|  |
| --- |
| **Servlet Basics** |

**Servlet Life Cycle Methods and Phases**

For a java program, there is one method which serves as the entry point to the application which is the main() method. For a servlet, we have three methods listed below.

**Life Cycle methods**

init()

service()

destroy()

These methods are called by the Web Container such as Apache Tomcat, WebLogic.

Init() method is called once. So, this is a good place to write code to establish connections to the database server.

The Service method is where all our business logic is kept. The service method is called n-times.

The destroy method should have code which does the opposite of what the init() method does. The destroy method is also called once.

Putting code in these method is our responsibility but calling the methods is the container’s responsibility.

There are 4 life cycle phases associated with the above three methods, which are listed below.

**Life Cycle Phases**

Instantiation – This is when the classes are loaded by the container.

Initialization – This is when the init method is used to open resources.

Servicing – this is where the service method is executed.

Destruction – this is when the destroy method is called.

**Web Application Folder Structure**

com/samsonmarikwa/OrderServlet.class

login.jsp

Mysql.jar spring.jar hibernate.jar

lib

classes

web.xml

home.html

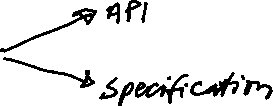
WEB-INF

WEBAPP

**Servlets**

A technology in the JEE that allows us to build dynamic web applications using java. The Servlet standard has an API and a Specification just like other standards from Oracle.

The API is for developers to develop dynamic web applications. There is a set of interfaces and classes. The Specification itself is a set of rules written in plain English which is for Application Server or Web Container developers like Apache Tomcat, WebSphere, WebLogic. The developers follow the rules described in the spec.



**Servlet**

is a program which runs on a web container. It receives input from a browser, processes it and depending on the request, can make calls to a database server. It can do a lot of things such as writing to files and reading from files. It receives responses from the database server and processes it before sending the response back to the client (browser).

OrderServlet.java

Servlet

Web Browser

DB

Web Container

**Servlet Annotations 3.0**

Servlet 3.0 version introduces annotations that can be used to configure various servlet components instead of using the web.xml file.

Examples include

@WebServlet

@WebInitParam

@WebFilter

@WebListener

*web.xml configurations override the annotations defined for the same component.*

**Registering a servlet**

Within the web.xml, create the following entries

|  |
| --- |
| <servlet>  <servlet-name>HelloServlet</servlet-name>  <servlet-class>com.samsonmarikwa.training.servlets.HelloWorldServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>HelloServlet</servlet-name>  <url-pattern>/hello</url-pattern>  </servlet-mapping> |

**Application Flow**

When a request comes from the browser to <http://localhost:8080/ServletBasics/hello> Apache Tomcat will handle the request as it is running on localhost port 8080. Tomcat determines the project name from the url that should handle the request. It then looks at the web.xml file for the /hello path configured in the servlet-mapping url-pattern to see which servlet should handle the request. It then instantiates the servlet HelloWorldServlet to handle the request. It will then execute the service method that is in the logic in our servlet.

**When using html**

To send input to the servlet via html form

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition</title>  </head>  <body>  <form action=*"additionServlet"*>  <h2>Enter the Numbers</h2><br />  Number1: <input type=*"text"* name=*"number1"* /><br/>  Number2: <input type=*"text"* name=*"number2"* /><br/>  <input type=*"submit"* name=*"submit"* />  </form>  </body>  </html> |

Add a servlet to retrieve the input, add and send the response

|  |
| --- |
| **public** **class** AdditionServlet **extends** GenericServlet {  @Override  **public** **void** service(ServletRequest request, ServletResponse response)  **throws** ServletException, IOException {  **if** (request.getParameter("number1") != **null** && request.getParameter("number2") != **null**) {  **int** num1 = Integer.*parseInt*(request.getParameter("number1"));  **int** num2 = Integer.*parseInt*(request.getParameter("number2"));  PrintWriter out = response.getWriter();  out.println("The result is " + (num1 + num2));  }  }  } |

