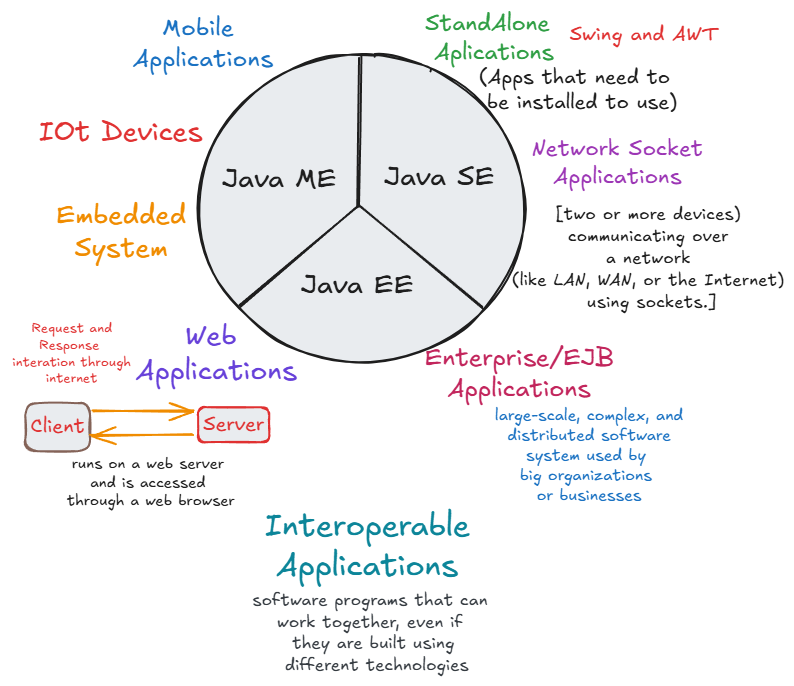
**Servlets and JSP's**

****

**Web Based Application / Web Application:**

An application which runs inside a distant system called as **server** and which can be accessed using **standard web browsers by clients** is called as web application or web-based application.

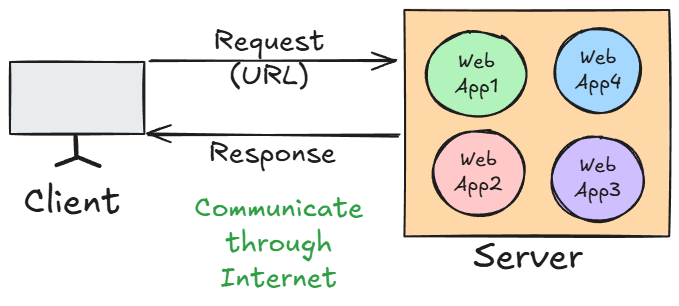
**Web resource:**

Web resource is the Data or the information present on the web or internet. It can be classified in two types as **static web resource and dynamic web resource.**

**1.Static web resource:** If the data or the information associated with the web resource will not change based on some inputs or conditions then that web resource is called as static web resource.

**2.Dynamic web resource**: If the data or the information associated with the web resource will change based on some inputs or conditions is called as dynamic web resource.

**Client-Server Interaction**

****

* **Request Side Parameters**

These parameters are sent **from the client (browser or any testing tool)** to the **server**.

**1. URL (Uniform Resource Locator)**

* The address used to access a particular resource on the server.
* **Example:**

**https://flipkart.com/products?id=5**

* **Parts:**
  + **https** → Protocol
  + **flipkart.com** → Domain name
  + **/products** → Path to the resource
  + **?id=5** → Query parameter (used to send data in GET request)

**2. Method Type**

* HTTP method used to send the request.
* **Common Types:**
  + **GET** → Fetch data
  + **POST** → Send new data
  + **PUT** → Update existing data
  + **DELETE** → Delete data

**3. Data**

* Data sent by the client to the server (mostly in POST, PUT).
* **Can be sent as:**
  + Query Parameters (e.g., ?id=1&name=John)
  + Form Data
  + JSON/XML payload in the request body
* **Example**

{

"username": "john",

"password": "abc123"

}

* **Response Side Parameters**

These parameters are sent **from the server back to the client** as a result of the request.

