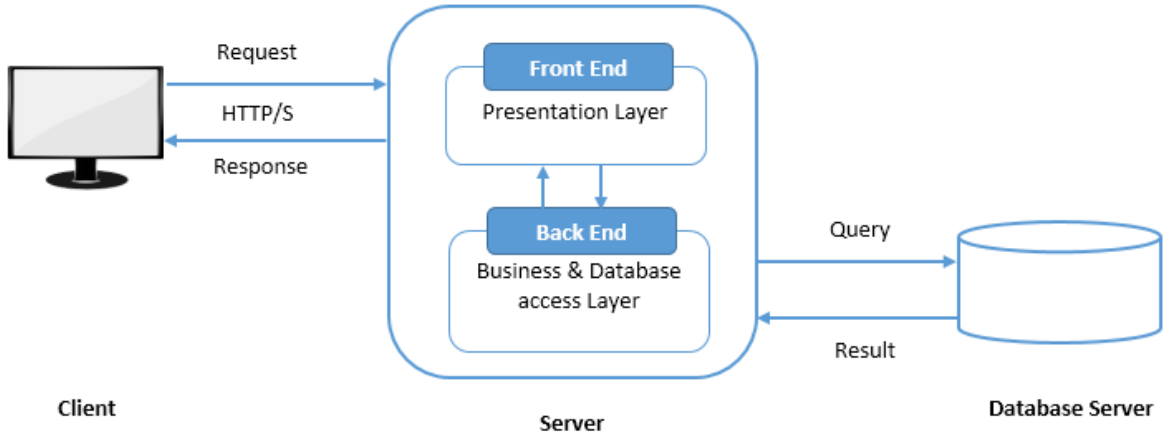
Web application Architecture/3-Tier Architecture

Web application architecture defines the interaction between components, databases, middleware systems, user interfaces, and servers in a web application.



Client: The client-side component or the browser is the key component that interacts with the user, receives the input, and manages the user interactions with the application. User inputs are validated as well if required.

Web Server: The web server or the server-side component handles the business logic and processes the user requests by routing the requests to the right component and managing the entire application operation.

Database Server: The database server provides the required data for the application. It handles data-related tasks. In a multi-tiered architecture, database servers can manage business logic with the help of stored procedures.

When a user types in a URL and hit enter the browser will find the Internet-facing computer the website lives on and requests that particular page. The server then processes the request and responds by sending files over to the browser. While processing the request if any data is required, the web server communicates with the database server. Once the response is received by the browser, it executes those files to show the requested page to the user and the user gets to interact with the website.

HTML

- Hypertext Markup Language
- In 1989 a physicist named Tim Berners Lee proposed the internet-based hypertext system and in late 1990 he specified HTML and wrote the browser and server software. HTML was publicly available in 1991 by Tim Berners Lee with the first documented version “HTML Tags”
- "Hypertext" refers to links that connect web pages to one another, either within a single website or between websites. Links are a fundamental aspect of the Web. By uploading content to the Internet and linking it to pages created by other people, you become an active participant in the World Wide Web.
- Markup language is a standard text-encoding system consisting of a set of symbols/tags inserted in a text document to control its structure, formatting, or the relationship between its parts. HTML describes the web page's structure and tells the browser how to display the element.
- HTML files must have *htm* or *html* extensions.
-

Basic Structure of an HTML Page:

```
<!DOCTYPE html>
<html>
   <head>
      <title>My First Web Page</title>
   </head>
   <body>
      Hello world!
   </body>
</html>
```

- DOCTYPE, a case insensitive element, defines this document as an HTML5 document
- html element is the root element of an HTML page
- head element contains meta-information about the HTML page
- title element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)

- body element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.

HTML Elements/Tags:

- HTML tags are case insensitive, but W3C recommends lowercase in HTML.
- Some of the html tags have end tags, like -
    - <html></html>, <head></head>, <body></body>, <h1></h1>,
- Some html tags do not require end tags, like -
    - <img >, <input>, <meta>
- HTML tag with no content is called an empty element or empty tag
    - <hr>,
- HTML elements that always start in a new line and take the full with of the browser are Known as *block level* elements. The browser automatically adds some margin before and after the element.
    - , , <h1>, <table>, <form>, <ul>, <ol>, <pre>
- HTML elements that do not start in a new line and only take up as much width as necessary are known as *inline* elements.
    - <b>, <i>, , <img>, <label>, <button>
- An HTML tag can have attributes, which provide additional information about the element. Attribute cone in a name/value pair format like name="value"
    - <img src="test_image.jpg" height="500" width="450" >
    - Google

CSS

- Cascading Style Sheet(CSS) is a style sheet language used for describing the presentation of a document written in a markup language.
- A style sheet language or style language is a computer language that expresses the presentation of a structured document.
- CSS was first proposed by Håkon Wium Lie on 10 October 1994, when he was working with Tim Berners Lee.
- Håkon Wium Lie is a Norwegian web pioneer, a standards activist, and the CTO of Opera Software from 1998 to 2016.
- The first W3C CSS recommendation(CSS1) was released in 1996.
- The CSS specifications are maintained by the World Wide Web Consortium(W3C).
- Internet media type (MIME type) text/css is regarded for use with CSS.

Syntax:

A CSS is a collection of style rules that are interpreted by the browser and then applied to the corresponding elements in the document. A style rule is made of three parts -

selector { property : value }

A selector points to the HTML element to which you want to apply a style.
A property is a type of attribute of an HTML tag and values are assigned to the property.
Multiple CSS declarations are separated by semicolons and declaration blocks are surrounded by curly braces.

Selectors:

CSS selectors are used to select or find the HTML elements in an HTML document. Selectors can be classified into the following categories.

1. Universal Selector: The universal selector (*) selects all HTML elements on a page rather than selecting elements of a specific type.

```
*{
   margin: 0px;
   padding: 0px;
   font-family: Arial, Helvetica, sans-serif;
```

```
}
```

2. Element selector: The element selector selects HTML elements based on the element name.

```
h1{
   background-color: yellow;
   padding: 5px;
   text-align: center;
}
```

3. ID Selector: The id selector uses the id attribute of an HTML element to select a specific element. The id of an element is unique within a page, so the id selector is used to select one unique element.

```
<h2 id="sub_heading">Welcome to CSS Class</h2>

#sub_heading {
   text-decoration: underline;
   color: blue;
}
```

4. Class Selector: The class selector selects HTML elements with a specific class attribute. To select elements with a specific class, write a period (.) character, followed by the class name.

```
<h4 class="tag_line">Learn CSS in a easy way</h4>

.tag_line {
   font-size: 18px;
   letter-spacing: 1.2px;
}
```

5. Group Selector: The grouping selector selects all the HTML elements with the same style definitions. separate the selectors with a comma(,).

```
h1, h2, .tag_line{
   text-align: center;
}
```

6. Descendent Selector: Descendent selectors are used to select and apply style to an element only when it lies inside a particular element.

```
<h2 id="sub_heading">Welcome to CSS <span>(Cascading Style
Sheet)</span> Class</h2>

h2 span{
   color: green;
   font-size: 16px;
   font-style: italic;
}
```

7. Child Selector: Child Selector is used to match all the elements which are children of a specified element. It gives the relation between two elements. The element > element selector selects those elements which are the children of the specific parent. This selector is very similar to descendants but has different functionality. This selector selects only the immediate descendent or direst child, whereas the descendent selector selects all the children(or grandchildren).

```
<p>
      <a href="">Click Me</a>
      <label><a href="">Click Me 2</a></label>
</p>

p a{ /* Descendent Selector */
   text-decoration: none;
}
p > a{ /* Child Selector */
   color: red;
}
```

8. Attribute Selector: The attribute selector is used to select elements with a specified attribute.

```
<input type="text">

input[type='text']{
   font-size: 20px;
   padding: 5px;
}
```

9. Pseudo-Class Selectors: A pseudo-class selector is used to define a special state of an element.

```
a:hover{
   font-size: 25px;
   color: green;
}
```

10. Pseudo-Element Selectors: A CSS pseudo-element is used to style specified parts of an element.

```
p::first-letter{
   font-size: 30px;
   color: blue;
}
```

Writing CSS into HTML:

- Inline CSS: An inline style may be used to apply a unique style for a single element. To use inline styles, add the **style attribute** to the relevant element. The style attribute can contain any CSS property.

```
<h1 style="color:blue;text-align:center;">This is a heading</h1>
```

- Internal CSS: An internal style sheet may be used if one single HTML page has a unique style. The internal style is defined inside the <style> element, inside the head section.

```
<html>
<head>
<style>
   h1 {
   color: yellow;
   text-align: center;
   }
</style>
</head>
<body>
   <h1>This is a heading</h1>
</body>
</html>
```

- External CSS: The <link> element can be used to include an external stylesheet file in your HTML document. An external style sheet is a separate text file with .css extension.

  Syntax:

```
<head>
  <link type = "text/css" href = "..." media = "..." />
</head>
```

  Example:

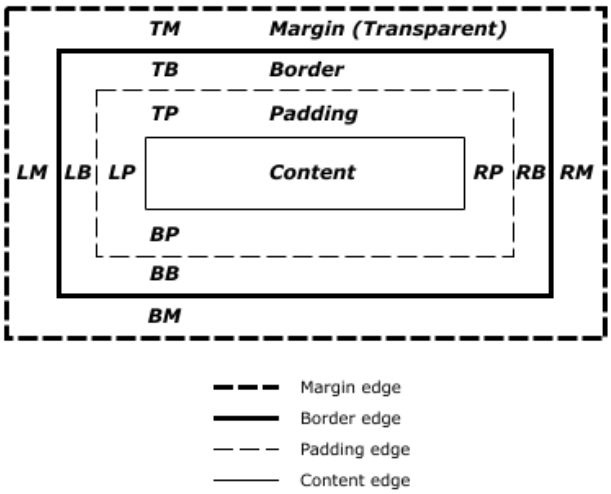
```
<html>
<head>
   <link rel="stylesheet" href="css/style.css">
</head>
<body>
   <h1>Hello World</h1>
</body>
</html>

css/style.css:
h1 {
   background-color: yellow;
   padding: 5px;
   text-align: center;
}
```

CSS Box Model:

The CSS box model describes the rectangular boxes that are generated for elements in the document tree and laid out according to the visual formatting model.

Each box has a content area (e.g., text, an image, etc.) and optional surrounding padding, border, and margin areas; the size of each area is specified by properties defined below. The following diagram shows how these areas relate and the terminology used to refer to pieces of margin, border, and padding:

The margin, border, and padding can be broken down into top, right, bottom, and left segments (e.g., in the diagram, "LM" for left margin, "RP" for right padding, "TB" for top border, etc.).



Margin properties specify the width of the margin area of a box. The 'margin' shorthand property sets the margin for all four sides while the other margin properties only set their respective side.

```
body { margin: 2em }         /* all margins set to 2em */
body { margin: 1em 2em }     /* top & bottom = 1em, right & left = 2em */
body { margin: 1em 2em 3em } /* top=1em, right=2em, bottom=3em, left=2em */
```

```
body {
  margin-top: 1em;
  margin-right: 2em;
  margin-bottom: 3em;
  margin-left: 2em;
}
```

The padding properties specify the width of the padding area of a box. The 'padding' shorthand property sets the padding for all four sides while the other padding properties only set their respective side.

The border properties specify the width, color, and style of the border area of a box. These properties apply to all elements.

JavaScript

- JavaScript (JS) is a lightweight, object-oriented, interpreted, or just-in-time compiled programming language.
- JavaScript engines were originally used as scripting language only in web browsers, but are now core components of some servers and various applications. The most popular runtime system for this usage is Node.js.
- JavaScript was invented by Brendan Eich in 1995 and became an ECMA standard in 1997. ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6. Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

Writing JavaScript into HTML:

JavaScript syntax is embedded into an HTML file. Since all JavaScript need to be included as an integral part of an HTML document when required, the browser needs to be informed that specified section of HTML code is JavaScript. The browser will then use its built-in JavaScript engine to interpret this code.

The browser is giving this information using <script>...</script> tag. <script> tag can be placed inside <head> or <body> tag or can be placed in both.

JS can be written in three places -

1. Inline JS: When JavaScript was written within the html element using attributes related to events of the element then it is called inline JavaScript.

```
<html>
<title>Inline JavaScript</title>
<body>
<button onclick="document.write('Hello World')">Click</button>
</body>
</html>
```

2. Internal JS: We can place any number of scripts in an HTML document in the <body> or in the <head> section of an HTML page, or in both. <script> tag is used to write JS in an HTML document.

```
<html>
<title>Internal JavaScript</title>
<body>
```

```
<script language=”JavaScript”>
   document.write("<h1>JavaScript</h1>")
</script>
</body>
</html>
```

3. External JS: We can create an external JavaScript file and embed it in any html page. It provides code reusability because a single JavaScript file can be used on several html pages. An external JavaScript file must be saved by the .js extension and embedded to the HTML page with the help of the src attribute of the <script> tag.

```
<html>
<title>External JavaScript</title>
<body>

<script type="text/javascript" src="js/js-master.js"></script>
</body>
</html>

js/js-master.js
document.write("Welcome to ES6");
```

Data Type & Variable:

JavaScript is a dynamic and weakly typed language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types. JavaScript is also a weakly typed language, which means it allows implicit type conversion when an operation involves mismatched types, instead of throwing type errors.

