded internally by application or redirected by application, than the variables having page scope will not be accessible on next page.

Request Scope: Request Scope makes variable available to the developer for the current request only. Once the current request is over, than the variables having request scope will not be accessible on next request. Single request may include multiple pages using forward.

Session Scope: Session Scope makes variable available to the developer for the current session only. Once the current session is over or timed out, than the variables having session scope will not be accessible on next session.

Application Scope: Application Scope makes variable available to the developer for the full application. It remains available till application is running on server.

Including and Forwarding from JSP Pages (Practical)

Working with Session and Cookies in JSP

HTTP is a "stateless" protocol which means each time a client retrieves a Webpage, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request. Let us now discuss a few options to maintain the session between the Web Client and the Web Server:

Cookies: A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie. This may not be an effective way as the browser at times does not support a cookie. It is not recommended to use this procedure to maintain the sessions.

Hidden Form Fields: A web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or the POST data. Each time the web browser sends the request back, the session_id value can be used to keep the track of different web browsers. This can be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting: You can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session. For example, with http://tutorialspoint.com/file.htm;sessionid=12345, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client. URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies. The drawback here is that you will have to generate every URL dynamically to assign a session ID though page is a simple static HTML page.

The session Object: Apart from the above mentioned options, JSP makes use of the servlet provided HttpSession Interface. This interface provides a way to identify a user across.

- a one page request or
- visit to a website or
- store information about that user

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

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

The JSP engine exposes the HttpSession object to the JSP author through the implicit session object. Since session object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().

Error Handling and Exception Handling with JSP

When you are writing a JSP code, you might make coding errors which can occur at any part of the code. There may occur the following type of errors in your JSP code:

Checked exceptions: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

Runtime exceptions: A runtime exception is an exception that probably could have been avoided by the programmer. As opposed to the checked exceptions, runtime exceptions are ignored at the time of compilation.

Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation. We will further discuss ways to handle run time exception/error occuring in your JSP code.

Using Exception Object: The exception object is an instance of a subclass of Throwable (e.g., java.lang. NullPointerException) and is only available in error pages. Following table lists out the important methods available in the Throwable class.

- public String getMessage(): Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
- public Throwable getCause(): Returns the cause of the exception as represented by a Throwable object.
- public String toString(): Returns the name of the class concatenated with the result of getMessage().
- public void printStackTrace(): Prints the result of toString() along with the stack trace to System.err, the error output stream.
- public StackTraceElement [] getStackTrace(): Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
- public Throwable fillInStackTrace(): Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

JSP gives you an option to specify Error Page for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

JDBC with JSP (Practical)

Introduction to JavaBean

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications. Following are the unique characteristics that distinguish a JavaBean from other Java classes:

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the Serializable interface.
- It may have a number of properties which can be read or written.
- It may have a number of "getter" and "setter" methods for the properties.

Advantages of JavaBean:

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages of JavaBean:

- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

JavaBean Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define. A JavaBean property may be read, write, read only, or write only. JavaBean properties are accessed through two methods in the JavaBean's implementation class:

- getPropertyName(): For example, if property name is firstName, your method name would be getFirstName() to read that property. This method is called accessor.
- setPropertyName(): For example, if property name is firstName, your method name would be setFirstName() to write that property. This method is called mutator.

A read-only attribute will have only a getPropertyName() method, and a write-only attribute will have only a setPropertyName() method. The useBean action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a page, request, session or application based on your requirement. The value of the id attribute may be any value as a long as it is a unique name among other useBean declarations in the same JSP.