**1. Status Code**

* A 3-digit code representing the result of the request.
* **Common Status Codes:**
  + **200 OK** → Request successful
  + **201 Created** → Resource created
  + **400 Bad Request** → Invalid input
  + **401 Unauthorized** → Authentication required
  + **404 Not Found** → Resource doesn’t exist
  + **500 Internal Server Error** → Server crashed
  + **405 – Method Not Allowed** → Invalid Method type

**2. Content Type**

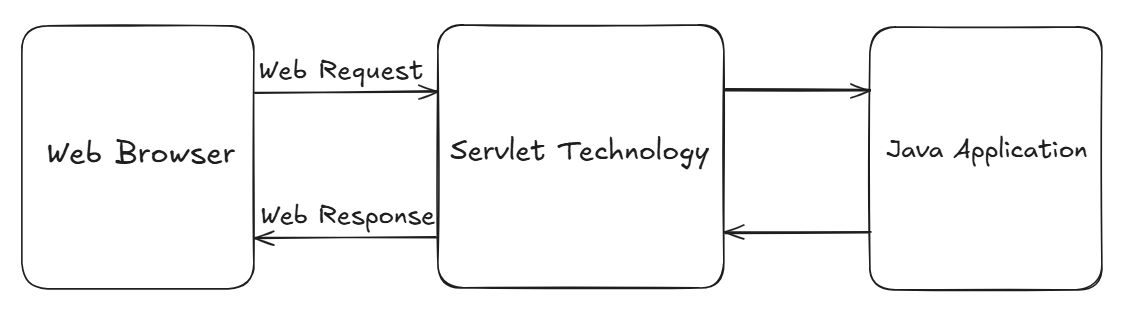
* Tells the client the format of the response content.
* **Common Values:**
  + **text/html** → HTML page
  + **application/json** → JSON data
  + **text/plain** → Plain text
  + **application/xml** → XML data

**3. Actual Content**

* The real body/content of the response (data or HTML page).
* **Examples:**
  + HTML webpage
  + JSON data (API response)
  + Image, PDF, etc.

**Servlet:**

* It is an API generally used for developing **web applications**
* All the classes and interfaces related to Servlet API are present in **javax.servlet** package
* **Servlet**  is an interface present in **javax.servlet** package
* It is the technology which is used to accept web request in the Java application and to generate web response from the java application.



**Servlet container/Web container**

* It is the component of web server which is responsible to manage servlet life cycle.

When the request is sent to any specific servlet then the servlet container will create request and response objects internally.

**Servlet life cycle**

* **Done by Servlet Container/Web Container**

1. **Loading Servlet Class**
2. **Instantiation of Servlet Class**
3. **Initialization of parameter of Servlet Class by invoking init()**

***(All the above steps will be executed only once!!!)***

1. **invoking service()**

**(service() method will get invoked N number of times for N number of requests!!!)**

1. **Destroying the Servlet by invoking destroy() method**

**web.xml (Deployment Descriptor):**

* Since any web request which has been generated from the web browser will be always accepted by the web server.
* But ***the web Server can serve only static web resources*** **so that if any request comes for the dynamic web resource, then web server is unable to serve that request .**
* **So, all the dynamic web resources will be served by** **servlet container or web container/Application Server.**
* But the web request will not go directly to the application server in that case we have to provide a **deployment descriptor (web.xml)** to the web server so that **the web server will refer deployment descriptor to forward the request to the application server to serve the request for dynamic web resource.**

**Servlet hierarchy:**

A diagram of a computer component

AI-generated content may be incorrect.

* **Servlet** is an interface present in **javax.servlet** package
* It has some methods used to implement the **Servlet API.** and perform some tasks when implemented further.

public void **init**(ServletConfig config) throws ServletException;

public ServletConfig **getServletConfig**();

public void **service**(ServletRequest req, ServletResponse res) throws ServletException, IOException;

public String **getServletInfo**();

public void **destroy**();

* **GenericServlet** is called as protocol independent servlet, since it allows any web request associated with any protocol.
* It is an **abstract class** which implements the **Servlet interface.**
* It will provide implementation to all the methods of servlet interface except **service method(remains abstract).**
* To create our **servlet class,** we must **extend/inherit GenericServlet** class.

**Steps to Create Servlet Maven Project**

**Step 1:** Create a new maven project( **don’t click create simple maven project!)**

**Step 2:** Search **maven-archetype-web,** select **1.5** and **next**

A screenshot of a computer

AI-generated content may be incorrect.

**Step 3:** Provide Group Id and Artifact Id and **finish**

**Step 4:** Press **Y** when asked on output console, wait for **BUILD SUCCESS** message.

**Step 5:** Visit [mvnrepository.com](https://mvnrepository.com/) and search for **Java Servlet API,** and select latest one.