Add a servlet to retrieve the input, add and send the response

|  |
| --- |
| <servlet>  <servlet-name>AdditionServlet</servlet-name>  <servlet-class>com.samsonmarikwa.training.servlets.AdditionServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>AdditionServlet</servlet-name>  <url-pattern>/additionServlet</url-pattern>  </servlet-mapping> |
| **JDBC** |

**JDBC Architecture**

|  |  |
| --- | --- |
| JDBC Client | The client is the code that we write. Connects to the DB. Perform CRUD operations. Processes the response. Close the connection. |
| JDBC API | Part of java.sql.\* package  We as developers, we learn and use the JDBC API  Comprises Connection, Statement, ResultSet and many more |
| JDBC Driver | Exists between the client software and the DB. Implements the JDBC API under the JDBC Specification. |
| JDBC Manager | Driver Manager establishes connection between the Client and Driver. After that the communication takes place between the client and the driver. |

Only Once to get a connection

Communicates ‘n’ times after a connection is acquired via the Driver Manager

DatabaseServer

JDBC API

Driver Manager

JDBC Driver

JDBC Client

**Steps to perform CRUD**

1. Establish the connection
2. Create the statement object
3. Submit the SQL query to DBMS
4. Close the statement
5. Close the connection

**Download MySQL Connector/J jar file.**

Copy the file to lib folder created in the project. Add the lib folder to the classpath under the Project Properties

Graphical user interface, text, application

Description automatically generated

**Steps to Use JDBC**

|  |  |
| --- | --- |
| Establish Connection interface | Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd"); // default port is 3306 |
| Create the Statement interface used to execute different different types of SQL statements | Statement statement = connection.createStatement(); |
| JDBC Driver | Exists between the client software and the DB. Implements the JDBC API under the JDBC Specification. |
| JDBC Manager | Driver Manager establishes connection between the Client and Driver. After that the communication takes place between the client and the driver. |

**executeUpdate()**

**try** {

Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

Statement statement = connection.createStatement();

**int** result = statement.executeUpdate("insert into account values(1, 'Marikwa', 'Samson', 10000)");

System.***out***.println(result + " rows got inserted");

**int** result = statement.executeUpdate("update account set bal = 25000 where accno = 1");

System.***out***.println(result + " rows got updated");

**int** result = statement.executeUpdate("delete from account where accno = 1");

System.***out***.println(result + " rows got deleted");

} **catch** (SQLException e) {

e.printStackTrace();

}

**ResultSet**

ResultSet interface is used to handle the data that comes back when we execute a select query. It is an OO representation of the table records. It consists of three areas as shown below.

No Record Area

Cursor

Record Area

Zero Record Area

Moves the cursor

next() getXXX();

used to read individual fields in the record. XXX represents Int, Boolean, String etc. for each Primitive Type. The parameters for the method can be the column name or number.

Returns Boolean

**executeQuery()**

**try** (Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery("select \* from account");

){



**int** result = statement.executeUpdate("insert into account values(1, 'Marikwa', 'Samson', 10000)");

System.***out***.println(result + " rows got inserted");

**int** result = statement.executeUpdate("update account set bal = 25000 where accno = 1");

System.***out***.println(result + " rows got updated");

**int** result = statement.executeUpdate("delete from account where accno = 1");

System.***out***.println(result + " rows got deleted");

**while** (resultSet.next()) {

System.***out***.println(resultSet.getString(2));

System.***out***.println(resultSet.getString(3));

System.***out***.println(resultSet.getInt(4));

}

} **catch** (SQLException e) {

e.printStackTrace();

}

Closing of resources like connection, statement, resultSet was done in the finally block before it was introduced in Java 7. Now, by using try-with-resources, the resources are auto-closed. The resources that can be closed are those that implement or extend the AutoCloseable interface.

**The Service Provider Mechanism**

A Service Provider is configured and identified through a provider configuration file which we put in the resource directory *META-INF/services*. The file name is the fully-qualified name of the SPI (SP Interface) and its content is the fully-qualified name of the SPI implementation.

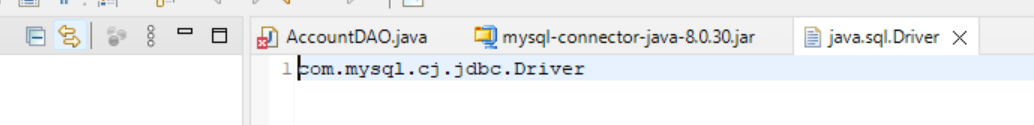
The Service Provider is installed in the form of extensions, a jar file which we place in the application classpath, the Java extension classpath or the user-defined classpath.

