National Forensic Sciences University
Delhi Campus
(An Institute of National Importance)

Advance Java Programming

Practical File

Subject Code: CTMTCS SVI P6 EL2

Submitted to
Dr. Parul Arora

Submitted by
Rupam Barui
102CTMBCS2122002

Contents

Experiment No. 1: Implement Client Socket Programming

Objective

The objective of this experiment is to implement a client socket program that communicates with a server using the TCP/IP protocol.

Introduction

Socket programming is a way of connecting two nodes on a network to communicate with each other. In this experiment, we will focus on the client-side implementation using the Java programming language.

Theory

Client socket programming involves creating a socket on the client side and connecting it to a server socket. The client can then send and receive data to/from the server using input and output streams.

Algorithm

1. Create a socket object and specify the server's IP address and port number.
2. Establish a connection with the server using the socket.
3. Create input and output streams for communication.
4. Send data to the server using the output stream.
5. Receive data from the server using the input stream.
6. Close the socket connection.

Program

Client.java:

```
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 3333);
             DataInputStream din = new DataInputStream(socket.getInputStream());
```

```
            DataOutputStream dout = new
                DataOutputStream(socket.getOutputStream());
            BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in))) {

            String str = "", str2 = "";
            while (!str.equalsIgnoreCase("stop")) {
                System.out.print("Enter message: ");
                str = br.readLine();
                dout.writeUTF(str);
                dout.flush();
                str2 = din.readUTF();
                System.out.println("Server says: " + str2);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Server.java:

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Server {
    public static void main(String[] args) {
        try (ServerSocket ss = new ServerSocket(3333);
            Socket socket = ss.accept();
            DataInputStream din = new DataInputStream(socket.getInputStream());
            DataOutputStream dout = new
                DataOutputStream(socket.getOutputStream());
            BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in))) {

            String str = "", str2 = "";
            while (!str.equalsIgnoreCase("stop")) {
                str = din.readUTF();
                System.out.println("Client says: " + str);
                System.out.print("Enter reply: ");
                str2 = br.readLine();
```

```
                dout.writeUTF(str2);
                dout.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Conclusion

In this experiment, we successfully implemented a client socket program in Java. The client program establishes a connection with the server, sends a message, receives a response, and then closes the connection. This experiment demonstrates the basic principles of client-side socket programming using the TCP/IP protocol.

```
❯ cd "/Users/rupambarui/Developer/Semester-6/Advanced Java/File/ClientServer/" &&
javac Server.java && java Server
Client says: Hello from Client Side
Enter reply: Hello From Server Side
```

Figure 1: Output of the Socket Program

```
❯ cd "/Users/rupambarui/Developer/Semester-6/Advanced Java/File/ClientServer/" &
& javac Client.java && java Client
Enter message: Hello from Client Side
Server says: Hello From Server Side
Enter message:
```

Figure 2: Output of the Client Program

Experiment No. 2: JDBC Connectivity

Objective

The objective of this experiment is to demonstrate JDBC connectivity and retrieve data from a MySQL database.

Introduction

JDBC (Java Database Connectivity) is a SQL-based API to connect to relational databases and execute queries. This experiment focuses on connecting to a MySQL database, executing a SQL query, and processing the results.

Theory

JDBC connectivity involves the following steps:

1. Load the JDBC driver.
2. Establish a connection to the database using `DriverManager`.
3. Create a `Statement` object to execute SQL queries.
4. Execute the query using the `Statement` object and get the `ResultSet`.
5. Process the `ResultSet` to retrieve the desired data.
6. Close the resources.

Algorithm

1. Load the JDBC driver using `Class.forName`.
2. Establish the connection to the database using `DriverManager.getConnection`.
3. Create a `Statement` object.
4. Execute a SQL query and get the `ResultSet`.
5. Iterate through the `ResultSet` to process the data.
6. Close the `ResultSet`, `Statement`, and `Connection` objects.

Program

```
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
        final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";

        final String USER = "rupamuser";
        final String PASS = "rupampass";

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            Class.forName(JDBC_DRIVER);

            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);

            System.out.println("Creating statement...");
            stmt = conn.createStatement();
            String sql = "SELECT id, name, email FROM users";
            rs = stmt.executeQuery(sql);

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String email = rs.getString("email");

                System.out.printf("ID: %d, Name: %s, Email: %s\n", id, name,
                    email);
            }
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
```

```
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
        System.out.println("Goodbye!");
    }
}
```

Conclusion

In this experiment, we successfully demonstrated JDBC connectivity to a MySQL database. We established a connection to the database, executed a SQL query to retrieve data from the `students` table, and processed the `ResultSet` to display the results. This experiment showcases the essential steps for connecting to a relational database and executing queries using JDBC.