A JS variable can be declared with var, let or const keyword. Till 2015 var was the only keyword to declare a JS variable. var and let are used to declare any reguler varailbe where as const is used to declare a constant.

Ex(Dynamic):

let foo = 42; // foo is now a number

foo = "bar"; // foo is now a string
foo = true; // foo is now a boolean

Ex(Weakly Typed):
const foo = 42; // foo is a number
const result = foo + "1"; // JavaScript coerces foo to a string, so it can be concatenated with the other operand
console.log(result); // 421

JS supports primitive data types like boolean, Null, Undefined, Number, BigInt, String and complex and some non-primitive data types like Array and Objects.

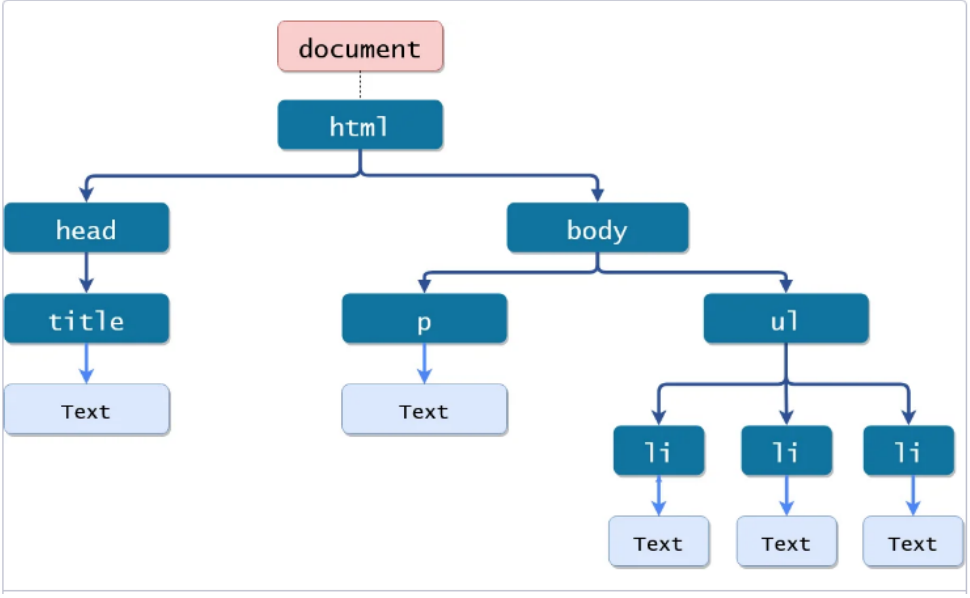
if-else:
if(condition1){
  //block of code
}
else if(condition2){
  //block of code
}
else{
  //block of code
}
	switch:
	switch(expression) {
	  case x:
	  // code block
	  break;
	  case y:
	  // code block
	  break;
	  default:
	  // code block
	}
		while loop:
		while (condition) {
		  // code block to be executed
		}
		
		do-while loop:
		do {
		  // code block to be executed
		}
		while (condition);

break:
if(x>y) break;
 	
continue:
if(x>y)continue;
	Function call:
	x=abs(y)
	
	return:
	return x*y;
		for loop:
		for (statement 1; statement 2; statement 3) {
		  // code block to be executed
		}

HTML DOM (Document Object Model):

- The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.
- When a web page is loaded, the browser creates a Document Object Model of the page.

- With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
- The HTML DOM model is constructed as a tree of Objects:



Several ways to find HTML elements are-

- By id: document.getElementById("element id")
- By Name: document.getElementsByName("element name")
- By tag name: document.getElementsByTagName("html tag name")
- By Class name: document.getElementsByClassName("class name")

JavaScript Event & Event Handling:

The change in the state of an object is known as an Event. JavaScript is used in HTML pages, JavaScript can react to these events. This process of reacting over the events is called Event Handling. Thus, js handles the HTML events via Event Handlers.
Once the HTML object recognizes that an event has occurred this knowledge has to be passed to JavaScript so that JavaScript also recognizes the event. To facilitate this JavaScript provides a number of named JavaScript 'Event Handlers'. The names of these event handlers are descriptively bound to an HTML objects event name.

Attribute / Event Handler	Description
Form Events:	
onblur()	Fires the moment that the element loses focus
onchange()	Fires the moment when the value of element is changed.
onfocus()	Fires, when the element gets focus.
oninput()	Script to be run when an element gets user input.
onreset()	Fires when the reset button in a form is clicked
onsubmit()	Fires when a form is submitted
onselect()	Fires after some text has been selected in an element.
Keyboard Events:	
onkeydown()	Fires when a user is pressing a key.
onkeypress()	Fires when a user presses a key
onkeyup()	Fires when a user release a key.
Mouse Event:	
onclick()	Fires on a mouse click on the element
ondblclick()	Fires on a mouse double-click on the element.
onmouseout()	Fires when the mouse pointer moves out of an element
onmouseover()	Fires when the mouse pointer moves over an element.
Page Events:	
onload()	Fires when page is loaded
onunload()	Fires when page is unloaded

Ex: (Events & selectors)

```
<!DOCTYPE html>
<html>
<body>
   <h1>Hello World</h1>
   <h1 id="js">Java Script</h1>
   <button onclick="changeToBlue()">Blue</button>
   <button onclick="changeToRed()">Red</button>


   <script>
      const h1 = document.getElementsByTagName("h1")
      h1[0].innerHTML = "Hi..."

      const js = document.getElementById("js")
      function changeToBlue(){
         js.style.color = 'blue'
      }
      function changeToRed(){
         js.style.color = 'red'
      }
   </script>
</body>
</html>
```

Dialog Boxes:

JavaScript provides the ability to pick up user input or to display small amount of text to the user by using dialog boxes.

These dialog boxes appear as separate windows and their content depends on the information provided by the user.

1. Alert Box: The JavaScript alert() method takes a string as an argument and displays an alert dialog box in the browser window when invoked by appropriate JavaScript. The alert dialog box displays the string passed to the alert() method, as well as an ok button. The JavaScript and the HTML program, in which this code snippet is held, will not continue processing until the ok button is clicked.

```
Ex:
<html>
<head>
<script>
function hello(){
        var name=document.getElementById("nm").value;
        alert("Hello "+name);
}
</script>
</head>
<body>
<center>
<input type="text" id="nm" />
<button onclick="hello()">Click</button>
</center>
</body>
</html>
```

2. Prompt Box: An alert dialog box cannot be used to customize any web page output based on user input, which is what user interaction requires. JavaScript provides a prompt dialog box for this. The prompt() method instantiate the prompt dialog box which displays a specified message. In addition, the prompt dialog box also provides a single data entry field, which accepts user input.

   The prompt dialog box has –

   i. Display predefined message.
   ii. Display a text box and accepts user input
   iii. Display ok and cancel button.

   Clicking on the ok button causes the text typed inside the textbox to be passed to the program environment (JavaScript) Clicking on the cancel button causes a NULL value to be passed to the environment.

```
Ex:
<html>
<head>
<script>
function add(){
```

```
        var a=parseInt(prompt("Enter first number: "));
        var b=Number(prompt("Enter second number: "));
        alert("Sum= "+(a+b));
}
</script>
</head>
<body onload=add()>
</body>
</html>
```

3. Confirm Box: JavaScript provides a third type of a dialog box, called the confirm dialog box. As the name suggests, this dialog box serves as a technique for confirming user action. The confirm dialog box display the following information:
    i. A predefined message
    ii. Ok and cancel button

    JavaScript dialog box, causes program execution to halt until user action takes place. User action can be either click ok or cancel button. Clicking on the ok button causes TRUE to pass to the program which called the confirmed dialog box and cancel button returns FALSE.

```
Ex:
<html>
<head>
<script>
function change(){
        var tt=confirm("Want to change the color?");
        if(tt==true)
        document.body.style.backgroundColor='CYAN';
}
</script>
</head>
<body>
<button onclick=change()>Change</button>
</body>
</html>
```

Form Validation:

```
<html>
<title>Form Validation using JavaScript</title>
<body>
   <form action="thanku.html" method="post" onsubmit="return validate();">
      <table align="center">
         <tr><td>Name:</td><td><input type="text" id="name"></td></tr>
         <tr><td>Email:</td><td><input type="email" id="email"></td></tr>
         <tr><td>Mobile:</td><td><input type="tel" id="mob"></td></tr>
         <tr><td>Password:</td><td><input type="password" id="password"></td></tr>
         <tr><td colspan="2" align="center">
            <input type="submit" value="Register">
         </td></tr>
         </table>
   </form>
<script type="text/javascript" src="js/js-master.js"></script>
</body>
</html>
```

js/js-master.js

```
function validate(){
   const name = document.getElementById("name").value;
   const email = document.getElementById("email").value;
   const mob = document.getElementById("mob").value;
   const password = document.getElementById("password").value;

   if(name==""){
      alert("Please enter your name");
      return false;
   }

   if(mob==""){
      alert("Please enter your mobile number");
      return false;
   } else if(isNaN(mob)|| mob.length !=10){
      alert("Please enter a valid 10 digit mobile number");
      return false;
```

```
    }

    if(password.length<6 || password.length>15){
        alert("Please enter a password of 6 - 15 character")
        return false;
    }

    let atPos = email.indexOf('@')
    let dotPos = email.lastIndexOf('.')
    console.log(email,atPos,dotPos)
    if(dotPos <= 0 || atPos<=0 || dotPos -atPos < 4 || dotPos === email.length-1){
        alert("Please enter a valid email");
        return false;
    }
}
```

JavaScript Object:

JavaScript object is a non-primitive data-type that allows us to store multiple data in a key-value pair.

```
const object_name = {
  key1: value1,
  key2: value2
}
```

JavaScript object are capable of holding the state(data) and behavior(function) of a variable.
In JavaScript, "key: value" pairs are called properties. We can access the values by using the keys and the dot operator or bracket notation.
A JS object is capable of storing an array, functions, and another object inside.

```
const student = {
        name: 'Jhon Doe',
        roll: 1,
        mobile: [7788668877, 9988778899],
```

```
        email: 'jhon@gmail.com',
        address:{
            city: 'Bhubaneswar',
            state: 'Odisha',
            pin: 751024
        },
        getDetails : function(){

console.log(this.name,this.roll,this.mobile[0],this.mobile[1],this.email,this.address.city,
this.address.state)
        },
        changeEmail: function(email){
            this.email=email
        }
    }

    console.log(student.name)
    console.log(student["roll"])
    student.getDetails()
    student.changeEmail('jhondoe@gmail.com')
    student.getDetails()
```

JSON:

- JSON stands for JavaScript Object Notation.
- This is an open standard file and data interchange format that uses human-readable text to store and transmit data objects.
- JSON stores and transmit data in “key : value” or “attribute : value” pair format. The key or attribute is enclosed by a double quote (“key”)
- Though JSON is derived from JavaScript, this is a language-independent data format.
- JSON filenames use the extension .json.

```
{
    "name" : "John Doe",
    "age" : 30,
    "mobile" : [7778889999, 9999888877]
}
```

JavaScript Object VS JSON:

JSON	JavaScript Object
The key in the key/value pair should be in double quotes.	The key in the key/value pair can be without double quotes.
JSON cannot contain functions.	JavaScript objects can contain functions.
JSON can be created and used by other programming languages.	JavaScript objects can only be used in JavaScript.

{
"name" : "John Doe",
"age" : 30,
"mobile" : [7778889999, 9999888877]
}
	const person = {
	name : "John Doe",
	age : 30,
	mobile : [7778889999, 9999888877]
	}

XML:

- XML or Extensible Markup Language is a markup language and file format for storing, transmitting, and reconstructing arbitrary data.
- XML is similar to HTML, but without predefined tags to use. Instead, we can define our own tags specifically for our needs.
- XML document may have a XML prolog. The XML prolog is optional. If it exists, it must come first in the document.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- XML documents must contain one root element that is the parent of all other elements.

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

```
<student>
   <name>John Doe</name>
   <age>22</age>
```

```
   <mobile>
      <mobile_one>9998899778</mobile_one>
      <mobile_two>9997788987</mobile_two>
   </mobile>
   <address>
      <city>Bhubaneswar</city>
      <state>Odisha</state>
   </address>
</student>
```

jQuery:

jQuery is a fast, small, feature-rich, and "write less, do more" JavaScript library. It is free, open-source software using the permissive MIT License. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish and wraps them into methods that you can call with a single line of code.

The jQuery syntax is tailor-made for selecting HTML elements and performing some action on the element(s).

Basic syntax is: $(selector).action()

- A $ sign to define/access jQuery
- A (selector) to "query (or find)" HTML elements
- A jQuery action() to be performed on the element(s)

jQuery selectors are used to "find" (or select) HTML elements based on their name, id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors, and in addition, it has its own custom selectors.