This is what is used by the DriverManager to find the appropriate driver to connect to the database.

The SPI feature was introduced in Java 6.

Graphical user interface, application, Word

Description automatically generated



|  |
| --- |
| **BUILDING DYNAMIC WEB APPLICATIONS** |

**HTTP Methods**

GET vs POST

|  |  |
| --- | --- |
| GET | POST |
| Default (if no HTTP method is mentioned, the web browser uses a GET by default) | The POST method has to be mentioned explicitly. <form method=”post” … |
| 3 cases |  |
| * User Specified URL – when a user types in a URL in the browser * User click on a hyperlink * User submits a form |  |
| Meant for getting data | Meant for posting data, creating, updating or deleting |
| No body | Body is present |
| Query String | Payload / Body |
| No sensitive data | Can be used for sensitive data |
| Restriction on amount of data | No restriction |
| Idempotent – we can execute GET any number of times and it will not affect the state of the application. It is considered a safe operation because no changes takes place to the data that is retrieved. | Not idempotent – if you submit data more than once, there is a chance that duplicate or multiple records will be created. Multiple updates can also take place. This can also occur if you refresh your browser. |

Creating a dynamic web project to use MySQL

1. Copy the MySQL Connector jar file to the lib folder shown below

Graphical user interface, text, application, email

Description automatically generated

The file will be added to the build path.

1. Create the html files and the CreateUserServlet.

Add the annotation for @WebServlet(“/addServlet”)

1. Implement the init() method in the CreateUserServlet. The init method is called once in the lifecycle of the servlet.

**public** **void** init() {

**try** {

// Before Java6 Service Provider Mechanism we had to load the driver class. This is required for Tomcat as

// this the auto-loading of the class is disabled due to memory leaks and we are required to load the class manually.

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

connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

} **catch** (SQLException | ClassNotFoundException e) {

e.printStackTrace();

}

}

1. Implement the doPost() method which gets called from every request from the browser.

|  |
| --- |
| **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String firstName = request.getParameter("firstName");  String lastName = request.getParameter("lastName");  String email = request.getParameter("email");  String password = request.getParameter("password");  **try** {  Statement statement = connection.createStatement();  **int** result = statement.executeUpdate(  "insert into user values('"  + firstName + "','" + lastName + "','"  + email + "','" + password + "')");  PrintWriter out = response.getWriter();  **if** (result > 0) {  out.print("<H1>USER CREATED</H1>");  } **else** {  out.print("<H1>Error Creating the User</H1>");  }  } **catch** (SQLException e) {  e.printStackTrace();  }  } |

1. Implement the destroy() method.

|  |
| --- |
| **public** **void** destroy() {  logger.info("destroy() started");  **try** {  connection.close();  } **catch** (SQLException e) {  logger.error(e.getMessage());  }    } |

|  |
| --- |
| **Init Parameters** |

**Servlet Init Parameters**

Init parameters are name value pairs of textual information that are supplied to a servlet declaratively, that is, through a web.xml file during its initialization phase. During the initialization phase, the servlet container creates the ServletConfig object from the web.xml through which parameters are passes to the servlet through the init method. A ServletConfig is applicable to a single Servlet.

Init(ServletConfig config)

Servlet

Container

String dbUser = config.getInitParameter(“dbuser”);

**Configure using Annotations**

|  |
| --- |
| @WebServlet(urlPatterns = "/addServlet", initParams = {  @WebInitParam(name = "dbUrl", value = "jdbc:mysql://localhost/mydb"),  @WebInitParam(name = "dbUser", value = "root"),  @WebInitParam(name = "dbPassword", value = "P@ssW0rd")  })  **public** **class** CreateUserServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Connection conn;  **public** **void** init(ServletConfig config) {  **try** {  Class.*forName*("com.mysql.cj.jdbc.Driver");  conn = DriverManager.*getConnection*(config.getInitParameter("dbUrl"), config.getInitParameter("dbUser"), config.getInitParameter("dbPassword")); |