**Step 6:** Add copied dependency into **pom.xml** dependencies tag.

**Step 7:** Right click on project🡪Properties🡪Java build path🡪order and export

🡪check all the boxes🡪apply and close

**Step 8:** Right click on project🡪maven🡪update project🡪select force update checkbox🡪ok

**Step 9:** Right click on project🡪Refresh

* **Note: Follow Step no, 7,8 and 9 if errors are comming or some problem is occurred while running the project**

**Apache Server Configuration**

**Step 1:** Go to Servers🡪Click to create a new server

**Step 2:** Select Apache🡪Select **Tomcat v9.0 server**

**Step 3:** Select **Tomcat installation directory**

* Create an new folder named **Server** in the system and select it.

**Step 4:** Download and install in same folder(**See footer for downloading details) 🡪**Finish after installation is completed.

**Creating a Servlet Class**

* Create a normal class in java resource🡪src/main/java
* Extend **GenericServlet** and implement the methods
* **To register the Servlet**
* Go to **src🡪main🡪webapp🡪WEB-INF🡪web.xml**
* Add this:
* In **servelet-name** tag: write the **servlet class name**
* In **servlet-class** tag: paste the **qualified name of servlet class**

**(right click on ClassName and copy the qualified name)**

**e.g.**

<!-- Registering the Servlet -->

<**servlet**>

<**servlet-name**>MyServletClass</**servlet-name**>

<**servlet-class**>edu.UDServlets.MyServletClass

</**servlet-class**>

</**servlet**>

* **Ctrl+ hover** on qualified name will display the link, which can redirect us to the **Servlet Class**

**e.g**

<!-- URL mapping -->

<**servlet-mapping**>

<**servlet-name**>MyServletClass</**servlet-name**> <**url-pattern**>/message</**url-pattern**>

</**servlet-mapping**>

* Whenever the **URL containd /message** , MyServletClass **service method** will be invoked
* [**http://localhost:8080/servlets/message**](http://localhost:8080/servlets/message) **will execute the service method(output can be visible in console)!!!**
* **To Display the content on web page:**

***@Override***

**public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException** {

System.*out*.println("service method is called!");

PrintWriter printWriter=**res**.getWriter();

***printWriter.print(***

***"<body>"***

***+"<h1>"***

***+ "Hello I am service method of MyServlet Class"***

***+"</h1>"***

***+ "</body>"***

***);***

}

**Running the Servlet Project on Server**

**Step 1:**Right click on project🡪Run as🡪Run on server🡪Select server and finish

**Step 2:** Select **window🡪web browser🡪Chrome**

Access the web page using:

**http://localhost:8080/your\_project \_name/(url given in url-pattern tag)**

* **To Link Servlet Classes with URLs**

**e.g.**

//Annotation used to declare a servlet.

//This annotation is processed by the container at deployment time,

//and the corresponding servlet made available at

//the specified URL patterns.

*@WebServlet*(value="/page-b")

public class PageB extends HttpServlet {

***Using this annotation, we can register and do url mapping without even touching web.xml!!!***

**Reading Form data using generic servlet**

**Step 1:** src🡪webapp🡪new🡪**HTML File(name it as userinfo.html)**

**Step 2:** write a html code to create one form

**e.g.**

* <**form** action=*"info"* method=*"post"*>
* <**label** for=*"name"*>Name:</**label**><**br**>
* <**input** type=*"text"* id=*"name"* name=*"name"* required><**br**><**br**>
* <**label** for=*"email"*>Email:</**label**><**br**>
* <**input** type=*"email"* id=*"email"* name=*"email"* required><**br**><**br**>
* <**label** for=*"age"*>Age:</**label**><**br**>
* <**input** type=*"number"* id=*"age"* name=*"age"* min=*"1"* required><**br**><**br**>
* <**input** type=*"submit"* value=*"Register"*>
* </**form**>

**Step 3:** Create one Servlet class (**UserInfo**) **extending GenericServlet and overriding service method**

* **Note:** Values passed **in getParameter()** and **name attribute of input tag must be same!!!**

**e.g.**

***@Override***

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {

String name = req.getParameter("name");

String email=req.getParameter("email");

String age=req.getParameter("age");

System.*out*.println("Name: "+name);

System.*out*.println("Email: "+email);

System.*out*.println("Age: "+age);

}

**Step 4:** Perform **Servlet Registering and URL mapping**  for this Servlet class in **web.xml** file  **or use @WebServlet annotation**

* **Note:** KEEP<**url-pattern**>/info</**url-pattern**> in web.xml and **<form action=**”info”**>** same!