```
MariaDB [(none)]> CREATE DATABASE mydatabase;
Query OK, 1 row affected (0.001 sec)

MariaDB [(none)]> USE mydatabase;
Database changed
MariaDB [mydatabase]>
MariaDB [mydatabase]> CREATE TABLE users (
    ->     id INT AUTO_INCREMENT PRIMARY KEY,
    ->     name VARCHAR(50) NOT NULL,
    ->     email VARCHAR(50) NOT NULL
    -> );
Query OK, 0 rows affected (0.008 sec)

MariaDB [mydatabase]>
MariaDB [mydatabase]> INSERT INTO users (name, email) VALUES
    ->     ('John Doe', 'john.doe@example.com'),
    ->     ('Jane Smith', 'jane.smith@example.com'),
    ->     ('Bob Johnson', 'bob.johnson@example.com');
```

```
┌──(rupam㉿kali)-[~/Downloads]
└─$ javac -cp /usr/share/java/mysql-connector-j-8.4.0.jar JDBCProg.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=tru
e

┌──(rupam㉿kali)-[~/Downloads]
└─$ java -cp /usr/share/java/mysql-connector-j-8.4.0.jar:. JDBCProg
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=tru
e
Connecting to database...
Creating statement...
ID: 1, Name: John Doe, Email: john.doe@example.com
ID: 2, Name: Jane Smith, Email: jane.smith@example.com
ID: 3, Name: Bob Johnson, Email: bob.johnson@example.com
Goodbye!
```

Figure 3: JDBC Connection

Experiment No. 3: Transaction Management

Objective

The objective of this experiment is to implement transaction management in Java using JDBC.

Introduction

Transaction management ensures that a series of database operations are executed as a single logical unit. It follows the principles of ACID (Atomicity, Consistency, Isolation, Durability) to ensure reliable processing of database transactions.

Theory

Using JDBC for transaction management involves:

1. Loading the JDBC driver.
2. Establishing a connection to the database with auto-commit set to false.
3. Executing multiple SQL statements as part of a transaction.
4. Committing the transaction.
5. Rolling back the transaction in case of any failures.

Algorithm

1. Load the JDBC driver using `Class.forName`.
2. Establish the connection to the database with `DriverManager.getConnection`.
3. Set auto-commit to false using `connection.setAutoCommit(false)`.
4. Create a `Statement` object.
5. Execute update queries to perform transaction operations.
6. Commit the transaction using `connection.commit()`.
7. Roll back the transaction using `connection.rollback()` in case of any errors.
8. Close the resources (`ResultSet`, `Statement`, and `Connection`).

Program

```
import java.sql.*;

public class JDBCTransaction {
    public static void main(String[] args) {
        final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
        final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";

        final String USER = "rupamuser";
        final String PASS = "rupampass";

        Connection conn = null;
        Statement stmt = null;

        try {
            Class.forName(JDBC_DRIVER);

            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);

            conn.setAutoCommit(false);

            stmt = conn.createStatement();

            String sql1 = "INSERT INTO users (name, email) VALUES ('Alice',
                'alice@example.com')";
            String sql2 = "UPDATE users SET email = 'bob@example.com' WHERE
                name = 'Bob Johnson'";
            String sql3 = "DELETE FROM users WHERE name = 'John Doe'";

            System.out.println("Executing transaction...");
            stmt.executeUpdate(sql1);
            stmt.executeUpdate(sql2);
            stmt.executeUpdate(sql3);

            conn.commit();
            System.out.println("Transaction committed successfully.");

        } catch (SQLException se) {
            se.printStackTrace();
            try {
                System.out.println("Rolling back transaction...");
                if (conn != null) {
                    conn.rollback();
                }
```

```
            } catch (SQLException re) {
                re.printStackTrace();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
        System.out.println("Goodbye!");
    }
}
```

Conclusion

In this experiment, we successfully implemented transaction management in Java using JDBC. We demonstrated how to execute multiple SQL statements as part of a transaction, and how to commit or roll back the transaction based on the success or failure of the operations. This ensures that a series of database operations are executed reliably and consistently as a single logical unit.

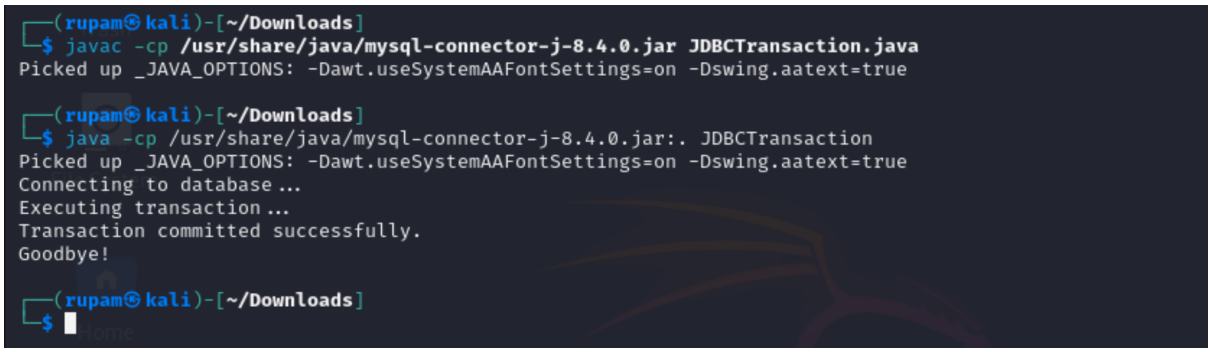

Figure 4: Output of the Transaction Management Program

Experiment No. 4: Establishing URL Connection

Objective

The objective of this experiment is to establish a URL connection and fetch content using Java.

Introduction

Java provides robust libraries for handling network connections. Using `java.net.URL` and `java.net.HttpURLConnection`, one can easily connect to a URL and fetch its content. This experiment demonstrates the key steps involved in fetching content from a URL.

Theory

Establishing a URL connection involves the following steps:

1. Create a `URL` object with the desired URL.
2. Open a connection to the URL using `HttpURLConnection`.
3. Set the request method (default is `GET`).
4. Retrieve the response code to ensure the connection is successful.
5. Fetch content from the URL using an `InputStream` and `BufferedReader`.
6. Read and print the content.
7. Close the input stream and disconnect the connection.

Algorithm

1. Define the URL string of the desired website.
2. Create a `URL` object.
3. Open a connection using `HttpURLConnection`.
4. Set the request method to `GET`.
5. Fetch the response code and check if it is `200` (HTTP OK).
6. If the connection is successful, read the content using a `BufferedReader`.
7. Print the fetched content.
8. Close the input stream and disconnect the connection.

Program