JavaScript

```
<!DOCTYPE html>
<html lang="en">
<body>
   <h1 id="js">Java Script</h1>
   <button onclick="changeText()">Change</button>

 <script>
      function changeText(){
         document.getElementById("js").innerHTML = "JS";
      }
```

```
    </script>
</body>
</html>
```

jQuery

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
    </head>
<body>
    <h1 id="jq">jQuery</h1>
    <button id="change">Change</button>

    <script>
        $("#change").click(function(){
            $("#jq").text("jQ");
        });
    </script>
</body>
</html>
```

JDBC

- JDBC Stands for Java Data Base Connectivity.
- JDBC driver was developed by “Sun Microsystem” in the year 1996. In java, JDBC is an API. JDBC API consists of a set of Java classes, interfaces, and exceptions bound to a specification called, X/Open SQL CLI (Call Level Interface).
- JDBC is a bridge between Java applications and database servers. JDBC converts the Java call into database calls.
- JDBC technology defines the “Write Once, Run Anywhere” paradigm, i.e. same java program can run on a PC, a workstation, or a network and can connect to any vendor’s DBMS, simply by changing the JDBC middleware.

JDBC Architecture:

- The singular purpose of JDBC API is to provide a resource to developers via which they can send a request to the database server and get a response.
- The JDBC API makes use of a driver manager, which is bound to the database-specific driver.

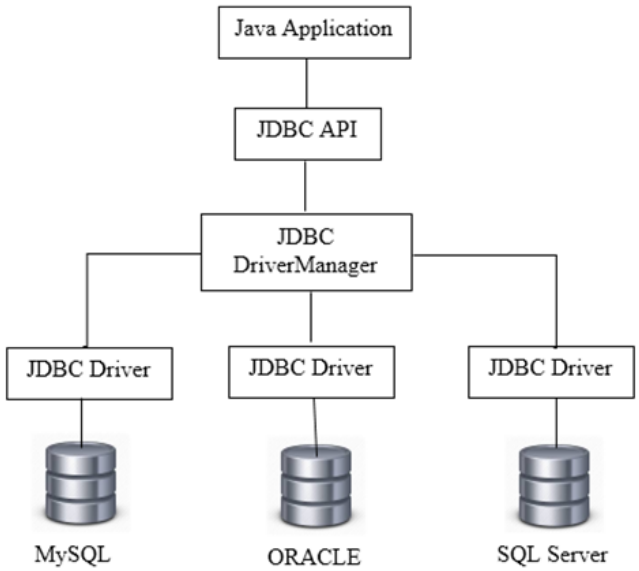


Fig: JDBC Architecture

- The JDBC driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases and also verifies whether the driver used for accessing each data source is correct or not.
- A JDBC driver translates standard JDBC calls into a native database call, which enables an application module to communicate with the database.
- The JDBC API is available in two packages: java.sql (core API) package and javax.sql (Additional API) package.

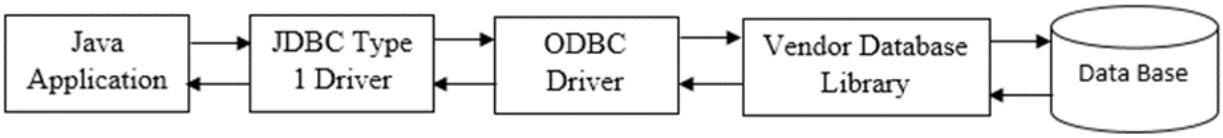
JDBC Driver:

To connect with individual databases, JDBC requires drivers for each database. Those drivers come in four varieties.

➔ Type – 1: JDBC-ODBC Bridge
➔ Type – 2: Java Native Interface
➔ Type – 3: Java Network Protocol Driver
➔ Type – 4: Java Database Protocol Driver

Type-1 Driver:

Type 1 drivers act as a bridge between JDBC and the ODBC (Open Database Connectivity) database connectivity mechanism. Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into ODBC function calls.



Advantages:

The type-1 driver is easy to use and is also called a Universal driver because it can be used to connect to any of the databases.

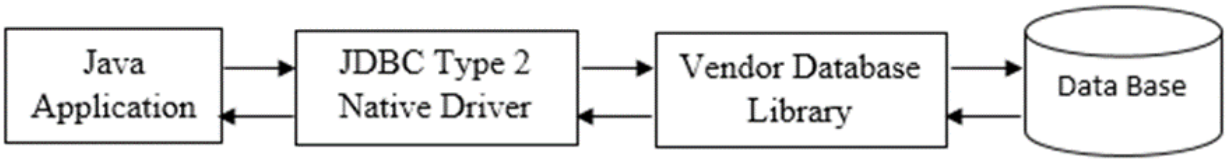
Disadvantage:

- Type 1 drivers are the slowest database access API of all, due to multiple levels of translation that have to occur.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secure.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.

Type-2 Driver:

Type-2 drivers mainly use the Java Native Interface (JNI) to translate calls to local database API. In order to interact with a different database, this driver needs their local API. The vendor-specific driver must be installed on each client machine.



Advantages:

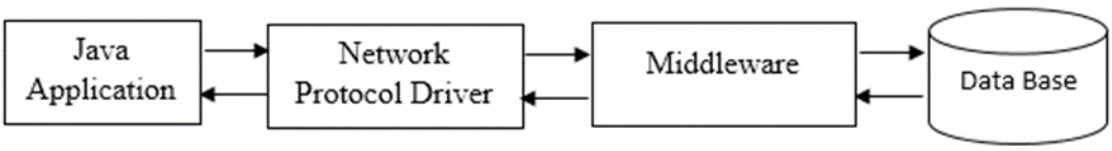
Secure and faster than Type 1 driver.

Disadvantages:

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on the client machine.
- Type-2 driver isn't written entirely in java, that's why it isn't a portable driver

Type-3 Driver:

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need for individual client-side installation.



Advantages:

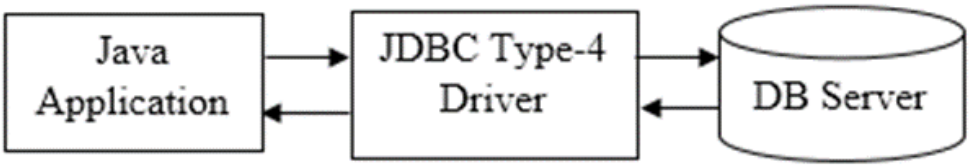
- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client-side library is required because the application server can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on the client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type-4 Driver:

Type-4 driver is designed to interact with the DB server directly without taking the support of the vendor DB library and ODBC driver.



Advantage:

- It gives better performance than Type-1, 2, and 3 drivers.
- No need to work with vendor DB library and ODBC driver.
- This driver is fully written in java. So it is portable.
- It is suitable for large-scale projects.
- This driver can be downloaded directly from the internet to the client side.

Disadvantage:

For every database server we need one separate Type-4 driver.

How to Access Database:

To access the database using JDBC, the following are the steps involved:

1. Configuring JDBC Driver
2. Creating a database connection
3. Executing queries
4. Processing the result
5. Closing the database connection

Configuring JDBC Driver:

The first step to establishing a database connection using the JDBC driver involves loading the specific driver class into the application's JVM. This makes the driver available later when required for opening the connection.

The method **Class.forName(String DriverClass).newInstance()** is used to load JDBC driver class.

Ex:

Class.forName("com.mysql.jdbc.Driver").newInstance(); //for MySQL

Class.forName("oracle.jdbc.driver.OracleDriver").newInstance(); //For ORACLE

In many cases it is possible to load a new JDBC driver using only the Class.forName() method and excluding the newInstance() method.

Ex:

Class.forName("oracle.jdbc.driver.OracleDriver"); //For ORACLE

Class.forName() is a static method. This instructs the JVM to dynamically locate, load, and link the class specified to it as a parameter. If the class cannot be located, then ClassNotFoundException() is thrown.

The newInstance() method indicates that a new instance of the current class should be created.

Creating a Database Connection:

Once the driver is loaded, a database connection needs to be made. A database URL identifies a database connection and notifies the driver manager about which driver and data source is used.

To create a database connection, the JDBC connection method getConnection() of DriverManager is used as follows:

Connection con=DriverManager.getConnection("Database URL", "Username", "Password");

Syntax for Database URL: jdbc: <subprotocol>:<subname>

Vendor	Database URL	Driver Used
ORACLE	jdbc:oracle:thin:@Server:Port:Instance_name	Oracle Type 4 JDBC Driver (Ojdbc14.jar)
MySQL	jdbc:mysql://Server[:Port]/Database_name	MySQL Connector/J

getConnection() method is passed with a specific formatted URL that specifies the database. The URL used is dependent upon the JDBC driver implemented.

Executing Queries:

After establishing a database connection there should be some way to execute queries. There are three ways of executing a query:

i. Standard Statement

ii. Prepared Statement

iii. Callable Statement

Processing the Results:

A ResultSet object is used to process the data of a database. Data retrieved by executing the queries are stored in a ResultSet object. Several methods are present in the ResultSet interface, those are used to process the data.

Closing the Database Connection:

Since database connections are valuable application resource and consume a lot of system resource to create, the DB connection should be closed only when all table data processing is complete. The connection object has a built-in method, close() for this purpose.

Though the JVM's built-in garbage collection process will eventually release resources that are no longer active, it is always a good practice to manually release these resources as soon as they are no longer useful.

close() method is used as follows:

➔ rs.close();
➔ stmt.close();
➔ conn.close();

Connection Interface:

- Connection is a pre-declared interface present in java.sql package.
- As it is an interface, it cannot be instantiated. If the programmer wants to instantiate it then the programmer have to call a method of DriverManager class-

  public static Connection getConnection(String url, String username, String password)throws SQLException.

- We construct the object of the Connection interface to access the session to established with a database server.

Methods:

- public void close()throws SQLException: used to close the connection between the JDBC API and the driver.

- public void commit()throws SQLException: used to commit a SQL query into the database.
- public statement createStatement()throws SQLException: used to create a statement object that able to execute in the JDBC API.
- public void setAutoCommit()throws SQLException: used to set the auto-commit mode.
- public boolean getAutoCommit()throws SQLException: retrieve the status of the auto-commit mode.
- public void rollback()throws SQLException: used to roll back the transaction.
- public DatabaseMetaData getMetaData()throws SQLException: used to retrieve the metadata object.

Example: How to establish Connection

```
import java.sql.*;
public class ConnectionEstablish{
    public static void main(String args[]){
            try{
                    //Step -1: Load The Class
                    Class.forName("oracle.jdbc.driver.OracleDriver");
                    //Step 2: Establish Connection
                    Connection con = DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","admin","admin");
                    if(con!=null){
                            System.out.println("Connection Successfully Established");
                    }
                    //Step 3: Connection close
                    con.close();
            }catch(ClassNotFoundException ce){
                    System.out.println("Class Not Found");
            }catch(SQLException se){
                    System.out.println("SQL Exception");
                    se.printStackTrace();
            }
    }
 }
```

Statement Interface:

- Statement is a pre-declared interface present in java.sql package.
- As it is an interface, we unable to instantiate the statement interface.
- We can constrict the object of the Statement interface by calling a method of the Connection interface. – **public abstract Statement createStatement() throws SQLException**
- At the time of calling this method, we are not sending any queries to the database server. After the object is constructed we can send the SQL query to the database server with the help of some pre-declared method.
- The main responsibility of the Statement object is to send the SQL queries to the database server.

Methods:

- public abstract int executeUpdate(String query)throws SQLException: use to execute all the DDL and DML commands. In the case of insert statement this method returns 1 and in the case of update and delete operation it returns a number of rows affected by the query.
- public abstract ResultSet executeQuery(String query)throws SQLException: returns the base address of ResultSet interface. In java ResultSet is a cursor. ResultSet cursor is used to retrieve the data from the table in row by row.
- public abstract boolean execute(String query)throws SQLException: is a method that is used to execute any SQL statements. This method returns a Boolean value indicating whether the SQL statement generates a ResultSet or not.
- public abstract ResultSet getResultSet()throws IOException: construct an object of the ResultSet interface.
- public abstract int getUpdateCount()throws SQLException: returns how many rows is updated.
- public abstract void close()throws SQLException: used to close the statement object.

Example: Create Table (use of Statement Object)

```
import java.sql.*;
class createTable{
public static void main(String args[]){
        try{
                //Step 1: Load the class
```

```
                Class.forName("oracle.jdbc.driver.OracleDriver");
                //Step 2: Establish the connection
                Connection con = DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","admin","admin");
                //Step 3: Execute Query
                Statement st=con.createStatement();
                int i=st.executeUpdate("create table student(roll number(5),name
varchar2(20),course varchar2(10),fees number(8,2))");
                // Step 4: Process Result
                if(i==0){
                        System.out.println("Table Created");
                        System.out.println(i);
                }
                //Step 5: CLose Connection
                st.close();
                con.close();
                }catch(Exception ee){
                        ee.printStackTrace();
                }
        }
}
```

Example: Insert Data into Table

```
import java.sql.*;
class InsertData{
        public static void main(String args[]){
                try{
                        //Step 1: Load the class
                        Class.forName("oracle.jdbc.driver.OracleDriver");
                        //Step 2: Establish the connection
                        Connection con = DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","admin","admin");
                        //Step 3: Execute Query
                        Statement st=con.createStatement();
                        int i = st.executeUpdate("insert into student
values(1,'Rakesh','BCA',50000.50)");
                        // Step 4: Process Result
                        if(i>0){
```