**Step 5:** Run the tomcat server and access the following URL:

[**http://localhost:8080/servlets/info.html**](http://localhost:8080/servlets/info.html)

**Step 6:** Enter the details In the form and click **Register Button,**

* **Url will be looking like:**

**http://localhost:8080/servlets/info?name=Rakesh&email=Rakesh123%40gmail.com&age=45**

* And the Entered values will be displayed on **Output console**

**Name: Rakesh**

**Email: Rakesh123@gmail.com**

**Age: 45**

**HttpServlet**

* It is **an abstract class** present in **javax.servlet** package without any **abstract methods**
* Its is the Child class of **GenericServlet Class**, which provides some implementation for **service method**
* **HttpServlet** is called as protocol dependent servlet, since it allows any web request associated with only http protocol.
* It is always recommended to create A Servlet Class by **extending HttpServlet as it provides higher security to the web application**

**Reading Form data using HttpServlet**

* **Using doGet() method:**
* The **doGet()** method is used in Java Servlets **to handle HTTP GET requests** from a client (usually a web browser).
* When a client sends a request to a servlet using the HTTP GET method **(e.g., by entering a URL in the browser, clicking a link, or submitting a form without method="post")**, the servlet container (like Tomcat) automatically calls the servlet’s doGet() method.
* **Purpose:**
* **Receive data from the client** (via query parameters in the URL).
* **Process the request** (e.g., fetch from a database, generate dynamic HTML, return a file).
* **Send a response back** to the client.
* **Follow same steps as *Reading form data using GenericServlet***
* **Different Steps:**

**Step 1:** Create a Servlet class(**Register.java**) by **extending HttpServlet class**

**Step 2:** Create a form to take user data,

<!-- If we don’t mention the method type, by default it will be

**method="get"** inside the **<form> tag**-->

<**form** action=*"register"* id=*"form"*>

**Step 3:** Override **doGet()** method of **HttpServlet class**

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

String name = req.getParameter("name");

String email=req.getParameter("email");

String age=req.getParameter("age");

PrintWriter printWriter=resp.getWriter();

printWriter.write(

"<body>"

+"<h1>User is Registered Successfully!!!</h1><br>"

+"<h2>Name: "+name+"</h2>"

+"<h2>Email: "+email+"</h2>"

+"<h2>Age: "+age+"</h2>"

+"<body>"

);

}

* **Using doPost() method:**
* The **doPost()** method is used in Java Servlets **to handle HTTP POST requests** from a client (usually a web browser).
* It is automatically called by the servlet container (like Tomcat) when it receives a POST request for any servlet.
* **Purpose**

1. **Process POST Requests**
   * Handles requests where data is sent in the **HTTP request body** **(not in the URL).**
   * Commonly used for:
     + Form submissions with **sensitive data (passwords, personal info)**
     + File uploads
     + Sending large amounts of data
2. **Secure & Large Data Handling**
   * POST requests are **not cached** and **not stored in browser history**.
   * Suitable for large data transfer (unlike GET, which has a URL length limit).

**Step 1:** Make form tag as:  
<**form** action=*"register"* **method=*"post"*** id=*"form"*>

**(We have to specify method type as post explicitly!!!**)

**Step 2:** Override **doPost() method** and provide implementation in **Register.java** or create another Servlet Class.

* **Difference**

| **Aspect** | **doGet()** | **doPost()** |
| --- | --- | --- |
| **HTTP Method** | Handles **GET** requests | Handles **POST** requests |
| **Where Data is Sent** | Appended to the **URL** as query string (e.g., ?name=John&age=25) | Sent in the **request body**, not visible in the URL |
| **Data Size Limit** | Limited (around **2 KB** depending on browser/server) | Can handle **much larger** amounts of data (depends on server configuration) |
| **Security** | Less secure — data is visible in URL and stored in browser history/server logs | More secure — data is hidden from URL and not stored in history/logs |
| **Use Cases** | Fetching/displaying data, navigation, search queries | Submitting sensitive or large data, uploading files, API requests with JSON/XML |
| **Performance** | Faster for small requests because of URL-based data | Slightly slower due to sending data in the body |
| **Default Method in HTML Forms** | Yes — if no method attribute is given, default is GET | No — must explicitly set method="POST" |

**Query String**

* It represents the data to be sent from the **client side to the server** in order to perform specific task at the server side.
* We can pass the data through **url** using **Query String.**
* Data can be sent in **key-value pair**, separated by **&**
* **Passing the value as url ( URL Re-Writing):**