JavaBean Methods

Common JavaBean packaging

Unit-4

MVC Architecture, EJB, Hibernate

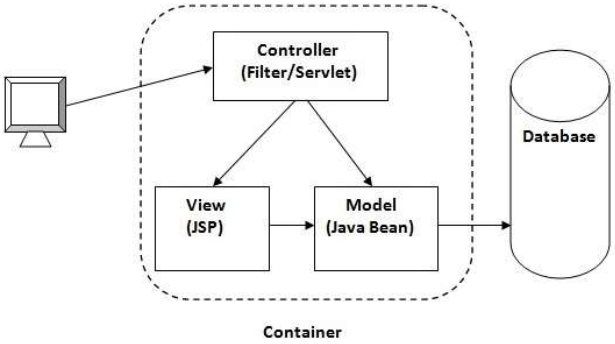
Introduction to MVC

MVC stands for Model View and Controller. It is a design pattern that separates the business logic, presentation logic and data. Controller acts as an interface between View and Model. Controller intercepts all the incoming requests. Model represents the state of the application i.e. data. It can also have business logic. View represents the presentation i.e. UI (User Interface).

Advantage of MVC Architecture:

- Navigation Control is centralized
- Easy to maintain the large application

Implementation of MVC Architecture



Servlet and JSP are the main technologies to develop the web applications. Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet. Problem in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.

As you can see in the figure, there is picture which show the flow of the model1 architecture.

- Browser sends request for the JSP page
- JSP accesses Java Bean and invokes business logic
- Java Bean connects to the database and get/save data
- Response is sent to the browser which is generated by JSP

Advantage of Model Architecture

- Easy and Quick to develop web application

Disadvantage of Model Architecture

- Navigation control is decentralized since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- Time consuming you need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- Hard to extend It is better for small applications but not for large applications.

Introduction to EJB

EJB is an acronym for enterprise java bean. It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications. To get information about distributed applications, visit RMI Tutorial first. To run EJB application, you need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

- life cycle management,
- security,
- transaction management, and
- object pooling.

EJB application is deployed on the server, so it is called server side component also. EJB is like COM (Component Object Model) provided by Microsoft. But, it is different from Java Bean, RMI and Web Services.

Benefits of EJB

Following are the important benefits of EJB:

- Simplified development of large-scale enterprise level application.
- Application Server/EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling, and so on. Developer has to focus only on business logic of the application.
- EJB container manages life cycle of EJB instances, thus developer needs not to worry about when to create/delete EJB objects.

Restriction on EJB

Contested some restrictions on EJBs are contested by many EJB developers and container vendors. For example, EJBs are prohibited from reading or writing files through the local filesystem. The EJB Specification merely says that file operations are "not well suited" to business data processing. Others have argued that it should be up to the EJB developer to decide what programming techniques are suited to the application

Types of EJB

EJB stands for Enterprise Java Beans. EJB is an essential part of a J2EE platform. J2EE platform has component based architecture to provide multi-tiered, distributed and highly transactional features to enterprise level applications. EJB provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability, and high performance. An EJB application can be deployed on any of the application server compliant with the J2EE 1.3 standard specification. EJB is primarily divided into three categories; following table lists their names with brief descriptions:

Session Bean: Session bean stores data of a particular user for a single session. It can be stateful or stateless. It is less resource intensive as compared to entity bean. Session bean gets destroyed as soon as user session terminates. Session bean contains business logic that can be invoked by local, remote or webservice client.

Entity Bean: Entity beans represent persistent data storage. User data can be saved to database via entity beans and later on can be retrieved from the database in the entity bean. EJB 3.0, entity bean used in EJB 2.0 is largely replaced by persistence mechanism. Now entity bean is a simple **POJO (Plain Old Java Object)** having mapping with table. It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

Message Driven Bean: Message driven beans are used in context of JMS (Java Messaging Service). Message Driven Beans can consumes JMS messages from external entities and act accordingly. A message driven bean is a type of enterprise bean, which is invoked by EJB container when it receives a message from queue or topic. Message driven bean is a stateless bean and is used to do task asynchronously. Like Session Bean, it contains the business logic but it is invoked by passing message.