```
                        System.out.println(i+" Row Inserted");
                }
                //Step 5: CLose Connection
                st.close();
                con.close();
        }catch(Exception ee){
                ee.printStackTrace();
        }
    }
}
```

ResultSet interface:

- ResultSet is a pre-declared interface present in java.sql package.
- ResultSet is a cursor, which is used to return the data from the table in row-by-row format.
- As it is an interface, it cannot be instantiated. If we want to instantiate the ResultSet interface then we have to call a method of the Statement interface –

  public abstract ResultSet executeQuery(String query)throws SQLException

  public abstract ResultSet getResultSet()throws IOException

- Every statement object must have a ResultSet object. When we construct a ResultSet object then it points to before the first object.
- We construct the ResultSet object to access data from a table in row-by-row format. When we retrieve the data from the row, we have to specify the column. The ResultSet object points to a cursor that is pointing to the current row.
- ResultSet interface is having a pre-declared method through which to check the next row is available or not. – public abstract Boolean next() throws SQLEeception. When the next() method is called, then it checks the first row is available or not.
- By using getXXX() method of the ResultSet interface we retrieve the data from the table.

Methods:

- public abstract boolean next()throws SQLException: used to navigate across data rows in the opened cursor one row at a time.
- public abstract String getString(String column_name)throws SQLException

- public abstract int getInt(String column_name)throws SQLException
- public abstract double getDouble(String column_name)throws SQLException
- public abstract char getChar(String column_name)throws SQLException
- public abstract byte getByte(String column_name)throws SQLException
- public abstract byte[] getBytes(String column_name)throws SQLException
- public abstract Date getDate(String column_name)throws SQLException
- public abstract InputStream getBinaryStream(String column_name)throws SQLException

Example: Display Table Data (Using ResultSet Object)

```
import java.sql.*;
class DisplayData{
	public static void main(String args[]){
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");
			Connection con =  DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","admin","admin");
			Statement st=con.createStatement();
			ResultSet rs=st.executeQuery("select * from student");
			while(rs.next()){
				int roll=rs.getInt("roll");
				String name=rs.getString("name");
				String course=rs.getString("course");
				double fee=rs.getDouble("fees");
				System.out.println(roll+"\t"+name+"\t"+course+"\t"+fee);
			}
			rs.close();
			st.close();
			con.close();
		}catch(Exception ee){
			ee.printStackTrace();
		}
	}
}
```

studen101 table:

Name	Null?	Type
ROLL		NUMBER(5)
NAME		VARCHAR2(20)
COURSE		VARCHAR2(10)
FEES		NUMBER(8,2)

JDBC Example (Insert Multiple Rows)

```
import java.sql.*;
import java.util.Scanner;
public class InsertMultiple{
	public static void main(String args[]){
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");
			Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","admin","admin");
			Statement smt=con.createStatement();
			Scanner sc=new Scanner(System.in);
			System.out.print("Number of rows to be inserted: ");
			int n=sc.nextInt();
			for(int i=0;i<n;i++){
				System.out.print("Name: ");
				String name=sc.next();
				System.out.print("Roll: ");
				int roll=sc.nextInt();
				System.out.print("Branch: ");
				String branch=sc.next();
				System.out.print("Fee: ");
				double fee=sc.nextDouble();
				smt.executeUpdate("Insert into student101 values("+roll+
","'"+name+"','"+branch+"',"+fee+")");
			}
			smt.close();
```

```
                con.close();
            }catch(Exception ee){
                System.out.print(ee.getMessage());
            }
        }
}
```

JDBC Example (Update)

```
import java.sql.*;
public class Update{
    public static void main(String args[]){
        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:XE","admin","admin");
            Statement smt=con.createStatement();
            int i=smt.executeUpdate("update student101 set fees=100000 where
course='BCA'");
            System.out.println(i+" Rows Updated");
            smt.close();
            con.close();
        }catch(Exception ee){
            ee.printStackTrace();
        }
    }
}
```

JDBC Example (Delete)

```
import java.sql.*;
public class Delete{
    public static void main(String args[]){
        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:XE","admin","admin");
            Statement smt=con.createStatement();
```

```
                int i=smt.executeUpdate("delete from student101 where roll=1");
                System.out.println(i+" Rows Deleted");
                smt.close();
                con.close();
            }catch(Exception ee){
                ee.printStackTrace();
            }
        }
}
```

JDBC Transaction:

A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed. There are times when we do not want one statement to take effect unless another one completes. For example, when you transfer money to your friend's account the amount should be deducted from your account and added to your friend's account. The way to be sure that either both actions occur or neither action occurs is to use a transaction.

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

The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where con is an active connection:

```
con.setAutoCommit(false);
```

After the auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly. All statements executed after the previous call to the method commit are included in the current transaction and committed together as a unit.

```
con.commit();
```

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. The method rollback aborts a transaction and restores values to what they were before the attempted update.

```
con.rollback( );
```

During the rollback process if you want to rollback to a specific satet of the transaction you can use a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.
setSavepoint(String savepointName) − Defines a new savepoint. It also returns a Savepoint object.
rollback (String savepointName) method rolls back work to the specified savepoint.

Ex:

```
import java.sql.*;
class transaction{
	public static void main(String args[]){
		Connection con = null;
		Statement smt = null;
		String qry = "";
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");
			con = DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","admin");
			con.setAutoCommit(false);
			qry = "update customer set amount=amount-100 where id=1";
			smt=con.createStatement();
			int i = smt.executeUpdate(qry);
			if(i>0){
				qry = "update customer amount=amount+100 where id=2";
				smt=con.createStatement();
				i = smt.executeUpdate(qry);
				if(i>0){
					System.out.println("Transaction Successful");
					con.commit();
				}
			}

		}catch(Exception ee){
			System.out.println(ee);
			try{
				con.rollback();
			}catch(Exception e1){
				System.out.println(e1);
```

```
                }
            }finally{
                try{
                    smt.close();
                    con.close();
                }catch(Exception e2){
                    System.out.println(e2);
                }

            }
        }
}
```

In the above example we are trying to transfer some amount from user 1 to user 2. But there is a problem in the 2nd update query which can be handled by the JDBC transaction control.

Java DatabaseMetaData interface

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

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.

The getMetaData() method of Connection interface returns the object of DatabaseMetaData. Syntax:

public DatabaseMetaData getMetaData()throws SQLException

Example:

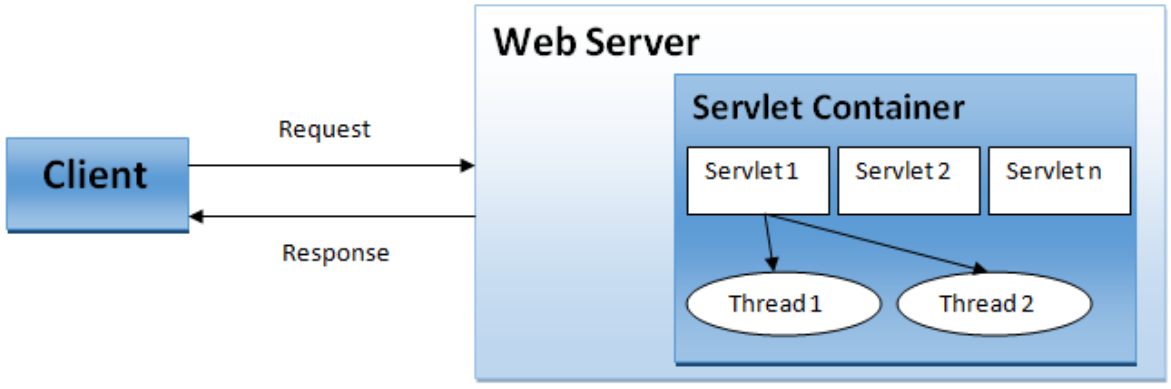
```
import java.sql.*;
class dbmd{
	public static void main(String args[]){
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");

			Connection con=DriverManager.getConnection(
			"jdbc:oracle:thin:@localhost:1521:xe","system","admin");
			DatabaseMetaData dbmd=con.getMetaData();

			System.out.println("Driver Name: "+dbmd.getDriverName());
			System.out.println("Driver Version: "+dbmd.getDriverVersion());
			System.out.println("UserName: "+dbmd.getUserName());
			System.out.println("Database Product Name:
"+dbmd.getDatabaseProductName());
			System.out.println("Database Product Version:
"+dbmd.getDatabaseProductVersion());

			con.close();
		}catch(Exception e){
			System.out.println(e);
		}
	}
}
```

Servlet

A servlet is a Java class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model.
Servlet is a java program, exists and executes in j2ee servers, used to receive the http request, process it and send response to client.
Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls and receive all the benefits of the mature Java language, including portability, performance, reusability, and crash protection.
The *javax.servlet* and *javax.servlet.http* packages provide interfaces and classes for writing servlets.

Servlet Architecture

Servlets are Java classes used to create dynamic web applications. These java classes are managed by Web Container or Servlet Container which can be part of the web server or application server. When the request comes to the web server, it forwards the request to the servlet container. The servlet container initializes the servlet class. Then the servlet class processes the request and sends back the response to the client.



Client: The client in the above architecture is basically a web browser which sends requests to the web server and processes the response when received from the web server.
Web Server: Primary job of a web server is to process the requests and responses that a user sends over time and maintain how a web user would be able to access the files that

has been hosted over the server. The server we are talking about here is a software which manages access to a centralized resource or service in a network.
Servlet Container: Servlet container is another typical component in servlet architecture which is responsible for communicating with the servlets. A servlet container is responsible for managing the lifecycle of servlets. Servlet container handles multiple requests by creating multiple threads of the required servlet.

Servlet Life Cycle

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

1. Load the servlet class
2. Create Servlet Object / Servlet instantiation
3. Initialize the servlet / call init()
4. Request processing / call service()
5. Destroy the servlet object / call destroy()

Step 1: Loading of Servlet
When the web server (e.g. Apache Tomcat) starts up, the servlet container deploys and loads all the servlets.

Step 2: Creating instance of Servlet
Once all the Servlet classes are loaded, the servlet container creates instances of each servlet class. Servlet container creates only one instance per servlet class and all the requests to the servlet are executed on the same servlet instance.

Step 3: Invoke init() method
Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet.

Step 4: Invoke service() method
Each time the web server receives a request for servlet, it spawns a new thread that calls service() method. If the servlet is GenericServlet then the request is served by the service() method itself, If the servlet is HttpServlet then service() method receives the request and dispatches it to the correct handler method(doGet, doPost) based on the type of request.

Step 5: Invoke destroy() method
When the servlet container shuts down(this usually happens when we stop the web server), it unloads all the servlets and calls destroy() method for each initialized servlet.

How to Create a Servlet Class ?

A servlet class can be created by inheriting from any one of the following three Interfaces or abstract classes.

1. Servlet Interface
2. GenericServlet Abstract Class
3. HttpServlet Abstract Class

Servlet Interface

Servlet interface provides common behavior 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.
3 Life Cycle Methods -

1. public void init(ServletConfig config) - initializes the servlet. It is the life cycle method of the servlet and is invoked by the web container only once.
2. public void service(ServletRequest request, ServletResponse response) - provides response for the incoming request. It is invoked at each request by the web container.
3. public void destroy() - is invoked only once and indicates that servlet is being destroyed.

2 Non-Life Cycle Methods -

1. public ServletConfig getServletConfig() - returns the object of ServletConfig.
2. public String getServletInfo() - returns information about servlets such as writer, copyright, version etc.

Ex:

ServletDemo.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
```

```
import javax.servlet.ServletResponse;
public class ServletDemo implements Servlet {

	@Override
	public void destroy() {
		System.out.print("Servlet Destroyed");

	}

	@Override
	public ServletConfig getServletConfig() {
		return null;
	}

	@Override
	public String getServletInfo() {
		return null;
	}

	@Override
	public void init(ServletConfig config) throws ServletException {
		System.out.println("Servlet Initialised");
	}

	@Override
	public void service(ServletRequest req, ServletResponse res) throws ServletException,
IOException {
		PrintWriter out = res.getWriter();
		out.print("<html><body>");
		out.print("<h1>Hello World</h1>");
		out.print("</html></body>");
		System.out.print("Service Called");
	}

}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
	<servlet>
		<servlet-name>First</servlet-name>
		<servlet-class>ServletDemo</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>First</servlet-name>
		<url-pattern>/home</url-pattern>
```

```
        </servlet-mapping>
</web-app>
```

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. We can 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.
2 important methods of HttpServlet Class -

1. protected void doGet(HttpServletRequest req, HttpServletResponse res) handles the GET request. It is invoked by the web container.
2. protected void doPost(HttpServletRequest req, HttpServletResponse res) handles the POST request. It is invoked by the web container.