```
import java.io.*;
import java.net.*;

public class URLConnection {
    public static void main(String[] args) {
        String urlString = "https://guthib.com";

        try {
            URL url = new URL(urlString);

            HttpURLConnection connection = (HttpURLConnection)
                url.openConnection();

            connection.setRequestMethod("GET");

            int responseCode = connection.getResponseCode();

            if (responseCode == HttpURLConnection.HTTP_OK) {
                InputStream inputStream = connection.getInputStream();

                BufferedReader reader = new BufferedReader(new
                    InputStreamReader(inputStream));
                String line;
                StringBuilder content = new StringBuilder();

                while ((line = reader.readLine()) != null) {
                    content.append(line);
                    content.append(System.lineSeparator());
                }

                System.out.println(content.toString());

                reader.close();
            } else {
                System.out.println("HTTP request failed with code: " +
                    responseCode);
            }

            connection.disconnect();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Conclusion

In this experiment, we successfully established a URL connection and fetched content using Java. By following the steps outlined, we demonstrated how to connect to a URL, retrieve the response, and read the content using `HttpURLConnection`, `InputStream`, and `BufferedReader`.

```
┌──(rupam㉿kali)-[~/Downloads]
└─$ javac URLConnection.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

┌──(rupam㉿kali)-[~/Downloads]
└─$ java URLConnection
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
<style type="text/css">
h1 {
  text-align: center;
  font-size: 120px;
  font-family: Helvetica, Verdana, Arial;
}
</style>
<h1>You spelled it wrong.</h1>
```

Figure 5: Output of the URL Connection Program

Experiment No. 5: Implementing Servlet Context Interface

Objective

The objective of this experiment is to implement the Servlet Context Interface and utilize its functionality within a servlet.

Introduction

The `ServletContext` interface in Java is used to communicate with the servlet container. It allows servlets to access and share information across the entire web application. This experiment demonstrates how to set and retrieve attributes using the `ServletContext`.

Theory

Implementing the `ServletContext` involves the following steps:

1. Create a servlet and annotate it with `@WebServlet`.
2. Override the `doGet` method.
3. Obtain the `ServletContext` object using `getServletContext()`.
4. Set attributes in the `ServletContext`.
5. Retrieve attributes from the `ServletContext`.
6. Display the retrieved attributes in the browser.

Algorithm

1. Create a servlet class and annotate it with `@WebServlet` mapping.
2. Override the `doGet` method to handle GET requests.
3. Obtain the `ServletContext` object using `getServletContext()` method.
4. Set an attribute in the `ServletContext` using `setAttribute(key, value)`.
5. Retrieve the attribute using `getAttribute(key)`.
6. Write the retrieved attribute value to the HTTP response.

Program

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

@WebServlet("/servlet-context-example")
public class ServletContext extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException {
        javax.servlet.ServletContext servletContext = getServletContext();

        servletContext.setAttribute("message", "Hello from ServletContext!");

        String message = (String) servletContext.getAttribute("message");