Timer service

Timer Service is a mechanism by which scheduled application can be build. For example, salary slip generation on the 1st of every month. EJB 3.0 specification has specified @Timeout annotation, which helps in programming the EJB service in a stateless or message driven bean. EJB Container calls the method, which is annotated by @Timeout. EJB Timer Service is a service provided by EJB container, which helps to create timer and to schedule callback when timer expires.

Introduction to Hibernate

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application. Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Need for hibernate

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence. An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.

Features of hibernate:

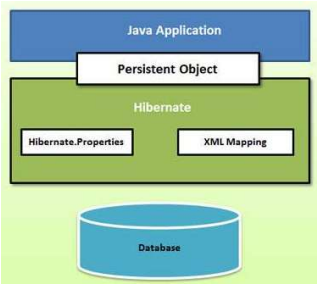
- Open Source and Lightweight: Hibernate framework is open source under the LGPL license and lightweight.
- Fast Performance: The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.
- Database Independent Query: HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.
- 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.
- Simplifies Complex Join: Fetching data from multiple tables is easy in hibernate framework.
- Provides Query Statistics and Database Status: Hibernate supports Query cache and provide statistics about query and database status.

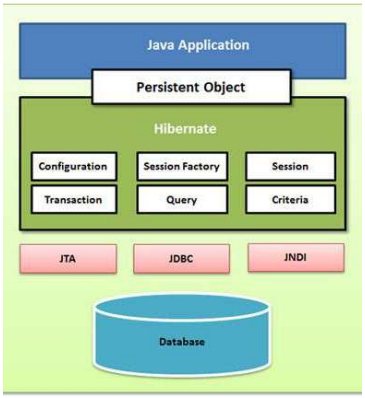
Disadvantages of Hibernate:

- Performance Cost: Hibernate generates lot of SQL statements in runtime based on our mapping, so it is bit slower than JDBC. If we use JDBC, then we directly write our queries, so no extra effort is required.
- Does not allow multiple inserts: Hibernate does not allow some queries which are supported by JDBC. For example, it does not allow to insert multiple objects (persistent data) to same table using single query. Developer has to write separate query to insert each object.
- More Complex with joins: If there are lot of mappings and joins among the tables, then code becomes bit complex to understand as we need to define the mapping and join information in the XML file.
- Poor performance in Batch processing: It advisable to use pure JDBC for batch processing as Hibernate performance is not good in Batch processing.
- Not good for small project: Small project will have less number of tables, introducing entire Hibernate framework will be overhead than useful.

Exploring Hibernate Architecture

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application. Following is a very high level view of the Hibernate Application Architecture.





Here is a detailed view of the Hibernate Application Architecture with its important core classes. Hibernate uses various existing Java APIs, like JDBC, Java Transaction API (JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common

to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Downloading and Configuring and necessary files to Hibernate in Eclipse, Jars files of hibernate. (Practical)

Hibernate Configuration file
Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file named hibernate.cfg.xml. I will consider XML formatted file hibernate.cfg.xml to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Mapping file
An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate — how to map the defined class or classes to the database tables? Though many Hibernate users choose to write the XML by hand, but a number of tools exist to generate the mapping document. These include XDoclet, Middlegen and AndroMDA for the advanced Hibernate users.

Basic Example of Hibernate Annotation (Practical)

Hibernate Inheritance
We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate: Table Per Hierarchy, Table Per Concrete class
Table Per Subclass.

- **Table Per Hierarchy:** In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table.
- **Table Per Concrete class:** In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.
- **Table Per Subclass:** In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

Hibernate Annotations
Hibernate uses XML mapping file for the transformation of data from POJO to database tables and vice versa. Hibernate annotations are the newest way to define mappings without the use of XML file. You can use annotations in addition to or as a replacement of XML mapping metadata. Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

If you going to make your application portable to other EJB 3 compliant ORM applications, you must use annotations to represent the mapping information, but still if you want greater flexibility, then you should go with XML-based mappings.