**Configure using web.xml**

No hardcoded values as values come from the web.xml file

|  |
| --- |
| <servlet>  <display-name>ReadUserServlet</display-name>  <servlet-name>ReadUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.ReadUserServlet</servlet-class>  <init-param>  <param-name>dbUrl</param-name>  <param-value>jdbc:mysql://localhost/mydb</param-value>  </init-param>  <init-param>  <param-name>dbUser</param-name>  <param-value>root</param-value>  </init-param>  <init-param>  <param-name>dbPassword</param-name>  <param-value>P@ssW0rd</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>ReadUserServlet</servlet-name>  <url-pattern>/readServlet</url-pattern>  </servlet-mapping> |

The init() method remains the same as the one when using annotations.

**Summary**

* Interface used to access init parameters which is available to a particular servlet

ServletConfig

* In the web.xml, what is the parent element of the init-param element

servlet

* Which annotation can be used inside the WebServlet annotation to configure init parameters?

@WebInitParam

|  |
| --- |
| **Servlet Context** |

**ServletConfig** and **ServletContext**, both are objects created at the time of [servlet](https://www.geeksforgeeks.org/introduction-java-servlets/) initialization and used to provide some initial parameters or configuration information to the servlet. But, the difference lies in the fact that information shared by ServletConfig is for a specific servlet, while information shared by ServletContext is available for all servlets in the web application.

**ServletContext:**

* ServletContext is the object created by Servlet Container to share initial parameters or configuration information to the whole application. There is one ServletContext for the entire application. Any servlet or JSP has access to the ServletContext which has data which has been put by another servlet or JSP, so the scope of the ServletContext is Application.
* Example:  
  Suppose, the name of one’s job portal is “NewWebsite.tg”. Showing the website name at the top of webpages delivered by different servlets, one needs to store the website name in every servlet inviting redundancy. Since the information shared by ServletContext can be accessed by every Servlet, it is better to go with ServletContext and retrieve the website name using getServletContext.getInitParameter(“Name”) whenever required.

<**context-param**>

<**param-name**>Website-name</**param-name**>

<**param-value**>NewWebsite.tg</**param-value**>

</**context-param**>

web.xml

Container

Servlet

injects

Initialization

**Retrieve ServletContext**

Once injected, the ServletContext can be retrieved in several ways.

1. **Init method with Config.**

The container sets the ServletContext into the ServletConfig when the servlet initializes.

public void init(ServletConfig config) {

ServletContext context = config.getServletContext();

1. **Init method without config**

The servlet inherits the getServletContext method from the Servlet class.

public void init() {

ServletContext context = getServletContext();

1. **In the service(…) methods that takes HttpServletRequest and HttpServletResponse**

ServletContext context = getServletContext();

**Using The ServletContext**

1. **Share and Manipulate Data**

You can set data in one servlet using **setAttribute()**

You can retrieve the data in another servlet using **getAttribute()**

You can remove data from another servlet or JSP using **removeAttribute()**

You can get all the attribute names into an array using **getAttributeNames()**

Enumeration<String> parameterNames = context.getInitParameterNames();

1. **To deal with Context params**
2. **To create the RequestDispatcher object for inter-servlet communication**
3. **To store information in to the server log files using the log() method**

This is rarely used since we used third-party libraries like log4j.

**Context Parameters**

Are name value pairs defined in the root element of the web.xml

<**context-param**>

<**param-name**>Website-name</**param-name**>

<**param-value**>NewWebsite.tg</**param-value**>

</**context-param**>

Summary

* Context-param are a child element of webapp
* Using which interface in the servlet API can we access the Context Parameters

ServletContext

* Context parameters can be accessed throughout the application

|  |
| --- |
| **Prepared Statement** |

A PreparedStatement is a child interface of the Statement interface.

A **PreparedStatement** is a pre-compiled SQL statement. Prepared Statement objects have some useful additional features than Statement objects. Instead of hard coding queries, PreparedStatement object provides a feature to execute a parameterized query.