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.println("<html><body>");
        out.println("<h1>" + message + "</h1>");
        out.println("</body></html>");
    }
}
```

Conclusion

In this experiment, we successfully implemented the `ServletContext` interface in a servlet. We demonstrated how to set and retrieve attributes using the `ServletContext` and displayed the retrieved information in the browser.

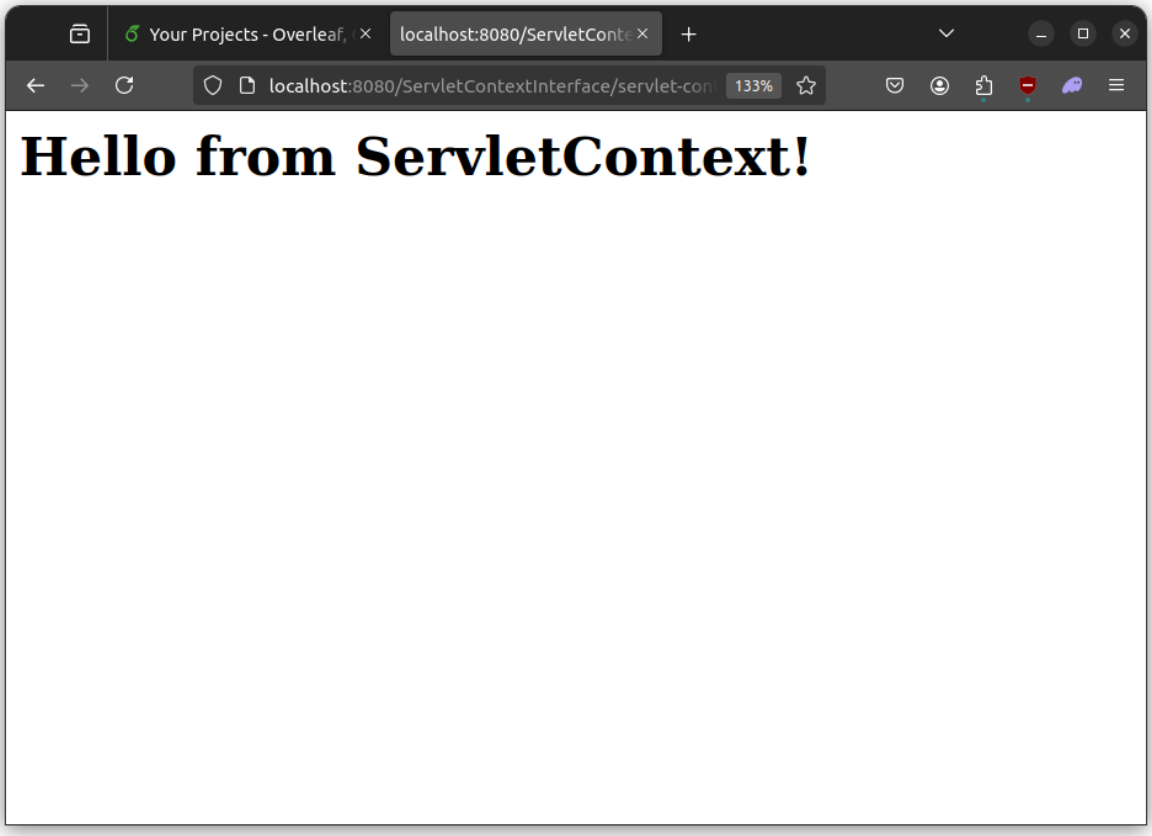

Figure 6: Output of the Servlet Context

Experiment No. 6: Implementing Servlet Config Interface

Objective

The objective of this experiment is to implement the Servlet Config Interface and utilize its functionality within a servlet.

Introduction

The `ServletConfig` interface in Java is used to pass configuration information to a servlet during initialization. This allows each servlet to have its own configuration settings as defined in the deployment descriptor (web.xml). This experiment demonstrates how to initialize and use `ServletConfig` parameters.

Theory

Implementing the `ServletConfig` involves the following steps:

1. Create a servlet class that extends `HttpServlet`.
2. Define servlet initialization parameters in the `web.xml` file.
3. Override the `init` method to read the initialization parameters using the `ServletConfig` object.
4. Use the `doGet` method to access and display the initialization parameters.

Algorithm

1. Create a servlet class and override the `init` method to initialize parameters.
2. Use `ServletConfig` to retrieve parameters defined in `web.xml`.
3. In the `doGet` method, use the parameters retrieved from `ServletConfig` to generate a response.
4. Define servlet initialization parameters in `web.xml`.

Explanation

1. The ServletConfigDemo servlet class is defined and mapped using the @WebServlet annotation with the URL pattern /configDemo.
2. Inside the init method of the servlet, the ServletConfig object config is initialized to access servlet configuration parameters.
3. Inside the init method of the servlet, the ServletConfig object config is initialized to access servlet configuration parameters.

4. In the doGet method, the servlet retrieves the initialization parameters adminEmail and welcomeMessage using config.getInitParameter() and displays them in the HTTP response.

Program

```
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("/configDemo")
public class ServletConfigDemo extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private javax.servlet.ServletConfig config;

    public void init(javax.servlet.ServletConfig config) throws
        ServletException {
        this.config = config;
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String adminEmail = config.getInitParameter("adminEmail");
        String welcomeMessage = config.getInitParameter("welcomeMessage");

        out.println("<html><body>");
        out.println("<h1>Servlet Config Demo</h1>");
        out.println("<p>Admin Email: " + adminEmail + "</p>");
        out.println("<p>Welcome Message: " + welcomeMessage + "</p>");
        out.println("</body></html>");
    }
}
```

Configuration (web.xml)