Ex:

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

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/httpdemo")
public class HttpDemo extends HttpServlet {

        @Override
        public void doGet(HttpServletRequest req, HttpServletResponse res)throws IOException,
ServletException{
                PrintWriter out = res.getWriter();
                out.print("<html><body>");
                out.print("<h1>Servlet Created Form HttpServlet</h1>");
                out.print("</body></html>");
                System.out.print("Service Called");
```

```
        }
}
```

Deployment Descriptor

Java web applications use an XML file named web.xml is known as the deployment descriptor. It resides in the WEB-INF directory of the app. Deployment descriptor maps the URL to the appropriate Servlet class. <web-app> is the root tag of the file and several other tags are used to perform the mapping. Deployment descriptor also can be used to define the welcome file list, authenticated routes etc.

Ex:

```
<web-app>
        <welcome-file-list>
                <welcome-file>index.html</welcome-file>
        </welcome-file-list>
        <servlet>
                <servlet-name>LoginPage</servlet-name>
                <servlet-class>Login</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>LoginPage</servlet-name>
                <url-pattern>/login</url-pattern>
        </servlet-mapping>
</web-app>
```

HTTP Request Methods:

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

1. GET - The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
2. HEAD - The HEAD method asks for a response identical to a GET request, but without the response body.

3. POST - The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.
4. PUT - The PUT method replaces all current representations of the target resource with the request payload.
5. DELETE - The DELETE method deletes the specified resource.
6. CONNECT - The CONNECT method establishes a tunnel to the server identified by the target resource.
7. OPTIONS - The OPTIONS method describes the communication options for the target resource.
8. TRACE - The TRACE method performs a message loop-back test along the path to the target resource.
9. PATCH - The PATCH method applies partial modifications to a resource.

HTTP Response Status Code

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped into five classes:

1. Informational responses (100–199)
2. Successful responses (200–299)
3. Redirection messages (300–399)
4. Client error responses (400–499)
5. Server error responses (500–599)

Status Code	Meaning
200	Ok
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
502	Bad Gateway
503	Service Unavailable

Access Form Data

Generally more reliable method of passing information to a backend program is the POST method. The Servlet class inherited from HttpServlet class handles this type of request with the help of the **doPost()** method. But if for is sending the data with the help of GET method then we have to override the **doGet()** method.
Servlets handles form data parsing automatically using the following methods depending on the situation −

- getParameter() − You call request.getParameter() method to get the value of a form parameter.
- getParameterValues() − Call this method if the parameter appears more than once and returns multiple values, for example, checkbox.

Ex 1 -Access form data

form.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Registration Form</title>
</head>
<body>

<h1>Registration Form</h1>
<form action="formservlet" method="post">
Name<br>
<input type="text" name="sname"><br><br>
Mobile<br>
<input type="text" name="mobile"><br><br>
Email<br>
<input type="email" name="email"><br><br>
Gender
<input type="radio" name="gender" value="Male">Male
<input type="radio" name="gender" value="Female">Female
<br><br>
Address<br>
<textarea rows="5" cols="22" name="address"></textarea><br><br>
Heighest Qualification
<select name="hq">
	<option value="B.Tech">B.Tech</option>
	<option value="H.S.">H.S.</option>
	<option value="10th">10th</option>
</select>
```

```
<br><br>
Language Known
<input type="checkbox" name="lk" value="English">English
<input type="checkbox" name="lk" value="Hindi">Hindi
<input type="checkbox" name="lk" value="Odiya">Odiya
<br><br>
Password<br>
<input type="password" name="password">
<br><br>
<input type="submit" value="Register">
</form>
</body>
</html>
```

FormData.java

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/formservlet")
public class FormData extends HttpServlet{

	private static final long serialVersionUID = 1L;

	@Override
	public void doPost(HttpServletRequest request, HttpServletResponse response)throws ServletException, IOException{
		String name = request.getParameter("sname");
		//long mob = Integer.parseInt(request.getParameter("mobile"));
		String mob = request.getParameter("mobile");
		String email = request.getParameter("email");
		String gender = request.getParameter("gender");
		String address = request.getParameter("address");
		String hq = request.getParameter("hq");
		String langs[] = request.getParameterValues("lk");
		String pass = request.getParameter("password");

		String lang = "";
		for(int i = 0; i<langs.length; i++) {
			lang += langs[i];
		}

		PrintWriter out = response.getWriter();
		out.println("<html><body>");
		out.println("Name: "+name);
```

```
                out.println("<br>Mobile: "+mob);
                out.println("<br>Email: "+email);
                out.println("<br>Gender: "+gender);
                out.println("<br>Address: "+address);
                out.println("<br>Heighest Qualification: "+hq);
                out.println("<br>Languages: "+lang);
                out.println("<br>Password: "+pass);

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

}
```

Ex 2 -Calculator

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
        <h1>Calculator</h1>
        <form action="calculator" method="post">
                <input type="text" name="fn" placeholder="First Number">
                <br>
                <select name="op">
                        <option value="+">+</option>
                        <option value="-">-</option>
                </select>
                <br>
                <input type="text" name="sn" placeholder="Second Number">
                <br>
                <input type="submit" value="Calculate">
        </form>
</body>
</html>
```

Calculator.java

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

```
@WebServlet("/calculator")
public class Calculator extends HttpServlet {
	private static final long serialVersionUID = 1L;

	protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
		int a = Integer.parseInt(request.getParameter("fn"));
		int b = Integer.parseInt(request.getParameter("sn"));
		String op = request.getParameter("op");

		PrintWriter out = response.getWriter();
		int res;
		out.println("<html><body>");

		switch(op) {
			case "+":
				res = a+b;
				out.println("Sum= "+res);
				break;
			case "-":
				res = a-b;
				out.println("Difference= "+res);
				break;

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

Servlet with JDBC:

Ex 1: Add a student

add.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1>Add Student Data</h1>
<form method="post" action="addstudent">
```

```
        <input type="text" name="roll" placeholder="Roll Number"><br><br>
        <input type="text" name="name" placeholder="Name"><br><br>
        <input type="text" name="mobile" placeholder="Mobile Number"><br><br>
        <input type="email" name="email" placeholder="Email"><br><br>
        <input type="submit" value="ADD">
</form>
</body>
</html>
```

AddStudent.java

```
import java.io.*;
import java.sql.*;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/addstudent")
public class AddStudent extends HttpServlet {
        private static final long serialVersionUID = 1L;

        Connection con = null;
        Statement smt = null;

        public void init() {
                try {
                        Class.forName("oracle.jdbc.driver.OracleDriver");
                        con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","admin");
                        smt = con.createStatement();
                }catch(ClassNotFoundException ce) {
                        System.out.println(ce);
                }catch(SQLException se) {
                        System.out.println(se);
                }
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
                int roll = Integer.parseInt(request.getParameter("roll"));
                String name = request.getParameter("name");
                long mob = Long.parseLong(request.getParameter("mobile"));
                String email = request.getParameter("email");
```

```
		PrintWriter out = response.getWriter();
		out.println("<html><body>");
		out.println(roll+" "+name+" "+mob+" "+email);

		String qry = "INSERT INTO std_info
VALUES("+roll+",'"+name+"',+"+mob+",'"+email+"')";
		try {
			int i = smt.executeUpdate(qry);
			if(i==1) {
				out.println("<br>1 row inserted");
			} else {
				out.println("<br>Something Wrong !!!");
			}
		}catch(Exception e) {
			System.out.println(e);
		}


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

	}

	public void destroy() {
		try {
			smt.close();
			con.close();
		}catch(Exception e) {
			System.out.println(e);
		}
	}

}
```

Ex 2: Display all Student Details

```
import java.io.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.servlet.ServletException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/students")
public class Students extends HttpServlet {
	private static final long serialVersionUID = 1L;

	Connection con = null;
	Statement smt = null;
	ResultSet rs = null;

	public void init() {
		try {
			Class.forName("oracle.jdbc.driver.OracleDriver");
			con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","admin");
			smt = con.createStatement();
		} catch (Exception e) {
			System.out.println(e);
		}
	}

	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		PrintWriter out = response.getWriter();
		out.println("<html><body>");

		out.println("<table border='1' width='50%' align='center'>" );

out.println("<tr><th>Roll</th><th>Name</th><th>Mobile</th><th>Email</th><th>Action</th></tr
>");
		try {
			String qry = "SELECT * FROM std_info";
			rs = smt.executeQuery(qry);
			while(rs.next()) {
				String url="details?roll="+rs.getInt("roll");

out.println("<tr><td>"+rs.getInt("roll")+"</td><td>"+rs.getString("name")+"</td><td>"+rs.getLong(
"mobile")+"</td><td>"+rs.getString("email")+"</td><td><a href="+url+">Details</a></td></tr>");
			}
		} catch (Exception e) {
			System.out.println(e);
		}
		out.println("</table>");
```

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

        public void destroy() {
            try {
                rs.close();
                smt.close();
                con.close();
            } catch (Exception e) {
                System.out.println(e);
            }
        }

}
```

Page Redirection:

Page redirection is a technique where the client is sent to a new location other than requested. Page redirection is generally used when a document moves to a new location or maybe because of load balancing.

The simplest way of redirecting a request to another page is using method sendRedirect() of response object. Following is the signature of this method −

public void HttpServletResponse.sendRedirect(String location) throws IOException

```
Ex: response.sendRedirect("students");
```

Session Tracking:

Session simply means a particular interval of time. Session Tracking is a way to maintain the state (data) of a user. It is also known as session management in servlet.

The HTTP protocol is stateless so we need to maintain the state using session tracking techniques. Each time user requests the server, the server treats the request as a new request. So we need to maintain the state of a user to recognize a particular user.

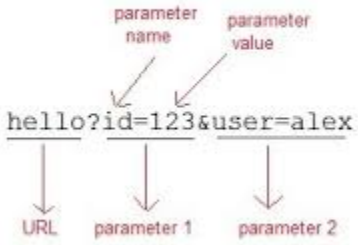
Session tracking can be done in different ways -

1. URL Rewriting
2. Session
3. Cookie
4. Hidden Form Field

URL Rewriting:

Url rewriting is a process of appending or modifying any url structure while loading a page. The request made by client is always a new request and the server can not identify whether the current request is send by a new client or the previous same client. Due to This property of HTTP protocol and Web Servers are called stateless. But many times we should know who is client in the processing request.

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:



A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.

Ex:

HTML File:

```
<!DOCTYPE html>
<html>
<body>
        <a href="urlrewrite?name=Jhon&roll=3">Click Me</a>
</body>
</html>
```

Servlet File:

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

```
@WebServlet("/urlrewrite")
public class UrlDemo extends HttpServlet {
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		String name = request.getParameter("name");
		String roll = request.getParameter("roll");

		PrintWriter out = response.getWriter();
		out.print("<html><body>");
		out.print(name+" "+roll);
		out.print("</body></html>");
	}
}
```

Session:

HttpSession 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. A session usually corresponds to one user, who may visit a site many times. An object of HttpSession can be used to perform two tasks:

- View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- Bind objects to sessions, allowing user information to persist across multiple user connections

How to get the HttpSession object?
The HttpServletRequest interface provides two methods to get the object of HttpSession:

- public HttpSession getSession(): Returns the current session associated with this request, or if the request does not have a session, creates one.
- public HttpSession getSession(boolean create): Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

- public String getId(): Returns a string containing the unique identifier value.

- public long getCreationTime(): Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- public long getLastAccessedTime(): Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- public void setAttribute(String name, Object value) This function uses the supplied name to tie an object to this session.
- public Object getAttribute(String name) This method returns the object in this session bound with the supplied name, or null if no object is bound with the name.
- public void invalidate(): Invalidates this session then unbinds any objects bound to it.

Ex:

Login.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Login Here</title>
</head>
<body>
	<form action="login" method="post">
		<input type="email" placeholder="Email" name="email"><br><br>
		<input type="password" name="password" placeholder="Password"><br><br>
		<input type="submit" value="Login">
	</form>
</body>
</html>
```

Login.java

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

@WebServlet("/login")
public class Login extends HttpServlet {
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		PrintWriter out = response.getWriter();
		out.print("<html><body>");
		String email = request.getParameter("email");
```

```
                String password = request.getParameter("password");
                if(email.equals("admin@gmail.com") && password.equals("admin")) {
                        HttpSession session = request.getSession(true);
                        session.setAttribute("email", email);
                        response.sendRedirect("welcome");
                }else {
                        out.print("Invalid email or password");
                }
                out.print(email+" "+password);
                out.print("</body></html>");
        }
}
```

Welcome.java

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

@WebServlet("/welcome")
public class Welcome extends HttpServlet {
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
                PrintWriter out = response.getWriter();
                HttpSession session = request.getSession(true);
                String email = (String)session.getAttribute("email");
                out.print("<html><body>");
                out.print("<h1>Welcome, "+email+"</h1>");
                out.print("<a href=logout>Logout</a>");
                out.print("</body></html>");
        }
}
```

Logout.java

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

@WebServlet("/logout")
public class Logout extends HttpServlet {
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

```
            HttpSession session = request.getSession(true);
            session.invalidate();
            response.sendRedirect("login.html");
        }

}
```

Cookie:

Cookies are text files stored on the client's computer and they are kept for various information tracking purposes. A cookie is persisted between multiple client requests. Java Servlets transparently supports HTTP cookies.

There are 2 types of cookies in servlets.

- Non-persistent cookie: It is valid for a single session only. It is removed each time when the user closes the browser.
- Persistent cookie: It is valid for multiple session. It is not removed each time when the user closes the browser. It is removed only if the user logout or signout.