**Advantages of PreparedStatement**

* When PreparedStatement is created, the SQL query is passed as a parameter. This Prepared Statement contains a pre-compiled SQL query, so when the PreparedStatement is executed, DBMS can just run the query instead of first compiling it like the Statement, which makes it fast.
* We can use the same PreparedStatement and supply with different parameters at the time of execution.
* An important advantage of PreparedStatements is that they prevent SQL injection attacks.

**Steps to use PreparedStatement**

1. Create Connection to Database

Connection myCon = DriverManager.getConnection(path,username,password);

1. Prepare Statement  
   Instead of hardcoding queries like,

select \* from students where age>10 and name ='Chhavi'

Set parameter placeholders(use question mark for placeholders) like,

select \* from students where age> ? and name = ?

PreparedStatement myStmt;

myStmt = myCon.prepareStatement(select \* from students where age> ? and name = ?);

1. Set parameter values for type and position (bind the values)

myStmt.setInt(1,10);

myStmt.setString(2,"Chhavi");

1. Execute the Query

ResultSet myRs= myStmt.executeQuery();

**Methods of PreparedStatement:**

* setInt(int, int): This method can be used to set integer value at the given parameter index.
* setString(int, string): This method can be used to set string value at the given parameter index.
* setFloat(int, float): This method can be used to set float value at the given parameter index.
* setDouble(int, double): This method can be used to set a double value at the given parameter index.
* executeUpdate(): This method can be used to create, drop, insert, update, delete etc. It returns int type.
* executeQuery(): It returns an instance of ResultSet when a select query is executed.

|  |
| --- |
| @WebServlet("/UpdateProductServlet")  public class UpdateProductServlet extends HttpServlet {  private static final long serialVersionUID = 1L;  Connection con;  PreparedStatement stmt;    public void init() {  try {  Class.forName("com.mysql.cj.jdbc.Driver");    ServletContext context = getServletContext();  String dbUrl = context.getInitParameter("dbUrl");  String dbUser = context.getInitParameter("dbUser");  String dbPassword = context.getInitParameter("dbPassword");  con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);    stmt = con.prepareStatement("update product set name = ?, description = ?, price = ? where id = ?");    } catch (ClassNotFoundException | SQLException e) {  e.printStackTrace();  }  }    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  int id = Integer.parseInt(request.getParameter("id"));  String name = request.getParameter("name");  String description = request.getParameter("description");  int price = Integer.parseInt(request.getParameter("price"));    try {  stmt.setString(1, name);  stmt.setString(2, description);  stmt.setInt(3, price);  stmt.setInt(4, id);    int result = stmt.executeUpdate();    response.setContentType("text/html");  PrintWriter out = response.getWriter();  out.print("<b>"+result+" Product(s) updated</b>");  } catch (SQLException e) {  e.printStackTrace();  }  }    public void destroy() {  try {  stmt.close();  con.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Product Update</title>  </head>  <body>  <h2>Enter Product Details</h2>  <br />  <form action=*"UpdateProductServlet"* method=*"post"*>  Id: <input type=*"text"* name=*"id"* /><br /> Name: <input type=*"text"*  name=*"name"* /><br /> Description: <input type=*"text"*  name=*"description"* /><br /> Price: <input type=*"text"* name=*"price"* /><br />  <input type=*"submit"* name=*"Update Product"* />  </form>  </body>  </html> |

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd http://xmlns.jcp.org/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <display-name>PreparedStatement</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.jsp</welcome-file>  <welcome-file>default.htm</welcome-file>  </welcome-file-list>  <context-param>  <param-name>dbUrl</param-name>  <param-value>jdbc:mysql://localhost/mydb</param-value>  </context-param>  <context-param>  <param-name>dbUser</param-name>  <param-value>root</param-value>  </context-param>  <context-param>  <param-name>dbPassword</param-name>  <param-value>P@ssW0rd</param-value>  </context-param>  </web-app> |

|  |
| --- |
| **Inter Servlet Communication** |

Inter servlet communication is nothing but communication between servlets or request dispatching.

JSP

Servlet

JSP

Servlet

The RequestDispatcher interface is used to forward request to another servlet.

RequestDispatcher rd = request.getRequestDispatcher(uri);

The uri is where you want to go next.