Hibernate Sessions
A Session is used to get a physical connection with a database. 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. 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 them as needed. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:

- transient − A new instance of a persistent class, which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.

- persistent − You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
- detached − Once we close the Hibernate Session, the persistent instance will become a detached instance.

A Session instance is serializable if its persistent classes are serializable.

Unit – 5

Spring, Struts

Introduction of Spring Framework

Spring 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. Spring framework is an open source Java platform. It was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of Spring framework is around 2MB. The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.

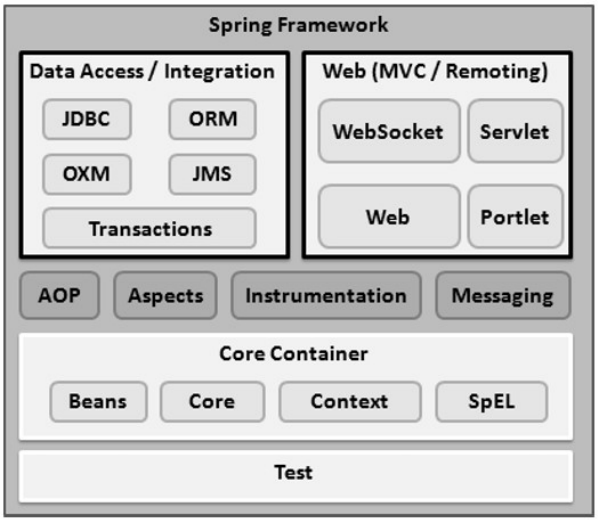
Benefits of Using the Spring Framework:

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.

- Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

Spring Architecture

Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework. The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container: The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are: The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features. The Bean module provides BeanFactory, which is a sophisticated implementation of the factory pattern. The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The

ApplicationContext interface is the focal point of the Context module. The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration: The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as: The JDBC module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding. The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis. The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream. The Java Messaging Service JMS module contains features for producing and consuming messages. The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web: The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are: The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. The Web-MVC module contains Spring's Model-View-Controller (MVC) implementation for web applications. The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications. The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous: There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are: The AOP module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. The Aspects module provides integration with AspectJ, which is again a powerful and mature AOP framework. The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers. The Messaging module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients. The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

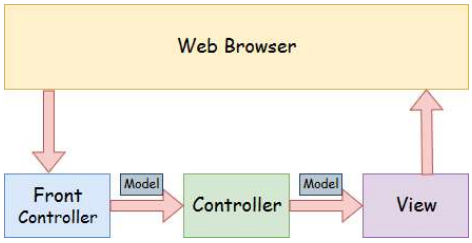
Spring Framework definition

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Spring & MVC

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection. A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet. Here, DispatcherServlet is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.



Model - A model contains the data of the application. A data can be a single object or a collection of objects. Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller. View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker. Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Spring Context definition
These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment. In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Inversion of Control (IoC) in Spring
The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC. Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework,

Aspect Oriented programming in Spring (AOP)
One of the key components of Spring is the Aspect Oriented Programming (AOP) framework. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect. The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing you to define method-interceptors and point cuts to cleanly decouple code that implements functionality that should be separated.

Introduction to Struts
The struts 2 framework is used to develop MVC-based web application. The struts framework was initially created by Craig McClanahan and donated to Apache Foundation in May, 2000 and Struts 1.0 was released in June 2001. The current stable release of Struts is Struts 2.3.16.1 in March 2, 2014.

Struts2 is a popular and mature web application framework based on the MVC design pattern. Struts2 is not just a new version of Struts 1, but it is a complete rewrite of the Struts architecture. The Webwork framework initially started with Struts framework as the basis and its goal was to offer an enhanced and improved framework built on Struts to make web development easier for the developers. After a while, the Webwork framework and the Struts community joined hands to create the famous Struts2 framework.