```
<servlet>
```

```
    <servlet-name>ServletConfigDemo</servlet-name>
    <servlet-class>com.example.ServletConfigDemo</servlet-class>
    <init-param>
        <param-name>adminEmail</param-name>
        <param-value>rupam@rupam.com</param-value>
    </init-param>
    <init-param>
        <param-name>welcomeMessage</param-name>
        <param-value>Hello World</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>ServletConfigDemo</servlet-name>
    <url-pattern>/configDemo</url-pattern>
</servlet-mapping>
```

Conclusion

In this experiment, we implemented the Servlet Config Interface to initialize and access configuration parameters defined in the web.xml deployment descriptor. This allows servlets to have specific configuration settings that can be accessed during their initialization phase.

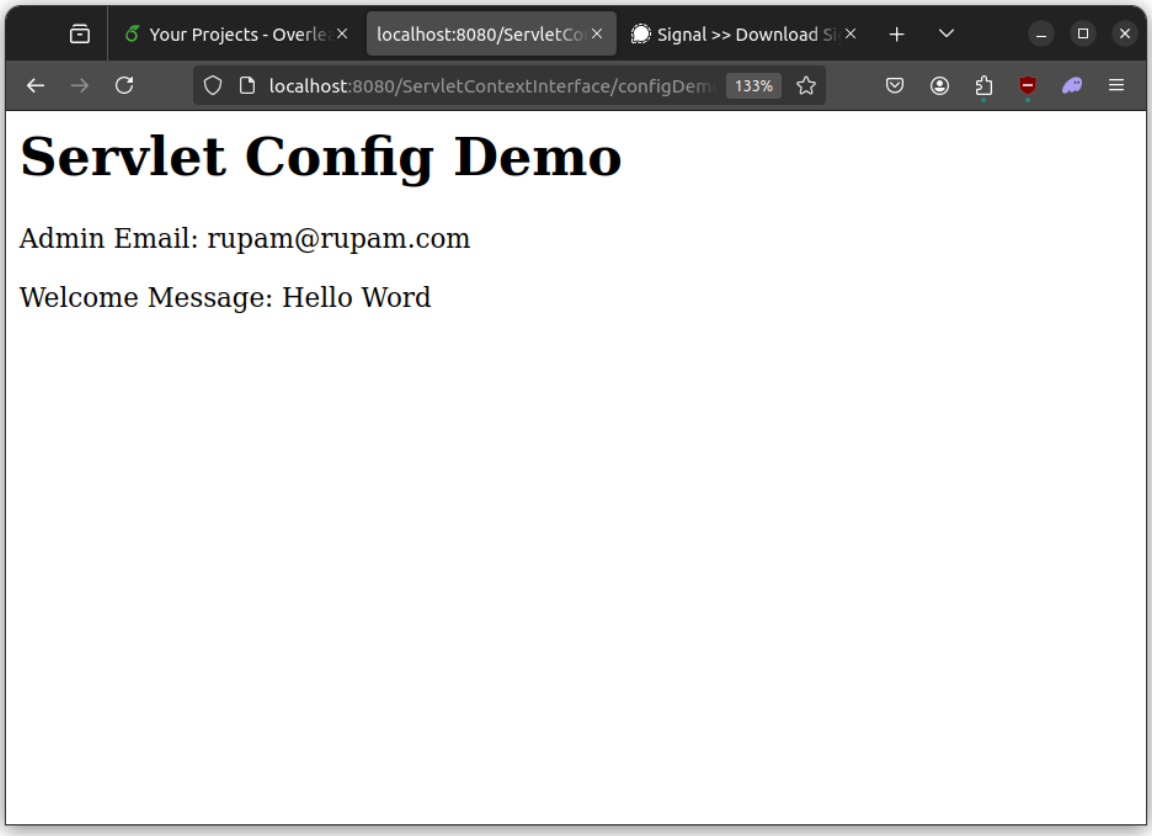

Figure 7: Output of Servlet Config Interface

Experiment No. 7: Implementing Filter Config Interface

Objective

The objective of this experiment is to implement a Filter Config Interface to log all the endpoints explored using a logging filter.

Introduction

A filter is an object that performs filtering tasks on either the request to a resource, the response from a resource, or both. The `FilterConfig` interface provides access to the filter's configuration information. This experiment demonstrates how to create a logging filter to log all incoming requests to different endpoints.

Theory

Implementing a filter involves the following steps:

1. Create a filter class that implements the `Filter` interface.
2. Override the `init` method to read the initialization parameters using the `FilterConfig` object.
3. Override the `doFilter` method to perform the logging and filtering tasks.
4. Define the filter and its initialization parameters in the `web.xml` file.
5. Map the filter to specific URL patterns in the `web.xml` file.

Algorithm

1. Create a filter class and implement the `Filter` interface.
2. Override the `init` method to initialize parameters using `FilterConfig`.
3. In the `doFilter` method, log the request details and pass the request along the filter chain using `chain.doFilter`.
4. Define the filter settings and mappings in `web.xml`.

Program

```
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.FilterServletRequest;
import javax.servlet.FilterServletResponse;
import javax.servlet.ServletException;
```

```
import java.io.IOException;

public class LoggingFilter implements Filter {
    private FilterConfig filterConfig = null;

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }

    @Override
    public void doFilter(FilterServletRequest request, FilterServletResponse
        response,
                         FilterChain chain) throws IOException, ServletException
                            {
        String uri = request.getRequestURI();
        System.out.println("Requested URI: " + uri);
        chain.doFilter(request, response); // Pass the request and response
            along the filter chain
    }

    @Override
    public void destroy() {
        this.filterConfig = null;
    }
}
```

Configuration (web.xml)