Servlet 1 will forward the request to Servlet 2 for further processing and Servlet 2 will then send the response to the client

There are two ways to do the request dispatching

rd.forward(request, response);

Servlet 2

Servlet 1

Client

The second way is as follows

The container will merge the response from Servlet 1 and Servlet 2 and Servlet 1 will then send the response to the client

rd.include(request, response);

+

Servlet 2

Servlet 1

Client

**Usecase**

HomeServlet

LoginServlet

forward

include

Login.html

DB

**Summary**

Which of the following interfaces in the servlet API can be used to do inter servlet communication?

* RequestDispatcher

Which of the following is not a method used for inter servlet communication?

* Forward
* Next
* Include

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Login Page</title>  </head>  <body>  <form action=*"loginServlet"* method=*"post"*>  Email: <input type=*"text"* name=*"userName"* /><br />  Password: <input type=*"password"* name=*"password"* /><br />  <input type=*"submit"* value=*"Login"* />  </form>  </body>  </html> |

|  |
| --- |
| @WebServlet("/homeServlet")  **public** **class** HomeServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  PrintWriter out = response.getWriter();  response.setContentType("text/html");  out.print(request.getAttribute("message"));  }  } |

|  |
| --- |
| @WebServlet(urlPatterns = "/loginServlet", initParams = {  @WebInitParam(name="dbUrl", value="jdbc:mysql://localhost/mydb"),  @WebInitParam(name="dbUsername", value="root"),  @WebInitParam(name="dbPassword", value="P@ssW0rd")  })  **public** **class** LoginServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Connection connection;  **private** PreparedStatement statement;  **private** ResultSet resultSet;    **public** **void** init(ServletConfig config) {  **try** {  Class.*forName*("com.mysql.cj.jdbc.Driver");    String dbUrl = config.getInitParameter("dbUrl");  String dbUsername = config.getInitParameter("dbUsername");  String dbPassword = config.getInitParameter("dbPassword");  connection = DriverManager.*getConnection*(dbUrl, dbUsername, dbPassword);    statement = connection.prepareStatement("select \* from user where email = ? and password = ?");  } **catch** (ClassNotFoundException | SQLException e) {  e.printStackTrace();  }    }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {    String userName = request.getParameter("userName");  String password = request.getParameter("password");    **try** {  statement.setString(1, userName);  statement.setString(2, password);  resultSet = statement.executeQuery();    RequestDispatcher dispatcher = request.getRequestDispatcher("homeServlet");    **if** (resultSet.next()) {  request.setAttribute("message", "Welcome to InterServlet Communication " + userName);  dispatcher.forward(request, response);  } **else** {  dispatcher = request.getRequestDispatcher("login.html");  dispatcher.include(request, response);  }  } **catch** (SQLException e) {  e.printStackTrace();  }    }    **public** **void** destroy() {  **try** {  resultSet.close();  statement.close();  connection.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| **PreInitialization of Servlets** |

There are two types of Servlet initializations

* Lazy initialization
* Pre-initialization

**Lazy initialization**

When a Container initializes a servlet only when the first request comes from a client, that is called a Lazy initialization.

Servlet

Container

Web Client

Lazy Initialization

By default, all servlets are lazily initialized.

**Pre-Initialization**

A container will initialize the servlet before any request is received.

Container

Servlet

Pre Initialization

We can request a container to do pre-initialization by configuring load-on-startup tag under the servlet tag.

|  |
| --- |
| <servlet>  <display-name>CreateUserServlet</display-name>  <servlet-name> CreateUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.CreateUserServlet</servlet-class>  <load-on-startup> 1 </load-on-startup>  </servlet>  <servlet>  <display-name>ReadUserServlet</display-name>  <servlet-name>ReadUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.ReadUserServlet</servlet-class>  <load-on-startup> 2 </load-on-startup>  </servlet> |

The numeric value tells the container the priority of the servlet. The lower the number the priority of the servlet which means it gets started first.

**Uses Cases**

Web Services using CXF or Jersey. They use on servlet which is loaded from the startup and from that point knows which servlet to be get initialized by calling the service endpoints.

Spring MVC Framework also has the DispatcherServlet which should be loaded on startup.