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

- Cookie() constructs a cookie.
- Cookie(String name, String value)constructs a cookie with a specified name and value.
- public void setMaxAge(int expiry) Sets the maximum age of the cookie in seconds.
- public String getName() Returns the name of the cookie. The name cannot be changed after creation.
- public String getValue() Returns the value of the cookie.
- public void setName(String name)changes the name of the cookie.
- public void setValue(String value) changes the value of the cookie.
- public void addCookie(Cookie ck):method of HttpServletResponse interface is used to add cookie in response object.
- public Cookie[] getCookies():method of HttpServletRequest interface is used to return all the cookies from the browser.

Ex:

basic-info.html

```
<!DOCTYPE html>
```

```
<html>
<body>
<h1>Basic Info</h1>
<form action="binfo" method="post">
	<input type="text" name="name" placeholder="Name"><br><br>
	<input type="email" name="email" placeholder="Email"><br><br>
	<input type="submit" value="Next">
</form>
</body>
</html>
```

BasicInfo.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.*;

@WebServlet("/binfo")
public class BasicInfo extends HttpServlet {
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		String name = request.getParameter("name");
		String email = request.getParameter("email");

		Cookie cName = new Cookie("name",name);
		Cookie cEmail = new Cookie("email", email);

		cName.setMaxAge(365*24*60*60);

		response.addCookie(cName);
		response.addCookie(cEmail);

		response.sendRedirect("acc-info.html");
	}
}
```

acc-info.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h2>Academic Info</h2>
<form action="ainfo" method="post">
	<input type="text" name="ten" placeholder="10 Score"><br><br>
	<input type="text" name="twelve" placeholder="12 Score"><br><br>
	<input type="submit" value="Next">
</form>
</body>
</html>
```

AccInfo.java

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

@WebServlet("/ainfo")
public class AccInfo extends HttpServlet {
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		String ten = request.getParameter("ten");
		String twelve = request.getParameter("twelve");

		Cookie cTen = new Cookie("ten", ten);
		Cookie cTwelve = new Cookie("twelve", twelve);

		response.addCookie(cTen);
		response.addCookie(cTwelve);

		response.sendRedirect("details");
```

```
	}
}
```

Details.java

```
import java.io.*;

import javax.servlet.*
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/details")
public class Details extends HttpServlet {
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		Cookie details[] = request.getCookies();
		PrintWriter out = response.getWriter();
		out.print("<html><body>");
		for(int i=0;i<details.length;i++) {
			out.print(details[i].getName()+ " "+details[i].getValue()+"
"+details[i].getMaxAge()+"<br>");
		}
		out.print("<a href=destroy>Destroy Cookie</a>");
		out.print("</body></html>");
	}
}
```

DestroyCookie.java

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

@WebServlet("/destroy")
public class Destroy extends HttpServlet {
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		Cookie details[] = request.getCookies();
```

```
                PrintWriter out = response.getWriter();
                out.print("<html><body>");
                for(int i=0;i<details.length;i++) {
                        details[i].setMaxAge(0);
                        response.addCookie(details[i]);
                }
                out.print("<a href=details>Detaisl</a>");
                out.print("</body></html>");
        }
}
```

Date Handling in Servlet

Passing Date from an HTML form to a servlet to an SQL database

Step 1: To convert String to java.util.Date, use SimpleDateFormat

```
String stringDate = "2011-03-22";
Date utilDate = new SimpleDateFormat("yyyy-MM-dd").parse(stringDate);
```

Step 2: To convert java.util.Date to java.sql.Date, just use its constructor.

```
java.sql.Date sqlDate = new java.sql.Date(utilDate);
```

Note: SimpleDateFormat throws ParseException. So, you need to handle it in your program.

File Upload and Display

A Servlet can be used with an HTML form tag to allow users to upload files to the server. An uploaded file could be a text file or image file or any document.

Following are the important points to be noted down −

- The form method attribute should be set to POST method and GET method can not be used
- The form enctype attribute should be set to multipart/form-data.

Servlet Specs 3.0 provided additional support for uploading files to server and we don't have to depend on any third party APIs for this. We need to annotate File Upload handler servlet with MultipartConfig annotation to handle multipart/form-data requests that is used for uploading file to server.

MultipartConfig annotation has following attributes:

- fileSizeThreshold: We can specify the size threshold after which the file will be written to disk. The size value is in bytes, so 1024*1024*10 is 10 MB.
- location: Directory where files will be stored by default, it's default value is "".
- maxFileSize: Maximum size allowed to upload a file, it's value is provided in bytes. It's default value is -1L means unlimited.
- maxRequestSize: Maximum size allowed for multipart/form-data request. Default value is -1L that means unlimited.

Part interface represents a part or form item that was received within a multipart/form-data POST request. Some important methods are getInputStream(), write(String fileName) that we can use to read and write file.

Ex:

file.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>File Uplaod Example</title>
</head>
<body>
	<form action="uploadImage" method="post" enctype="multipart/form-data">
		<label>Name</label>
		<input type="text" name="name"><br><br>
		<label>Choose your image</label>
		<input type="file" name="dp" accept="image/*">
		<br><br>
		<input type="submit">
	</form>

</body>
</html>
```

UplaodImage.java

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/uploadImage")
@MultipartConfig
public class UploadImage extends HttpServlet {
```

```
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		String name = request.getParameter("name");

//		Getting file from the form
		Part filePart = request.getPart("dp");

//		Getting the file name
		String fileName = filePart.getSubmittedFileName();

		System.out.println(name+" "+fileName);

//		Prepare the path of the folder where the image will be uploaded
		String uploadDir = "images";
		String serverPath = request.getServletContext().getRealPath(""); // Get Server
path
		String uploadDirPath = serverPath+File.separator+uploadDir;

		System.out.println(serverPath);

//		Create the folder if does not exists
		File dir = new File(uploadDirPath);
		if(!dir.exists()) {
			dir.mkdir();
		}

//		Upload the file
		String fileUploadDirPath = uploadDirPath+File.separator+fileName;
		filePart.write(fileUploadDirPath);
		System.out.println("File Uploaded");

		String imgPath = uploadDir+"/"+fileName;

		PrintWriter out = response.getWriter();
		out.print("<html><body>");
		out.print("<img src="+imgPath+">");
		out.print("</body></html>");

	}
}
```

Packaging Applications

A Java EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (.jar) file with a .war or .ear extension. Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE JAR, WAR, or EAR files.

The WAR file (Web Application Resource or Web Application ARchive) is a container for JAR files, JavaServer Pages, Java Servlets, Java classes, XML files, tag libraries, static sites (HTML and associated files), and other resources that make up an online application. A file entitled web.xml is located in the /WEB-INF directory of the WAR file, and it describes the structure of the online application. The web.xml file isn't technically essential if the online application is only providing JSP files. If the online apps utilize servlets, the servlet container looks at web.xml to figure out which servlet a URL request should go to.
An EAR file contains Java EE modules and, optionally, deployment descriptors. A deployment descriptor, an XML document with a .xml extension, describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

Java Internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n."

An internationalized program has the following characteristics:

- With the addition of localization data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.

- Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- It can be localized quickly.

JSP

JSP:

Jakarta Server Pages (JSP; formerly JavaServer Pages) technology, part of the Java EE 5, gives web and Java developers a simple yet powerful mechanism for creating a platform-independent web application that supports dynamic content.
The JSP technology:

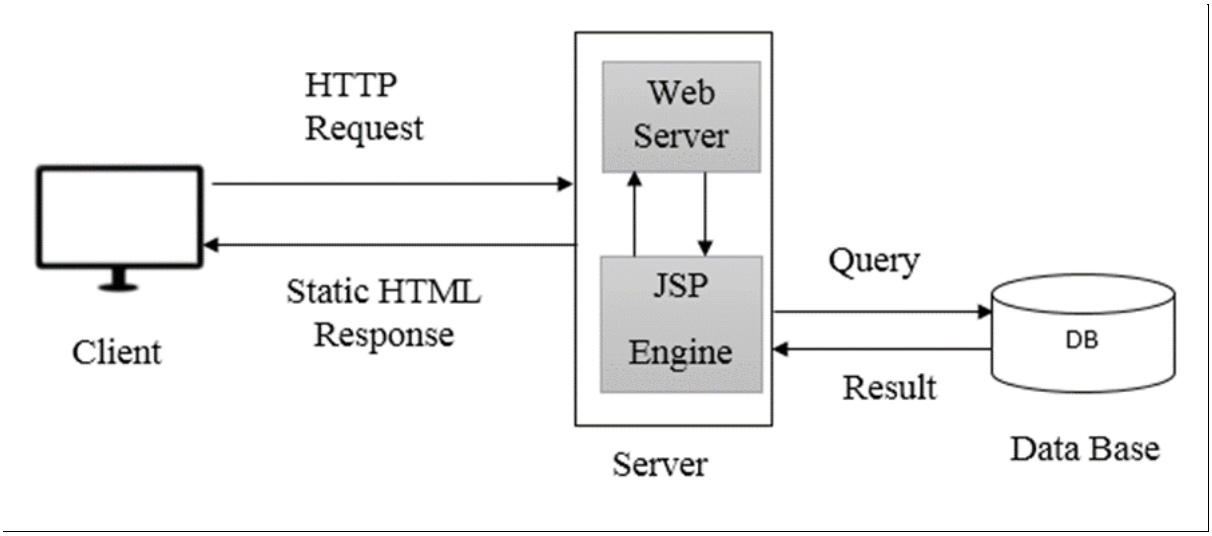
- Is a language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response.
- Allow creating web content with both static and dynamic components.
- Provides developers with the ability to access remote data via mechanisms such as EJB, RMI, and JDBC.
- Provides developers with the ability to encapsulate and separate program logic from the presentation logic.
- Makes available all the dynamic capabilities of Java Servlet Technology but provides a more natural approach to creating static content.

Advantages of JSP (Over Servlet):

- JSP is an interface on top of Servlets. In another way, we can say that JSPs are extension of servlets to minimize the effort of developers to write User Interfaces using Java programming.
- Implicit objects, predefined tags, expression language and Custom tags in JSP makes JSP development easy and reduce the code size.
- It is more convenient to write and to modify regular HTML than to have plenty of println statements that generate the HTML. JSP encapsulate and separate program logic from the presentation logic very easily.
- The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application. If JSP page is modified, we don't need to recompile and redeploy the project.

Architecture & Anatomy of JSP Page:

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



1. The user goes to a JSP page and makes the request via the internet in user's web browser.
2. 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.
3. If the JSP file has been called the first time then the JSP file will be loaded in the JSP engine and convert it into Java Equivalent Servlet (JES) file. Otherwise existing servlet will be initialized.
4. The JSP engine compiles the servlet into an executable class and forwards the original request to servlet engine, which generates the output in HTML format.
5. The webserver forwards the HTTP response to the client's browser as static HTML content.

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.

JSP life cycle:

A JSP page services requests as a servlet. Thus, the life cycle and many of the capabilities of JSP pages are determined by Java Servlet technology. Life cycle of JSP defines the different stages and lifecycle methods that a JSP page passes through from its creation to destruction. Since JSP architecture, a java enabled web server provides the mechanism that deals with both.
The following are the paths followed by a JSP −

- Translation of JSP page to Servlet
- Compilation of JSP page
- Class loading
- Instantiation
- Initialization
- Request processing / Execution
- JSP Clean-up

Translation of JSP page to Servlet :
This is the first step of the JSP life cycle. This translation phase deals with the Syntactic correctness of JSP. Here index.jsp file is translated to index.java.
Compilation of JSP page :

Here the generated java servlet file (index.java) is compiled to a class file (index.class).
Class Loading :
Servlet class which has been loaded from the JSP source is now loaded into the container.
Instantiation :
Here an instance of the class is generated. The container manages one or more instances by providing responses to requests.
Initialization :
jspInit() method is called only once during the life cycle immediately after the generation of Servlet instance from JSP. We generally initialize database connections, open files, and create lookup tables in the jspInit method.

```
public void jspInit(){
    //Initialization Code
}
```

JSP Execution:
This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed. Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.
The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows –

```
    void _jspService(HttpServletRequest request, HttpServletResponse response){
        // Service handling code
    }
```

JSP Cleanup :
In order to remove the JSP from the the container or to destroy the method for servlets jspDestroy()method is used.
This method is called once, if you need to perform any cleanup task like closing open files, releasing database connections jspDestroy() can be overridden.

```
public void jspDestroy() {
    // Your clean up code goes here.
}
```

JSP Scripting Elements:

1. JSP scriptlet Tag:

Syntax: <% code fragment %>
A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language. A scriptlet is a block of Java code spec that is executed at run time. A scriptlet can produce output passed using an output stream back to the client.
Variables declared in a scriptlet are local to a method in the JSP page implementation class i.e. they are visible only within their defining code block.

2. JSP Declaration Tag:

Syntax: <%! Declaration %>
A declaration is a java code block that holds one or more variables or methods that can be accessed in java code later in JSP file.
Methods can be defined only by declaration and not by scriptlets.
Variables declared in a declaration are instance variables of the JSP page implementation class.

3. Expression:

Syntax: <%= expression %>

A JSP expression element contains a scripting language expression that is evaluated, converted to a string and inserted where the expression appears in the JSP file. JSP expression element is explicitly used for output generation.