```
<filter>
    <filter-name>LoggingFilter</filter-name>
    <filter-class>LoggingFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>LoggingFilter</filter-name>
    <url-pattern>/*</url-pattern> <!-- Applies filter to all URL patterns -->
</filter-mapping>
```

Conclusion

In this experiment, we successfully implemented a FilterConfig Interface to log all incoming requests to different endpoints using a logging filter. By following the steps outlined, we demonstrated how to create a filter, initialize parameters, log request details, and configure the filter in the `web.xml` file.

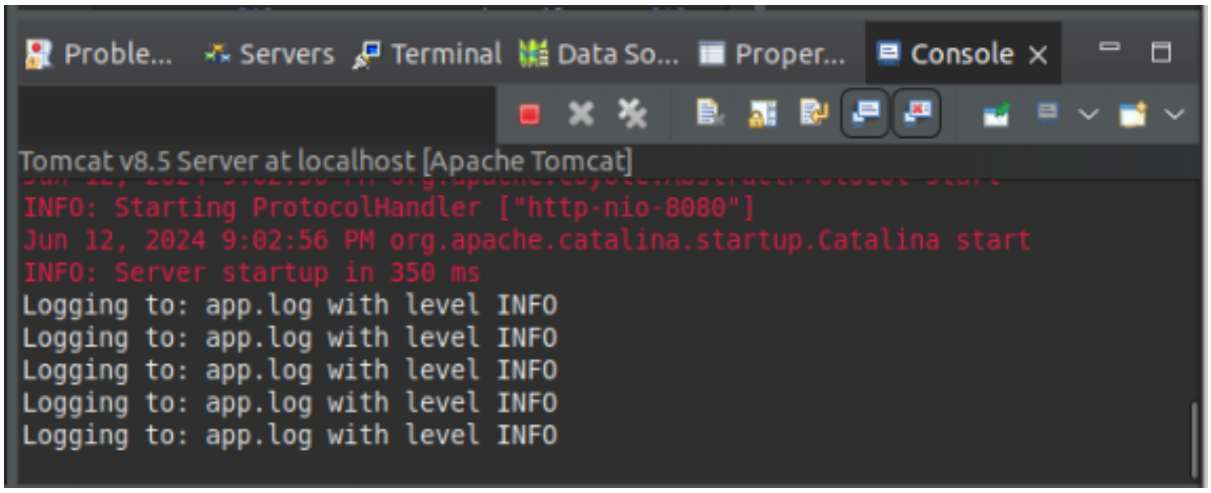

Figure 8: Output of the Logging Filter

Experiment No. 8: Session Management

Objective

The objective of this experiment is to implement a servlet application demonstrating session management.

Introduction

Session management is a way to manage user data across multiple requests in a web application. This experiment focuses on creating a to-do list using Java Servlets to demonstrate session management.

Theory

Session management allows web applications to store user-specific data for the duration of the user's session. This can be used for purposes such as authentication, maintaining user preferences, and tracking items in a to-do list.

Algorithm

1. Create a servlet class that extends `HttpServlet`.
2. Override the `doGet()` method to handle user requests.
3. Retrieve the session object using `request.getSession()`.
4. Retrieve or create a list of to-do items in the session.
5. Add new to-do items to the list based on user input.
6. Generate HTML to display the to-do list and input form.

Program

```
package com.example.servlets;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.time.LocalDateTime;
import java.util.Arrays;
import java.util.List;
```

```
@WebServlet(name = "SessionServlet", urlPatterns = {"/SessionServlet"})
public class SessionServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
            throws ServletException, IOException {
        // Get the current session, or create one if it doesn't exist
        HttpSession session = request.getSession(true);

        // Set session attributes
        session.setAttribute("userRole", "Admin");
        session.setAttribute("lastLoginTime", LocalDateTime.now().toString());
        session.setAttribute("fullName", "John Doe");
        session.setAttribute("email", "johndoe@example.com");

        // Create a list of recent activities
        List<String> recentActivities = Arrays.asList("Logged in", "Viewed
            profile", "Edited settings");
        session.setAttribute("recentActivities", recentActivities);

        // Redirect to a new page
        response.sendRedirect("session.jsp");
    }
}
```

Configuration (web.xml)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="java.util.List" %>
<html>
<head>
    <title>Session Details</title>
</head>
<body>
    <h1>Session Details</h1>
    <p>User Role: <%= session.getAttribute("userRole") %></p>
    <p>Last Login Time: <%= session.getAttribute("lastLoginTime") %></p>
    <p>Full Name: <%= session.getAttribute("fullName") %></p>
    <p>Email: <%= session.getAttribute("email") %></p>
    <h2>Recent Activities:</h2>
    <ul>
        <%
```

```
            List<String> recentActivities = (List<String>)
                session.getAttribute("recentActivities");
            if (recentActivities != null) {
                for (String activity : recentActivities) {
                    out.println("<li>" + activity + "</li>");
                }
            } else {
                out.println("<li>No recent activities found.</li>");
            }
        %>
    </ul>
</body>
</html>
```

Conclusion

In this experiment, we successfully implemented session management using Java Servlets to create a to-do list. By following the outlined steps, we demonstrated how to manage user-specific data throughout their session, allowing for functionalities such as adding items to a to-do list and displaying the contents across multiple requests.