**Lazy Initialization Example**

|  |  |
| --- | --- |
| @WebServlet("/preInitServlet")  **public** **class** InitServer **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **public** **void** init() {  System.***out***.println("Inside the init method");  }    **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.getWriter().write("From the pre init servlet");  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  }  } | Output when container starts, servlet not initialized |
| Sep 09, 2022 6:25:44 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] has started  Sep 09, 2022 6:25:44 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] is completed |
| When request is sent to container, servlet is initialized |
| INFO: Starting ProtocolHandler ["http-nio-8080"]  Sep 09, 2022 6:30:16 PM org.apache.catalina.startup.Catalina start  INFO: Server startup in [1122] milliseconds  Inside the init method |

When the container starts, none of the Servlet methods are executed. This indicates that the container does not initialize the Servlet. Only when we execute the servlet, we get the input which indicates that the Servlet is only loaded when there is a request from the browser.

**Pre-Initialization Example Using Annotation**

|  |  |
| --- | --- |
| @WebServlet(urlPatterns = "/preInitServlet", loadOnStartup = 0)  **public** **class** InitServer **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **public** **void** init() {  System.***out***.println("Inside the init method");  }    **protected** **void** doGet(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  res.getWriter().write("From the pre init servlet");  }  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  }  } | Servet is initalized when container starts when loadOnStartup is specified |
| Sep 09, 2022 6:19:13 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] has started  Inside the init method  Sep 09, 2022 6:19:14 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] is completed |
|  |

Executes the init() method when the Container starts, therefore pre-initializing the server. The integer determines the Servlet priority. 0 indicates a higher priority and the servlet will be initialized before all the other servlets.

**Pre-Initialization Example Using Web.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <display-name>ServletInitialization</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.jsp</welcome-file>  <welcome-file>default.htm</welcome-file>  </welcome-file-list>  <servlet>  <servlet-name>InitServer</servlet-name>  <servlet-class>com.samsonmarikwa.Initilization.InitServer</servlet-class>  <load-on-startup>0</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>InitServer</servlet-name>  <url-pattern>/preInitServlet</url-pattern>  </servlet-mapping>  </web-app> |

Executes the init() method when the Container starts, therefore pre-initializing the server. The integer determines the Servlet priority. 0 indicates a higher priority and the servlet will be initialized before all the other servlets.

**Summary**

What is the name of the attribute on the WebServlet annotation that should be used for pre initialization?

* loadOnStartup
* loadOnInit
* loadOnStart
* preInit

Which of the following element can be used in the web.xml to configure pre initialization?

* load-on-start
* load-on-startup
* load-on-it
* load-fast

|  |
| --- |
| **Servlet Listeners** |

Servlet listeners are used for event handling in web applications. Once an event is invoked, the application can take appropriate action.

These events could be

Request creation and destruction

Session being created or destroyed

Context – a web application being loaded or unloaded from the container.

Async – we can also handle asynchronous events

To create a listener, we can implement the HttpSessionListener. The events specified above have specific interfaces just like the SessionListener.

class UserCountListener implement HttpSessionListener

Once we implement the class, we need to tell the container about the listener by marking it.

Annotation

@WebListener

Container

web.xml

<listener>

<listener-class> UserCountListener</listener-class>

When container starts, it scans the classpath and all the classes marked as listeners will be registered in the container as Listeners to be invoked when a certain event occurs

</listener>

Use Case

The term “**event**” refers to the occurrence of something. An event occurs when the state of an object changes. When these exceptions occur, we can conduct certain crucial actions, such as collecting total and current logged-in users, establishing database tables when deploying the project, building database connection objects, and so on. The javax.servlet and javax.servlet.http packages contain numerous Event classes and Listener interfaces.

|  |
| --- |
| **Filters** |

A Filter is a java class that can intercept a request from a client, do some processing and then pass on the request to a servlet or another filter for additional processing. A response from the servlet is handled in the same way before being passed back to the client.

A filter is pre-initialized by the container. Unlike the servlet, there is no lazy-initialization of filters. Filters are pre-initialized.

Post-processed logic applied

Browser

Container

SERVLET

JSP

F

I

L

T

E

R

Pre-processed logic applied

**Use Case**

Injection Attack Filter – this will look for string input with malicious content and the request can be rejected.