Ex:

```
Current time: <%= new java.util.Date() %>
Your hostname: <%= request.getRemoteHost() %>
```

Ex -1 (Dynamic web page using JSP tags and Different comment tags):

```
<!--HTML Comment Tag -->
<h1>This is My 1st JSP Page</h1>
<%--
        JSP Comment
--%>

<%!
/*Global Variable and Function declaration*/
int a=10;
int b=10;
int c;
public int add(){
        return a+b;
}
%>
<%
//Scriptlet tag
c=add();
%>
<h3>Sum= <%=c %><h3>
```

Output:

This is My 1st JSP Page

Sum= 20

Ex-2:

```
<html>
<head>
<title>JSP Expression Tag Example</title>
</head>
<body>
<b>
Today is: <%= new java.util.Date() %>
<br>
Your Session ID is: <%= session.getId() %>
</b>
</body>
</html>
```

Output:

Today is: Fri Mar 08 16:41:03 IST 2019

Your Session ID is: 96CBF65E8392868857ECF30AFCA1BD01

Implicit JSP Object:

JSP engine exposes a number of internal Java Objects to the developer. These objects do not need to be declared or initialized by the developer but are provided by the JSP engine in its implementation and execution. These objects are known as Implicit Object.
Implicit objects are only available in the _jspService() method and only available to scriptlets or expressions.
In JSP 9 implicit objects are there –

Object	**Class Name**
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
exception	java.lang.Throwable
out	javax.servlet.jsp.JspWriter
page	java.lang.Object
pageContext	javax.servlet.jsp.PageContext
request	javax.servlet.ServletRequest
response	javax.servlet.ServletResponse
session	javax.servlet.http.HttpSession

Request: The implicit request object represents the HTTP request. The request object provides access to all information associated with a request, including its source, the requested URL, and any header, cookies or parameters associated with the request.
Response: The Implicit Response object represent the response sent back to the browser as a result of processing the JSP page. Some of the tasks done by using response object are setting headers, cookies for the client and sending a redirect response to the client.
Out: The Out object represents the output stream of the JSP page. The content of the out object are displayed in the client browser. print () and println() method is used to display the content of the out object.
Session: HTTP is a stateless protocol. As far as Web server is concerned, each client's request is a new request. A session refers to the entire conversation between a client and a server. The session object allows accessing the client's session data, managed by the server. Sessions are created automatically even if there is no incoming session reference.
Application: The application object represents the application to which the JSP page belongs. The application object holds references to another object that more than one user may require access to such as a database connection pool.
Config: The configuration information contains initialization parameters, defined in the web application deployment descriptor.

PageContext: The PageContext object encapsulate the page context of a JSP page. The PageContext object implements methods for transferring control from the current page to another page.
Page: The Page object represents the JSP page itself and can be accessed using this reference. In other words, the page object is simply a synonym of 'this' keyword.
Exception: The exception object refers to the runtime exception that results in an error page being invoked.

-

Directive Tags:

JSP directive serve special processing information about the page to the JSP server. Directives specifies translation time instruction to JSP engine.
Syntax: <%@ directive attribute="value">
Multiple attributes can be specified separated by space and multiple value of each attribute can be placed as a key value pair separated by comma(,).
3 different directives are –

1. page directive
2. include directive
3. taglib directive

page directive:

page directive is used to specify any of a number of page-dependent attributes such as scripting language to use, import classes, setting the content type, class to extend and more.
Syntax:
<%@ page attribute1="value" attribute2="value1,value2" %>
Different attributes of page directive are –

Attribute	Default Value	Operation
language	java	Defines which scripting language the file use. <%@ page language="java" %>
extends	-	It defines parent class of generated servlet class. <%@ page extends="javax.servlet.http.HttpServlet" %>
import	-	It is used to import classes, interface of a package. <%@ page import= "java.io.*, java.sql.*" %>

info	-	It is used by the developer to add details such as author, version, copyright, date and so on. <%@ page info=”Message” %>
buffer	8 kb	Holds the data stream size through which the application will pass data to the browser. <%@ page buffer= “24kb” %>
autoFlash	true	It defines whether output should be flushed automatically when buffer is filled. <% page autoFlash= “true” %>
contentType	text/html	It sets the MIME type and character set of the JSP page. <%@ page contentType= “text/xml” %>
pageEncoding	ISO-8859-1	Defines the character set of the current page.
session	true	Allows the page to participate in an HTTP session. If the value is true the page will use the existing session if one exists, otherwise a new session is created. <%@ page session= “true” %>
errorPage	-	Specifies a jsp page, Servlet or Static page or URI path of the page to redirect the current page if any exception is thrown by code specified in the current page. <%page errorPage= “ErrorHandler.jsp” %>
isErrorPage	false	Indicates whether or not the current page can act as the error page for another jsp page. <%page isErrorPage= “true” %>
isThreadSafe	true	State whether JSP engine can pass multiple requests simultaneously.
isScriptingEnabled	-	Determines other scripting elements are allowed or not.

isElIgnored	false	Specifies whether or not the EL (Expression Language) expression within the JSP page will be ignored. ${pram.name};

include directive:

Include directive is used to include an HTML,JSP or Servlet file into a JSP file.
This directive tells the container to merge the content of other external files with the current JSP during the translation phase.
<%@ include file="FileName/RelativePath" %>
file attribute specifies the path of the file or the name of the file that is to be included.

taglib directive:

taglib is a collection of custom tags that can be used by the page.
<%@ taglib uri="TagLibraryURI" prefix= "TagPrefix"%>
uri identifies the tag library descriptor.
prefix attribute informs a container what bits of markup are custom actions.

Example (page and include directive):

head.jsp

```
<%@page language="java" info="Demonstration of include directive" import="java.util.Date"
%>
<html>
<body>
<center>
<h1>ABC Institute of Technology</h1>
<h3>Learn Smart</h3>
<hr>
<a href="home.jsp">HOME</a>
<a href="login.jsp">LOGIN</a>
<a href="#">REGISTRATION</a>
<a href="#">ABOUT</a>
<%=new Date() %>
<hr>
</center>
</body>
</html>
```

home.jsp

```
<%@include file="head.jsp" %>
<center><img src="image.jpg" height="500" width="950"></center>
```

login.jsp

```
<%@include file="head.jsp" %>
<html>
<body>
<form action="second.jsp" method="post">
```

```
<table align="center">
<tr><td>Username:</td><td><input type="text" name="username"></td></tr>
<tr><td>Password:</td><td><input type="password" name="password"></td></tr>
<tr><td><input type="submit" value="Login"></td></tr>
</table>
</form>
</body>
</html>
```

JSP Action Tags:

JSP actions are XML syntax tags used to control servlet engine.The Action tags are used to control the flow between pages and to use Java Bean. We can insert file, forward one page to another page or we can create HTML page for java plugin. Jsp action tags are listed in the table below:

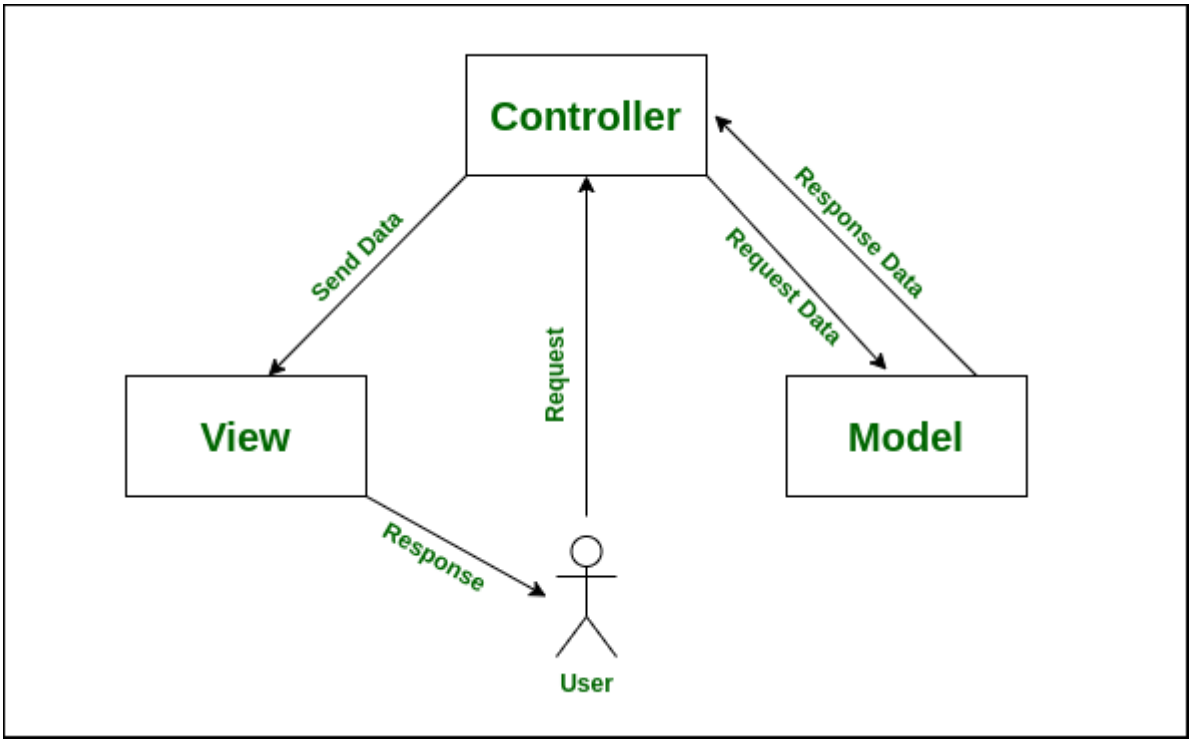
Action Elements	Operation
<jsp:include>	This action allows a static or dynamic resource to be included in the current JSP at request time. It allows inserting files into the page being generated. Flush attribute is optional and determines whether the output buffer should be flushed before including the file. <jsp:include page="Filename" flush="true/false"/>
<jsp:forward>	This action tag forward the control form one page to another page resides on the local server. <jsp:forward page="URL" />
<jsp:param>	This element is used to specify additional request parameters for the target resource by appending parameters to the request object. This element is used in the body of a <jsp:forward> or <jsp:include> <jsp:forward page="URL" > <jsp:param name="ParameterName" value="ParamaterValue" /> </jsp:forward>
<jsp:useBean>	A <jsp:useBean> element makes a Java Bean available to a JSP page. <jsp:useBean id="BeanVariableName" class="PackageName" beanName="ClassName">

<jsp:setProperty>	Use to set the variable value of the bean. <jsp:useBean id="obj" class="PN.ClassName"> <jsp:setProperty name="obj" property="variableName" value="VariableValue" /> </jsp:useBean>
<jsp:getProperty>	Use to get the variable value of the bean. <jsp:useBean id="obj" class="PN.ClassName"> <jsp:getProperty name="obj" property="variableName" /> </jsp:useBean>
<jsp:plugin>	It is used to add a component object like applet within JSP.
<jsp:fallback>	It is used to print the message if the plugin is not supported.

MVC:

Model-View-Controller (MVC) is a pattern used in software engineering to separate the application logic from the user interface. As the name implies, the MVC pattern has three layers.
The Model defines the business layer of the application, the Controller manages the flow of the application, and the View defines the presentation layer of the application.

- It separates the presentation layer from the business layer
- The Controller performs the action of invoking the Model and sending data to View
- The Model is not even aware that it is used by some web application or a desktop application



Model Layer:
This is the data layer which contains business logic and data access logic of the system, and also represents the state of the application.
It's independent of the presentation layer, the controller fetches the data from the Model layer and sends it to the View layer.
View Layer:
This layer represents the output of the application, usually some form of UI. The presentation layer is used to display the Model data fetched by the Controller. These pages are basically JSP & HTML pages.
Controller Layer:
Controller layer acts as an interface between View and Model. It receives requests from the View layer and processes them, including the necessary validations.
The requests are further sent to the Model layer for data processing, and once they are processed, the data is sent back to the Controller and then displayed on the View.
Advantages of MVC:

1. Faster development process: MVC supports rapid and parallel development. With MVC, one programmer can work on the view while others can work on the controller to create business logic of the web application. The application developed using MVC can be three times faster than an application developed using other development patterns.
2. Ability to provide multiple views: In the MVC Model, you can create multiple views for a model. Code duplication is very limited in MVC because it separates data and business logic from the display.
3. Support for asynchronous technique: MVC also supports asynchronous technique, which helps developers to develop an application that loads very fast.
4. Modification does not affect the entire model: Modification does not affect the entire model because model part does not depend on the views part. Therefore, any changes in the Model will not affect the entire architecture.

5. MVC model returns the data without formatting: MVC pattern returns data without applying any formatting so the same components can be used and called for use with any interface.