**<a href=*"page-b?id=10&name=priya&age=22"*>**

**Click to visit Page B**

**</a>**

* Here, ***?id=10&name=priya&age=22*** will act as a Quey String.
* **Servlet Class Implementation**

***@Override***

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

**String** id=**req**.getParameter("id"),

name=**req**.getParameter("name"),

age=**req**.getParameter("age");

**PrintWriter** printWriter=resp.**getWriter**();

printWriter.**write**(

"<h1>This is Page B</h1>"

+"<h3>Id: "+id+"</h3>"

+"<h3>Name: "+name+"</h3>"

+"<h3>Age: "+age+"</h3>"

);

* After Running the**queryString.html,** we will get
* **Url in search bar:**

**http://localhost:8080/servlets/pageb?id=91&name=priyanka&age=31**

**This is Page B**

**Id: 91**

**Name: priyanka**

**Age: 31**

* *If we change the query string in url, changes will also get affected in the Page Content!*

**<welcome-file-list>**

* The <welcome-file-list> is an element in the web.xml deployment descriptor in a Java web application.
* It defines ***the default resource(s) (usually HTML or JSP files) that the web container should serve when a user accesses a root URL of the application without specifying a file.***
* **Root url : http://localhost:8080/your-project-name/**

**e.g.**

<**welcome-file-list**>

<**welcome-file**>PageA.html</**welcome-file**>

<**welcome-file**>PageB.html</**welcome-file**>

<**welcome-file**>PageC.html</**welcome-file**>

</**welcome-file-list**>

* **Working:**
* It works as **else-if** ladder
* If PageA is available, display it. If A is not available display **PageB** and so on.
* If none of the page is available, it will Give **404 Error(page not found).**
* If we don’t provide <**welcome-file-list**>, Web container will search for **index.html** and opens it as **welcome file.**
* If index.html is not present, it will look for **index.htm** and opens it as **welcome file.**
* If index.htm is not present, it will look for **index.jsp** and opens it as **welcome file.**
* **Note:** Priority is given to **<welcome-file-list>** over other files!!!

**✅ Purpose of welcome-file-list**

* It is used when we want to set a **custom html/jsp file as our welcome page**, rather than the built in file like index.html.

**<load-on-startup>**

* It tells the container whether a servlet should be **loaded immediately when the application starts** or **only when it is first requested by the Web Container**.
* It is decided on the basis of **priority in number(lower the number, higher the priority while loading)**

**e.g.**

<servlet>

<servlet-name>**ServletClass1** </servlet-name>

<servlet-class>edu.UDServlets.ServletClass1 </servlet-class>

**<!-- load on startup -->**

**<load-on-startup>**

**2**

**</load-on-startup>**

</servlet>

<servlet>

<servlet-name>**UserInfo**</servlet-name>

<servlet-class>edu.UDServlets.UserInfo</servlet-class>

**<load-on-startup>**

**9**

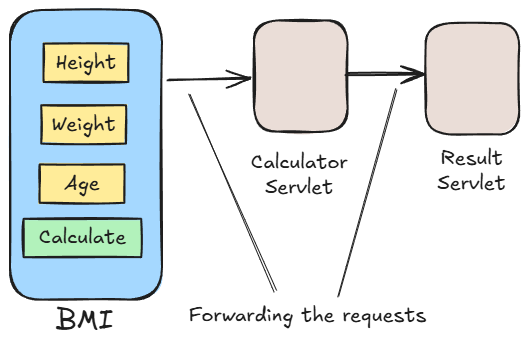
**</load-on-startup>**

</servlet>

* Here, **ServletClass1** is loaded first, then **UserInfo** Servlet will be loaded after the Application starts
* **Servlets Classes** which are not provided with any **<load-on-startup>** will not be loaded ***immediately after the application starts, they will only get loaded when request is sent for them!!!***

**RequestDispatcher<<I>>**

* It is an interface present in **javax.servlet** package.
* It is used
* to forward a request to another resource (like another servlet, JSP, or HTML file) on the server side, or
* to **include the content** of another resource in the response.
* It is used to dispatch the request from
* **Servlet to Servlet**
* **Servlet to jsp/html**
* **Jsp to jsp**
* **Jsp to Servlet**
* **getRequestDispatcher()** method is used to get the **RequestDispatcher object** with the help of **Servlet request object**
* Using **RequestDispatcher object,** we can use some methods like:
* **forward(req, resp):** forwards the request to another resource (no response sent yet)

****

* **forward() implementation**
* Create **CalculateBMI and BMIResult** servlet classes. Register and map url for both in **web.xml or use @WebServlet**
* Create **bmi.html** to create a form.
* Override doPost() or doGet() in **CalculateBMI** and provide implementation