**Filter Chaining**

This is where a request is passed through two or more filters which perform different processes.

**Implementation**

@Filter annotation is used to mark a class as a filter. A web.xml file can also be used.

The life cycle of a filter is similar to that of a servlet.

Pre-Initialized by default

Container

Init(FilterConfig config)

doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

destroy()

The FilterChain has information on where the filter should send the pre-processed request, that is another filter or a servlet or JSP.

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"* xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"* version=*"4.0"*>  <servlet>  <description></description>  <display-name>FilterServletDemo</display-name>  <servlet-name>FilterServletDemo</servlet-name>  <servlet-class>com.samsonmarikwa.servlet.FilterServletDemo</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>FilterServletDemo</servlet-name>  <url-pattern>/filterDemoServlet</url-pattern>  </servlet-mapping>  <display-name>FiltersDemo</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.htm</welcome-file>  <welcome-file>default.jsp</welcome-file>  </welcome-file-list>  <filter>  <display-name>DemoFilter</display-name>  <filter-name>DemoFilter</filter-name>  <filter-class>com.samsonmarikwa.filters.DemoFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>DemoFilter</filter-name>  <url-pattern>/filterDemoServlet</url-pattern>  </filter-mapping>  </web-app> |

|  |
| --- |
| **package** com.samsonmarikwa.servlet;  //@WebServlet("/filterDemoServlet")  **public** **class** FilterServletDemo **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  response.getWriter().append("From the servlet\n");  }  } |

|  |
| --- |
| //@WebFilter("/filterDemoServlet") // "/\*" will make this filter available for all the Servlets in the application  // Filter order is as per filter-mapping entries in the web.xml  **public** **class** DemoFilter **extends** HttpFilter {    **public** **void** destroy() {  // **TODO** Auto-generated method stub  }  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  PrintWriter out = response.getWriter();  out.print("Pre Servlet\n");  chain.doFilter(request, response);  out.print("Post Servlet\n");  }  **public** **void** init(FilterConfig config) **throws** ServletException {  // **TODO** Auto-generated method stub  }  } |

|  |
| --- |
| **Session Management** |

HTTP Statelessness

The server does not maintain a continuous connection whilst it handles the request and sends back the response.

request

Web Server

Web Client

response

Each time a request comes, the server opens a fresh connection and after sending the response it destroys the connection. HTTP protocol does not keep state of the connection. There is no need to remember anything from the previous connection.

**Advantages**

* Performance – No overhead – there is no need to maintain a state of the connection even where the client is not doing anything after sending the initial request.
* Scalability – The application can be deployed on several servers because the client does not need to be connected to the server that handled the initial request as there is no state from the previous connection.

**Disadvantages**

* The server cannot remember anything, for example, a shopping cart. Hence, we have to use session tracking.

**Steps for Session Management**

1. Create the session

HttpSession session = request.getSession();

1. Maintain data using the four attribute methods on HttpSession object across multiple user requests.
2. End the session

**How session management works?**

Web Server

Web Client

**Session Tracking**

To maintain state despite the Statelessness of HTTP.

request.getSession() is a method on the HttpServletRequest object which retrieves the session object in the request. If the session object is not found, the web server creates a unique HttpSession object which is written into the HttpServletRequest object and keeps the HttpSession object in memory in a key/value map, with the session id as the key and the session object as the value. When the server returns a response, it also includes the session id sent as a cookie in JSESSIONID cookie. Subsequent requests will have all the cookies generated by the server in the initial session. The server will see the JSESSIONID cookie which is used to retrieve the session object. This process of session tracking is important for HTTP as the protocol is stateless. In this way, we are able to create a stateful web application. This is necessary for applications such as e-commerce application which needs to match requests in created a Cart.

**What is a session?**

**Login to Logout** - activities that take place from the time of logging in to logging out are termed a session.

**Track user interaction** – for example when you access a ticket booking website where you do not login. The entire process of booking and checkout of the ticket is a session.

In Java, javax.servlet.http.HttpSession represents the session object.

In session tracking, there are two things that are key:

* Client / User Identification
* State Maintenance