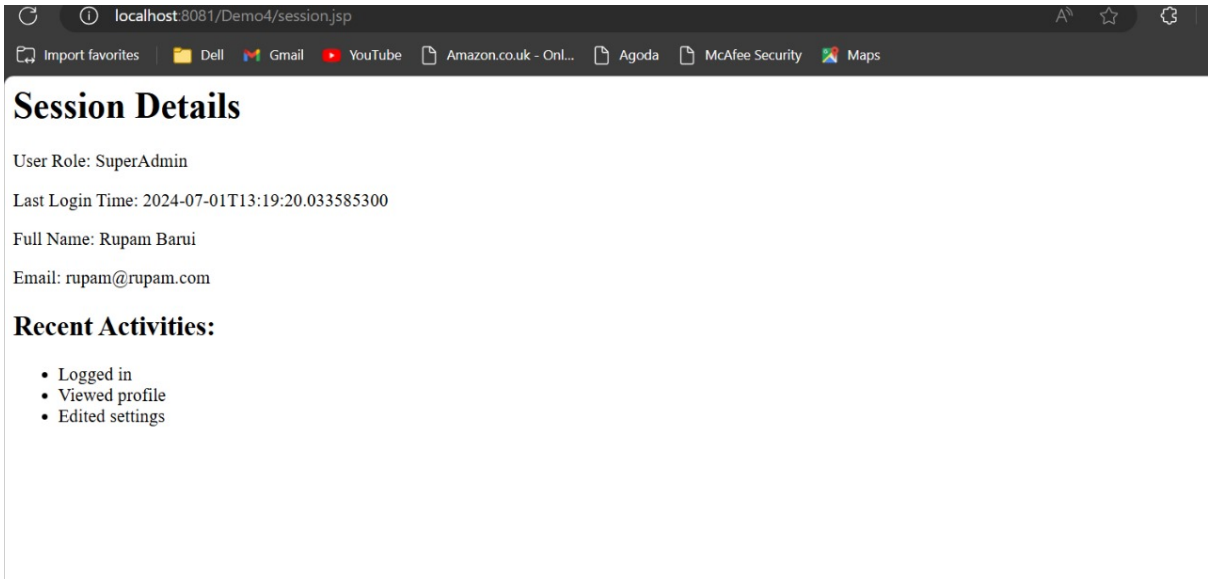

Figure 9: Output of the Session Management

Experiment No. 9: Implementing JSP Implicit Objects

Objective

The objective of this experiment is to implement JSP implicit objects to create a dynamic web page.

Introduction

JSP (JavaServer Pages) provides several implicit objects that allow developers to access various server-side functionalities without explicitly declaring them. These objects simplify the process of writing server-side code and promote rapid web application development.

Theory

JSP implicit objects are automatically available in JSP pages and include objects such as ‘request‘, ‘response‘, ‘session‘, ‘application‘, ‘out‘, ‘config‘, ‘page‘, and ‘pageContext‘. These objects provide various functionalities, including handling client requests, managing server responses, maintaining session data, and accessing application-wide parameters.

Algorithm

Steps to implement JSP implicit objects:

1. Create a JSP file and declare the page directive.
2. Use the ‘request‘ object to obtain client information.
3. Use the ‘response‘ object to set the response MIME type.
4. Use the ‘session‘ object to store and retrieve user-specific data.
5. Use the ‘application‘ object to share data across the web application.
6. Use the ‘out‘ object to write data to the response.
7. Use the ‘config‘ object to access Servlet configuration information.
8. Use the ‘page‘ object to refer to the current JSP instance.
9. Use the ‘pageContext‘ object to access page-scoped attributes and other contextual information.

Implementation

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>JSP Implicit Objects </title>
</head>
<body>
    <h1>JSP Implicit Objects </h1>

    <h2>Request Object</h2>
    <p>Client IP Address: <%= request.getRemoteAddr() %></p>
    <p>Request Method: <%= request.getMethod() %></p>

    <h2>Response Object</h2>
    <%
        response.setContentType("text/html;charset=UTF-8");
        out.println("Response MIME type set to text/html");
    %>

    <h2>Session Object</h2>
    <%
        session.setAttribute("username", "Rupam Barui");
    %>
    <p>Username from Session: <%= session.getAttribute("username") %></p>

    <h2>Application Object</h2>
    <%
        application.setAttribute("appName", "JSP Implicit Application
            Application");
    %>
    <p>Application Name: <%= application.getAttribute("appName") %></p>

    <h2>Out Object</h2>
    <%
        out.println("This is printed using the out object.");
    %>