To get **height and weight** from **form,** and calculate **BMI**

* Put all attributes(**height, weight and bmi)** in url :

**//Setting the attributes in this request object**

**req.setAttribute("bmi", bmi);**

**req.setAttribute("height", h);**

**req.setAttribute("weight", w);**

* Create a RequestDispatcher object and pass URL of the Resource to which we have to forward the request.( **BMIResult** )

**RequestDispatcher dispatcher=req.getRequestDispatcher("result**");

//url pattern of the Servlet to which we want to dispatch this request

**//i.e BMIResult**

* Forward the request to **BMIResult**

**dispatcher.forward(req, resp);**

* Provide implementation to doPost() or doGet() method in **BMIResult** to display the content
* **include(req,resp):** Includes the content of another resource in the current response.

**A diagram of a email password

AI-generated content may be incorrect.**

* **include() implementation**
* Create a **login.html** file in **src/main/webapp**
* Create a Servlet class (**UserLogin) ,** using **@WebServlet**
* Override **doPost()** and provide implementation
* TO include **login.htm**l if validation fails**(to show login page along with message)**

**RequestDispatcher dispatcher=req.getRequestDispatcher("login.html");**

**dispatcher.include(req, resp);**

**}**

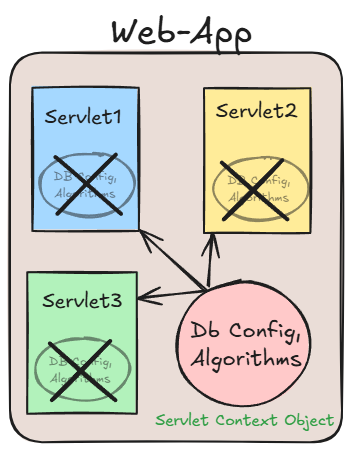
* **sendRedirect()**
* It is used to redirect the client to a different resource (URL).
* This method is part of the **HttpServletResponse** interface.

**e.g.: resp.sendRedirect("index.jsp");**

**//it will redirect user to index.jsp page!**

**// we can provide any url!!!**

**Servlet Context**



* **ServletContext is an interface** provided by the **Servlet API** that allows **servlets to interact with their servlet container** and **share data across the entire web application**.
* **One ServletContext object** per web application, created by **ServletContainer/Web Container**.
* **ServletContext object** is shared among **all servlets** in the application.
* Used for **application-wide parameters** and **resources**.
* Accessed via **getServletContext()** method from Servlet
* **Storing and Getting Values in Servlet Context**
* **In web.xml,** declare:

<**context-param**>

<!-- Data in key value pair -->

<**param-name**>user</**param-name**>

<**param-value**>postgres</**param-value**>

</**context-param**>

<**context-param**>

<**param-name**>password</**param-name**>

<**param-value**>root1234</**param-value**>

</**context-param**>

<**context-param**>

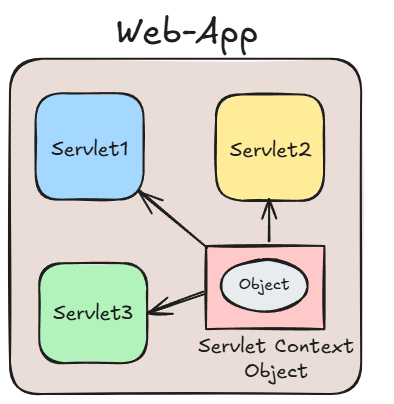
<**param-name**>url</**param-name**>

<**param- value**> jdbc:postgresql://localhost:5432/MyDB</**param-value**>

</**context-param**>

* *These are shared/common for all the* ***Servlet Class in the project.***

* **Using these parameters in Servlet Class**
* Create a java class**(ServletContextExample)** and make it as a Servlet Class **by extending HttpServlet and using @WebServlet**
* Override doGet() method and provide implementation
* **Storing and Getting Objects in Servlet Context**

****

* Refer **ServletContextExample2** and **DbInfo** classes

**ServletConfig**

* **ServletConfig** is an **interface** in the **javax.servlet** package that provides configuration information to a single servlet.
* It is used to pass initialization parameters from the web.xml (deployment descriptor) file to a particular servlet instance.
* **ServletConfig Objects** are **one** for the Servlet, independent of other Servlets.!! and are created by **Web Container** when Servlet Class get instantiated.
* We can get **ServletConfig Object** by calling **getServletConfig()** method.
* **🔧 When is ServletConfig Used?**

When you want to provide **custom configuration data** to a specific servlet during its initialization, you use **ServletConfig**.