Struts 2 Framework Features:

- POJO Forms and POJO Actions − Struts2 has done away with the Action Forms that were an integral part of the Struts framework. With Struts2, you can use any POJO to receive the form input. Similarly, you can now see any POJO as an Action class.
- Tag Support − Struts2 has improved the form tags and the new tags which allow the developers to write less code.
- AJAX Support − Struts2 has recognized the take over by Web2.0 technologies, and has integrated AJAX support into the product by creating AJAX tags, this function is very similar to the standard Struts2 tags.
- Easy Integration − Integration with other frameworks like Spring, Tiles and SiteMesh is now easier with a variety of integration available with Struts2.

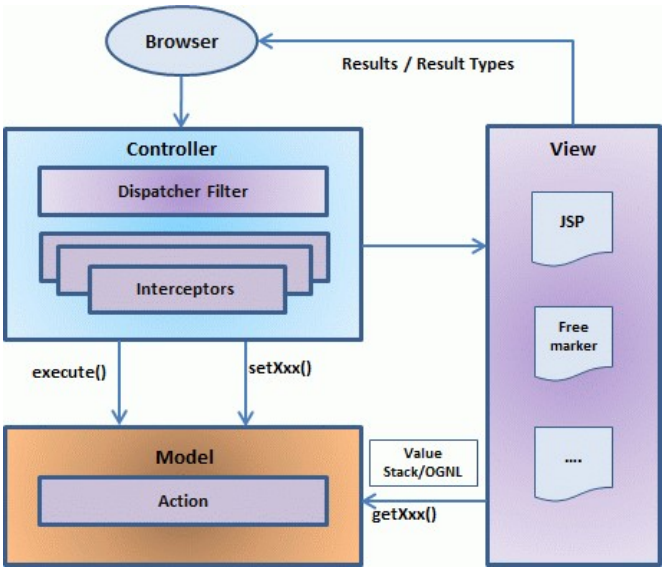
- Template Support − Support for generating views using templates.
- Plugin Support − The core Struts2 behavior can be enhanced and augmented by the use of plugins. A number of plugins are available for Struts2.
- Profiling − Struts2 offers integrated profiling to debug and profile the application. In addition to this, Struts also offers integrated debugging with the help of built in debugging tools.
- Easy to Modify Tags − Tag markups in Struts2 can be tweaked using Freemarker templates. This does not require JSP or java knowledge. Basic HTML, XML and CSS knowledge is enough to modify the tags.
- Promote Less configuration − Struts2 promotes less configuration with the help of using default values for various settings. You don't have to configure something unless it deviates from the default settings set by Struts2.
- View Technologies − Struts2 has a great support for multiple view options (JSP, Freemarker, Velocity and XSLT)
- Listed above are the Top 10 features of Struts 2 which makes it as an Enterprise ready framework.

Struts 2 Disadvantages:
Though Struts 2 comes with a list of great features, there are some limitations of the current version - Struts 2 which needs further improvement. Listed are some of the main points −

- Bigger Learning Curve − To use MVC with Struts, you have to be comfortable with the standard JSP, Servlet APIs and a large & elaborate framework.
- Poor Documentation − Compared to the standard servlet and JSP APIs, Struts has fewer online resources, and many first-time users find the online Apache documentation confusing and poorly organized.
- Less Transparent − With Struts applications, there is a lot more going on behind the scenes than with normal Java-based Web applications which makes it difficult to understand the framework.

Understanding Struts Framework

From a high level, Struts2 is a pull-MVC (or MVC2) framework. The Model-ViewController pattern in Struts2 is implemented with the following five core components: Actions, Interceptors, Value Stack / OGNL, Results / Result types, View technologies. Struts 2 is slightly different from a traditional MVC framework, where the action takes the role of the model rather than the controller, although there is some overlap.