JSP Database Access:

All web applications usually interact with different types of databases. Databases are basically used to store various types of information for different purposes. There are also different types of databases used in the industry nowadays. But this is very important that the relational databases are by far the most widely used.
In a JSP page, the connection to the database is established once and it will be closed at the end of the communication. By using the JSP declaration tag inside the jspInit() method a connection is established. The connection is closed in the jspDestroy() method by using the declaration tag.
Communication with the database and all the data transfer from or to the database is done inside the scriptlet tag.
JSP use the standard JSP API to communicate with the Database. Connection interface is used to establish the connection, DriverManager class is used to load the driver class, Statement object is used to pass the query to the database and ResultSet object is used to handle the data.
Following example shows how the database connection is established:
add.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Add a Student</title>
</head>
<body>
<h1>Add Student Data</h1>
<form method="post" action="addStudentTwo.jsp">
	<input type="text" name="roll" placeholder="Roll Number"><br><br>
	<input type="text" name="name" placeholder="Name"><br><br>
	<input type="text" name="mobile" placeholder="Mobile Number"><br><br>
	<input type="email" name="email" placeholder="Email"><br><br>
	<input type="submit" value="ADD">
</form>
</body>
</html>
```

addStudentTwo.jsp

```
<%@ page language="java" import="java.sql.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%!
	Connection con = null;
	Statement smt = null;
	public void jspInit(){
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
			con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","admin");
			smt = con.createStatement();
		}catch(Exception ee){
			ee.printStackTrace();
		}
	}

	public void jspDestroy(){
		try{
			smt.close();
			con.close();
		}catch(Exception ee){
			ee.printStackTrace();
		}
	}
%>
<%
	int roll = Integer.parseInt(request.getParameter("roll"));
	String name = request.getParameter("name");
	long mob = Long.parseLong(request.getParameter("mobile"));
	String email = request.getParameter("email");
	String qry = "INSERT INTO std_info
VALUES("+roll+",'"+name+"',+"+mob+",'"+email+"')";
	int i = smt.executeUpdate(qry);
	if(i>0){
		%>
			<h1>One Row Inserted</h1>
			<a href="add.html">Go Back</a>
		<%
	}else{
		%>
			<h1>Error !!!</h1>
			<a href="add.html">Go Back</a>
		<%

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

As exception is an implicit object in jsp we can perform the database operation as follows. But this way is **not recommended** as connection objects will be created every time service method is called.

```
<%@ page language="java" import="java.sql.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	int roll = Integer.parseInt(request.getParameter("roll"));
	String name = request.getParameter("name");
	long mob = Long.parseLong(request.getParameter("mobile"));
```

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

	Class.forName("oracle.jdbc.driver.OracleDriver");
	Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","admin");
	Statement smt = con.createStatement();
	String qry = "INSERT INTO std_info
VALUES("+roll+",'"+name+"',+"+mob+",'"+email+"')";
	int i = smt.executeUpdate(qry);
	if(i>0){
		%>
			<h1>One Row Inserted</h1>
			<a href="add.html">Go Back</a>
		<%
	}else{
		%>
			<h1>Error !!!</h1>
			<a href="add.html">Go Back</a>
		<%

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

Display Student Records:

```
<%@ page language="java" import="java.sql.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%!
	Connection con = null;
	Statement smt = null;
	ResultSet rs= null;
	public void jspInit(){
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");
			con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","admin");
			smt = con.createStatement();
		}catch(Exception ee){
			ee.printStackTrace();
		}
	}

	public void jspDestroy(){
		try{
			smt.close();
			con.close();
		}catch(Exception ee){
			ee.printStackTrace();
		}
```

```
	}
%>
<%
	String qry = "SELECT * FROM std_info";
	rs = smt.executeQuery(qry);
	%>
	<table border="1" cellpadding="8" rules="all">
	<tr><th>Roll</th><th>Name</th><th>Mobile</th><th>Email</th></tr>
	<%
	while(rs.next()){
		%>
			<tr>
				<td> <%= rs.getInt("roll") %> </td>
				<td> <%= rs.getString("name") %> </td>
				<td> <%= rs.getLong("mobile") %> </td>
				<td> <%= rs.getString("email") %> </td>
			</tr>
		<%
	}
	%>
		</table>
	<%
%>
</body>
</html>
```

-

Session Tracking:

HTTP is a stateless protocol. Which means, the protocol cannot remember prior connections and thus cannot distinguish one visitor request from that of another.
In contrast, FTP is a stateful protocol. Connections are not opened and closed with every request. After the initial login, the FTP server maintain the visitor's credentials throughout the session.
The lack of state prevents a web server from uniquely identifying each visitor. This limitation prevents implementing visitor defined preferences, as the web server cannot distinguish one visitor from other.
Because of HTTP's stateless nature, a visitor who navigate through a web site cannot be tracked. Every request from information from the visitor's browser uses a new connection from anonymous client.
Web application have used several technologies to get around the stateless options of HTTP:

1. Hidden Form Field
2. URL Rewriting
3. Cookie
4. Session

Hidden Form Field:

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

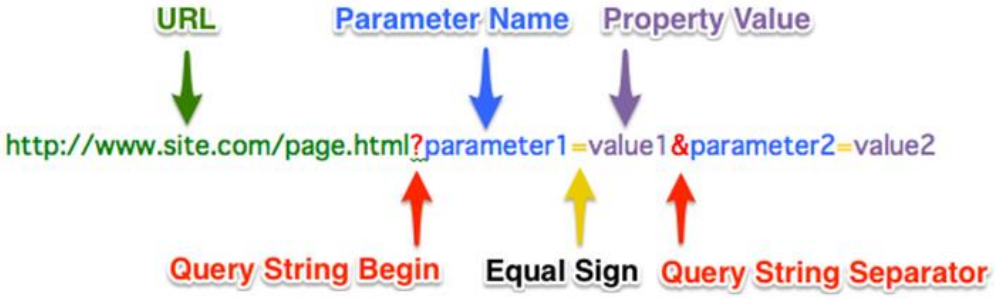
```
<html>
<body>
<center>
<br><br><br>
<form action="RegData.jsp" method="post">
Name: <input type="text" name="name"><br><br>
Reg No: <input type="text" name="reg"><br><br>
<input type="hidden" name="page" value="Reg.html">
<input type="submit" value="Registration">
</form>
</center>
</body>
</html>
```

RegData.jsp:

```
<h1>Name:<%=request.getParameter("name") %>
<br>
Reg No: <%=request.getParameter("reg") %>
<br>
You are forwarded from <u><%=request.getParameter("page") %></u> page
</h1>
```

URL Rewriting:

URL rewriting is a process of appending or modifying any url structure while loading a page. The request made by the client is always a new request and the server cannot identify whether the current request is sent by a new client or the previous same client.
The requested data is retrieved by the web server through the request parameter. Here the data is passed by appending the query string with the URL in a key and value format.



HTML Page:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<a href="urljsp.jsp?fname=Jhon&lname=Smith">Next</a>
</body>
</html>
```

urljsp.jsp:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	String fname = request.getParameter("fname");
	String lname = request.getParameter("lname");
%>
<h1>Welcome, <%=fname+" "+lname %></h1>
</body>
</html>
```

Session:

HttpSession 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. A session usually corresponds to one user, who may visit a site many times. An object of HttpSession can be used to perform two tasks:

- View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- Bind objects to sessions, allowing user information to persist across multiple user connections

How to get the HttpSession object?
The HttpServletRequest interface provides two methods to get the object of HttpSession:

- public HttpSession getSession(): Returns the current session associated with this request, or if the request does not have a session, creates one.
- public HttpSession getSession(boolean create): Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

- public String getId(): Returns a string containing the unique identifier value.
- public long getCreationTime(): Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- public long getLastAccessedTime(): Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.

- public void setAttribute(String name, Object value) This function uses the supplied name to tie an object to this session.
- public Object getAttribute(String name) This method returns the object in this session bound with the supplied name, or null if no object is bound with the name.
- public void invalidate(): Invalidates this session then unbinds any objects bound to it.

Ex:

Html File:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="sessionnext.jsp" method="post">
	<input type="text" name="email" placeholder="Email"><br><br>
	<input type="password" name="password" placeholder="Password"><br><br>
	<input type="submit" value="Next">
</form>
</body>
</html>
```

sessionnext.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	String email = request.getParameter("email");
	String password = request.getParameter("password");

	session.setAttribute("username", email);
	session.setAttribute("pass", password);

	response.sendRedirect("sessiondisplay.jsp");
%>
</body>
</html>
```

sessiondisplay.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	//String email = (String)session.getAttribute("username");
```

```
%>
<h1>Email: <%= session.getAttribute("username") %></h1>
<h1>Password: <%= session.getAttribute("pass") %></h1>
</body>
</html>
```

Cookie:

Cookies are text files stored on the client's computer and they are kept for various information tracking purposes. A cookie is persisted between multiple client requests. Java Servlets transparently supports HTTP cookies.
There are 2 types of cookies in servlets.

- Non-persistent cookie: It is valid for a single session only. It is removed each time when the user closes the browser.
- Persistent cookie: It is valid for multiple session. It is not removed each time when the user closes the browser. It is removed only if the user logout or signout.

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

- Cookie() constructs a cookie.
- Cookie(String name, String value) constructs a cookie with a specified name and value.
- public void setMaxAge(int expiry) Sets the maximum age of the cookie in seconds.
- public String getName() Returns the name of the cookie. The name cannot be changed after creation.
- public String getValue() Returns the value of the cookie.
- public void setName(String name) changes the name of the cookie.
- public void setValue(String value) changes the value of the cookie.
- public void addCookie(Cookie ck):method of HttpServletResponse interface is used to add cookie in response object.
- public Cookie[] getCookies():method of HttpServletRequest interface is used to return all the cookies from the browser.

Ex:
Html File:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="cookienext.jsp" method="post">
	<input type="text" name="email" placeholder="Email"><br><br>
	<input type="password" name="password" placeholder="Password"><br><br>
```

```
	<input type="submit" value="Next">
</form>
</body>
</html>
```

cookienext.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	String email = request.getParameter("email");
	String password = request.getParameter("password");

	Cookie ckEmail = new Cookie("uname", email);
	Cookie ckPass = new Cookie("pass", password);

	response.addCookie(ckEmail);
	response.addCookie(ckPass);

	response.sendRedirect("cookiedisp.jsp");
%>
</body>
</html>
```

cookiedisp.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
	Cookie ck[] = request.getCookies();
	for(int i=0;i<ck.length; i++){
		out.print(ck[i].getName()+" "+ck[i].getValue()+"<br>");
	}
%>
</body>
</html>
```

Pagination

Pagination is used to break the records into more convenient readable bits, both for the user and the processor, as a single page having multiple records may take time to load and can be tiring for the user to go through.
User can use links such as "next", "previous", and page numbers to navigate between pages that display one page of results at a time.

Note:
In MySQL we can use LIMIT & OFFSET together to limit number of records.
Ex: SELECT * FROM students LIMIT 25 OFFSET 10
In Oracle 12c Onwards we can use OFFSET and FETCH NEXT n ROWS to do the same
Ex: SELECT * FROM students OFFSET 10 ROWS FETCH NEXT 25 ROWS ONLY

JSTL

The Jakarta Standard Tag Library (JSTL; formerly JSP Standard Tag Library) is a component of the Java EE Web application development platform.
It extends the JSP specification by adding a tag library of JSP tags for common tasks, such as –

- XML data processing
- conditional execution
- database access
- Loops
- internationalization.

JSTL allows programming JSP pages using tags, rather than the scriptlets. JSTL was introduced to allow the JSP developer to create web applications using tags rather than the Java code.
Advantages:

- Neat and Clean Coding approach: Since scripts confuse developers a bit, their usage can be replaced with JSTL, making coding easier.

- Standard Tag: JSTL provides an affluent layer of manageable functionality in your JSP pages. This makes the JSP code easier to understand.

- Automatic Support of Javabeans Introspection: It is also beneficial to use JSTL over JSP's normal scriptlet. The JSTL Expression language is efficient in handling JavaBean code. Downcasting of objects is not required, which can also be retrieved as scoped attributes. JSP scriptlet generates a range of complicated codes, but JSTL has cut down to ease that.
- Easy for computers to understand: There are applications like Dreamweaver and front page, which generate more and more HTML code. In such tools, the HTML code gets mixed with the JSP scriptlet code in the back end of the development. However, since JSTL is expressed as an XML yielding tag, it becomes easy for HTML generation in parsing these JSTL codes without many hurdles within your document code.

- Easier for humans to read: JSTL is XML based custom tags similar to that of HTML. Hence, this makes it easy for developers to read and understand.

JSTL Tags:

Area	Subfunction	Prefix	URI
Core	Variable support	c	http://java.sun.com/jsp/jstl/core
	Flow control		
	URL management		
	Miscellaneous		
XML	XML Processing	x	http://java.sun.com/jsp/jstl/xml
	Transformation		
I18N	Locale	fmt	http://java.sun.com/jsp/jstl/fmt
	Message formatting		
	Number and date formatting		
Database	SQL	sql	http://java.sun.com/jsp/jstl/sql
Functions	Collection length	fn	http://java.sun.com/jsp/jstl/functions

How to use?
Syntax:
<%@ taglib uri="URI of the Library" prefix="Prefix of the library" %>
Ex:
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
Example 1:
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>

```
<c:forEach var="i" begin="1" end="20" step="1">
        <c:out value="${i}" />
</c:forEach>
```
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Example 2:

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

<c:set var="Date" value="<%=new java.util.Date()%>" />
<fmt:formatDate type="time" value="${Date}" />  <br>
<fmt:formatDate type="date" value="${Date}" />  <br>
<fmt:formatDate type="both" dateStyle="medium" timeStyle="medium"  value="${Date}" />
```

Output:

12:20:28 PM
Aug 21, 2019
Aug 21, 2019 12:20:28 PM