**Web-App**

**A diagram of a diagram

AI-generated content may be incorrect.**

**Storing and Retrieving values from config object**

* **In web.xml:**
* <**servlet**>
* <**servlet-name**>CircleServlet</**servlet-name**>
* <**servlet-class**>edu.UDServlets2.CircleServlet</**servlet-class**>
* <!-- Init Parameters -->
* <**init-param**>
* <**param-name**>PI</**param-name**>
* <**param-value**>3.142</**param-value**>
* </**init-param**>
* </**servlet**>
* **Initialization Parameters** are given while registering the particular servlet because they belong to that servlet only!!
* **//OR**
* **Without Writing anything in web.xml:**

*@WebServlet*(

* value="/circleservlet",
* initParams= {
* *@WebInitParam*(name="PI",value="3.142")
* }
* )
* **Refer Circle class for implementation**

**Session Management**

* http protocol is a stateless protocol **i.e.** Every request to the server is treated as new request bcz Server will not remember the client! In every next/new request.
* So Client need to introduce self for each new request

e.g. **Providing username and password for every request in insta: login, then for accessing photos again and so on…..**

* So, to avoid this, we will make use of **Session.**
* **Session** is a **state/object** which will include the data of the client and store it for a particular time.

**Why is Session Management Needed?**

* **HTTP is stateless**: Each request is treated independently, and the server does not remember who made previous requests.
* For personalized experiences, such as:
  + Logging in users
  + Shopping carts
  + User preferences
  + Tracking user activity

We need a way to persist information across requests. That’s where **session management** comes in.

* So, **Session management** helps us **maintain user state/data** across multiple requests.
* **Techniques** for session management in Java Servlets:
* **Cookie**
* **Cookie** is an object which can store a **small piece of information.**
* It is stored on the **Client Side.(Client’s Browser)**
* In Cookie, **small piece of information** is stored in **Key-Value pair.(Only String data)**

**A diagram of a process

AI-generated content may be incorrect.**

* In Java, **Cookie** is a class present in **javax.servlet.http** package
* We can create a Cookie object using:
* **new Cookie()**
* **new Cookie(String key, String value)**

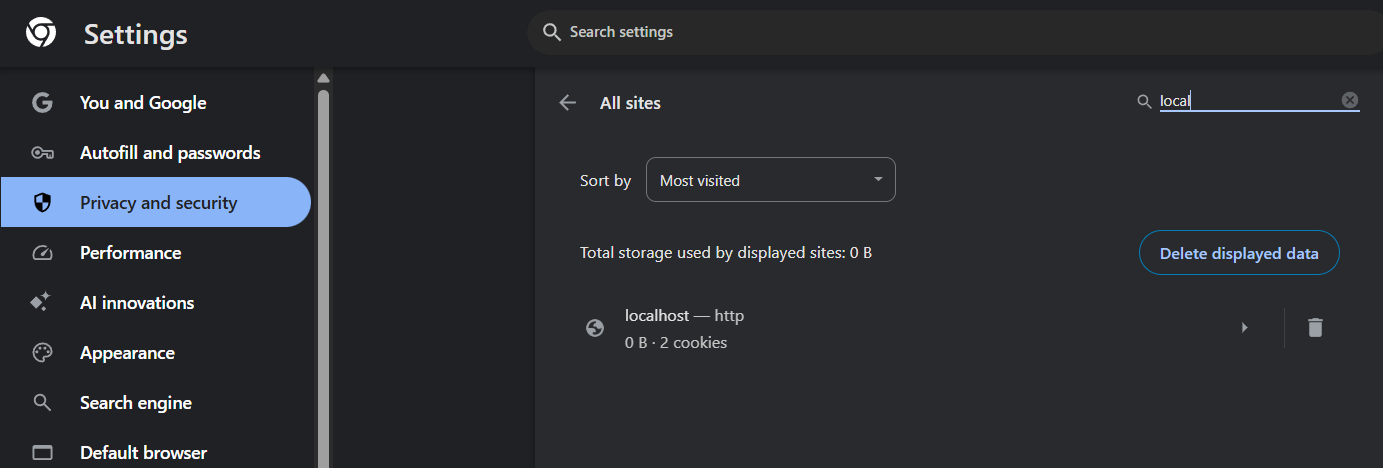
**Advantages:**

* Simple Design
* Client Side Management

**Dis-Advantages:**

* Only Stores data in form of **String**
* Can be disabled from **Client Side**
* **Refer Cookies class in sessionmanagement package**
* **After Running the Cookies class, Checking Cookie data on Browser:**