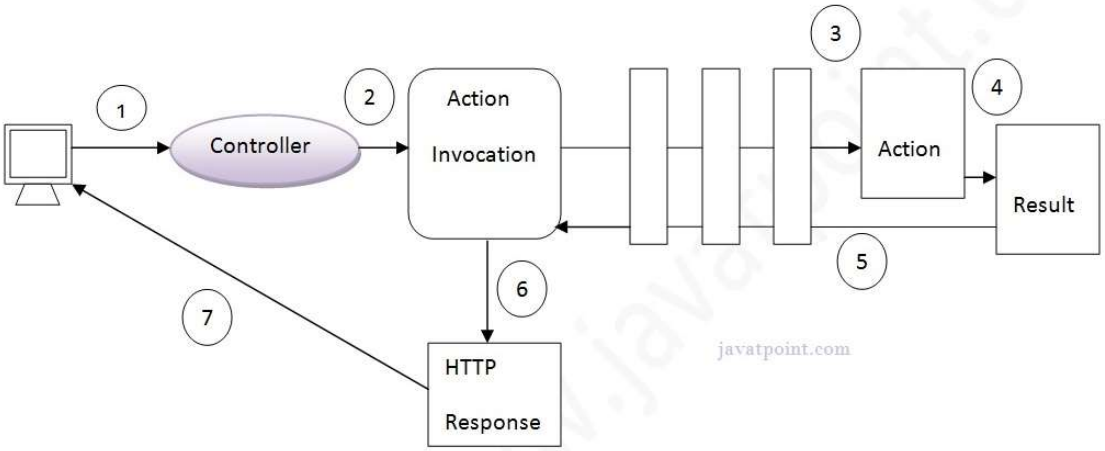
The above diagram depicts the Model, View and Controller to the Struts2 high level architecture. The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors, this model is implemented with actions, and the view is a combination of result types and results. The value stack and OGNL provides common thread, linking and enabling integration between the other components. Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc. This is the architectural overview of the Struts 2 MVC pattern. We will go through each component in more detail in the subsequent chapters.

Comparision with MVC using RequestDispatcher and the EL

Struts Flow of Control

The architecture and flow of struts 2 application, is combined with many components such as Controller, ActionProxy, ActionMapper, Configuration Manager, ActionInvocation, Inerceptor, Action, Result etc. Here,

we are going to understand the struts flow by 2 ways: 1) struts 2 basic flow 2) struts 2 standard architecture and flow provided by apache struts. Let's try to understand the basic flow of struts 2 application by this simple figure:



1. User sends a request for the action
2. Controller invokes the ActionInvocation
3. ActionInvocation invokes each interceptors and action
4. A result is generated
5. The result is sent back to the ActionInvocation
6. A HttpServletResponse is generated
7. Response is sent to the user

Processing Requests with Action Objects

In struts 2, action class is POJO (Plain Old Java Object). POJO means you are not forced to implement any interface or extend any class. Actions are the core of the Struts2 framework, as they are for any MVC (Model View Controller) framework. Each URL is mapped to a specific action, which provides the processing logic which is necessary to service the request from the user.But the action also serves in two other important capacities. Firstly, the action plays an important role in the transfer of data from the request through to the view, whether its a JSP or other type of result. Secondly, the action must assist the framework in determining which result should render the view that will be returned in the response to the request.

The only requirement for actions in Struts2 is that there must be one noargument method that returns either a String or Result object and must be a POJO. If the no-argument method is not specified, the default behavior is to use the execute() method. Optionally you can extend the ActionSupport class which implements six interfaces including Action interface.

Handling Request Parameters with FormBeans

Prepopulating and Redisplaying Input Forms

Using Properties Files

This configuration file provides a mechanism to change the default behavior of the framework. Actually, all the properties contained within the struts.properties configuration file can also be configured in the web.xml using the init-param, as well using the constant tag in the struts.xml configuration file. But, if you like to keep the things separate and more struts specific, then you can create this file under the folder WEB-INF/classes. The values configured in this file will override the default values configured in default.properties which is contained in the struts2-core-x.y.z.jar distribution. There are a couple of properties that you might consider changing using the struts.properties file