**Settings🡪Privacy and Security🡪Third Party Cookies🡪See all site data and permissions🡪**



**URL re-writing**

* **Sending the information in URL**
* **Passing the value as url:**

**<a href=*"page-b?id=10&name=priya&age=22"*>**

**Click to visit Page B**

**</a>**

* Here, ***?id=10&name=priya&age=22*** will act as a **Query String.**
* Passing the data in every link is **very hectic and not optimal approach for managing the session!!**

**Managing Session using HttpSession**

* It is an **interface** present in **javax.servlet.http** package.
* We can create **HttpSession** **object** by invoking:
* **getSession(),** which will return us the **existing HttpSession object** if present, else it will return new **HttpSession object.**
* **getSession(boolean)**
* if true is passed, it will return **new HttpSession object** **even If it exists**
* if false is passed, It will work like **getSession()**
* Create Servlet Classes **HttpSessionExample**and override **doGet():**
* Create one normal class **UserDetails**
* class UserDetails{
* String firstName,lastName,email,gender;
* long phone;
* public UserDetails(String firstName, String lastName, String email, String gender, long phone) {
* super();
* this.firstName = firstName;
* this.lastName = lastName;
* this.email = email;
* this.gender = gender;
* this.phone = phone;
* }
* }

**HttpSession in login and logout Operation**

* In **web.xml,** set Credentials:
* <!-- Session Management -->
* <**context-param**>
* <!-- Data in key value pair -->
* <**param-name**>Username</**param-name**>
* <**param-value**>admin</**param-value**>
* </**context-param**>
* <**context-param**>
* <**param-name**>Password</**param-name**>
* <**param-value**>admin@123</**param-value**>
* </**context-param**>
* Create one **SessionLogin.html** form

**<form action=*"session-a"* >**

* Create **3 Servlet classes,** override **doGet()** and provide implementation
* ***SessionA.java***
* **ContentPage.java**
* **Logout.java**

**JSP**

* **JSP stands for java server page.**
* It allows developers ***to embed Java code*** directly into ***HTML pages*** *using special tags.*
* **Servlets** can be used if the focus is more on java logic and **JSP's** can be used if the focus is more on HTML logic.
* ***All the JSP files will be translated into servlets by jsp translator.***

**Creating a jsp file**

* src/main/webapp🡪new jsp file
* **JSP Tags**

**1. Declaration tag <%! %>**

* This tag is used to declare the variables or methods of a java.
* Anything inside this tag will be present outside the **service() method!.**
* Variables declared here are **global variables.**

**2. Scriptlet tag <% %>**

* This tag is used to write java logic for dynamic response.
* Anything inside this tag will be returned within the **service() method!.**

**3. Expression tag <%= %>**

* This tag is used to **print/display** results on web browser.
* It is used to **evaluate and output the result of an expression**
* We cannot use **void methods** , as it don’t return anything!!

**4. Directive tag <%@ %>**

* This tag is used to import external resources in a JSP file.
* All the JSP files are capable of serving only HTTP requests because all the JSP files will be **translated/converted** into HttpServlet.

**JSP life cycle:**

JSP life cycle is very much similar to servlet life cycle.

**Different phases of JSP life cycle**

*1. JSP file will be translated to the servlet by the component called as* ***JSP translator, a part of Web Container*** *and then compiled.*

*2. The translated servlet class will be loaded.*

*3. An* ***instance*** *will be created for the servlet class.*

*4. The servlet instance will be initialized by invoking* ***jspInit()*** *method.*

*5. The service will be provided for each request by invoking* ***jspService()*** *method, for* ***n number of times,*** *whenever the request is made!!!*

*6. Once the server stops, the jsp will get destroyed by invoking* ***jspDestroy().***

* **Servlet container** itself will manage the JSP life cycle.
* Servlets are faster than JSP's in giving responses.

**JSP implicit Object(request)**

* **request object** is implicitly present in jsp
* Refer **Register.jsp** and **Display.jsp**
* **Run Register.jsp**

**JSP implicit Object(session)**

* **session object** is implicitly present in jsp.
* **Reading the content set in Servlet Through jsp**
* Create a servlet class **StudentInformation** and one normal **Student** class with all public fields and **all args constructor**
* Refer **StudentInformation** and **Student** present in **UnderstandingJsp package**
* Create one **studentInfo.jsp**  in **webapp**