    <h2>Config Object</h2>
    <p>Servlet Name: <%= config.getServletName() %></p>

    <h2>Page Object</h2>
    <p>Current JSP Page: <%= page.toString() %></p>
```

```
    <h2>PageContext Object</h2>
    <%
        pageContext.setAttribute("pageAttribute", "This is a page context
            attribute");
    %>
    <p>Page Context Attribute: <%= pageContext.getAttribute("pageAttribute")
        %></p>
</body>
</html>
```

Conclusion

In this experiment, we successfully implemented JSP implicit objects to create a dynamic web page. We demonstrated how to use each implicit object to access various server-side functionalities, enhancing the dynamic capabilities of JSP pages.

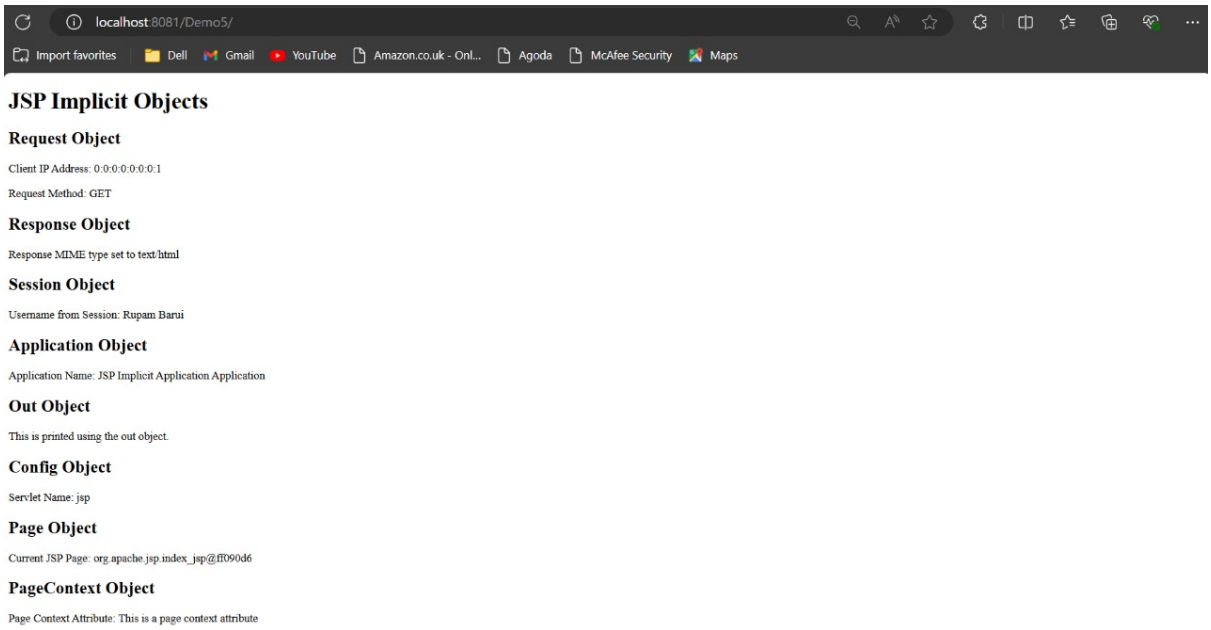

Figure 10: Output of the JSP Implicit Objects

Experiment No. 10: Hibernate Mapping

Objective

The objective of this experiment is to implement a simple Java application using Hibernate to demonstrate object-relational mapping (ORM) for data persistence.

Introduction

Hibernate is a powerful and popular ORM framework that simplifies the process of interacting with relational databases in Java applications. This experiment focuses on creating a basic application that stores and retrieves employee data using Hibernate.

Theory

Hibernate's core functionality revolves around the concept of mapping Java objects (entities) to database tables. This mapping allows developers to manipulate data in a more object-oriented manner, reducing the need to write complex SQL queries. Hibernate handles the tedious tasks of translating object operations into database actions, providing a streamlined and efficient approach to data persistence.

Algorithm

1. Create Java entities to represent the database tables.
2. Define the mapping between entities and database tables using annotations.
3. Configure Hibernate with database connection details.
4. Create a Hibernate session factory to manage database sessions.
5. Open a session and perform CRUD (Create, Read, Update, Delete) operations on entities.
6. Close the session and session factory.

Program

```
package com.example;

import com.example.model.Book;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
```

```
        SessionFactory sessionFactory = new
            Configuration().configure().buildSessionFactory();

        // Create a new book
        Book book = new Book();
        book.setTitle("The Great Gatsby");
        book.setAuthor("F. Scott Fitzgerald");

        // Save the book
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        session.save(book);
        transaction.commit();
        session.close();

        System.out.println("Book saved successfully!");

        // Retrieve the book
        session = sessionFactory.openSession();
        Book retrievedBook = session.get(Book.class, book.getId());
        System.out.println("Retrieved book: " + retrievedBook.getTitle() + "
            by " + retrievedBook.getAuthor());
        session.close();

        sessionFactory.close();
    }
}
```

Configuration (web.xml)

```
package com.example.model;

import javax.persistence.*;

@Entity
@Table(name = "books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

```
    @Column(name = "author", nullable = false)
    private String author;

    // Constructors
    public Book() {}

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", author='" + author + '\'' +
                '}';
    }
}
```

Configuration (web.xml)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>library-project</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>library-project</name>
  <url>http://maven.apache.org</url>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.32.Final</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.26</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
```

```
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>3.0.0</version>
        <configuration>
          <mainClass>com.example.App</mainClass>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Configuration (web.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
            name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property
            name="hibernate.connection.url">jdbc:mysql://localhost:3306/library_db</property>
        <property name="hibernate.connection.username">libraryuser</property>
        <property name="hibernate.connection.password">password</property>
        <property
            name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- List your entity classes here -->
        <mapping class="com.example.model.Book"/>
    </session-factory>
</hibernate-configuration>
```

Conclusion

In this experiment, we successfully implemented a basic Java application using Hibernate for object-relational mapping (ORM). The application demonstrates how to store and retrieve em-

ployee data in a relational database with ease, leveraging Hibernate's ORM capabilities.

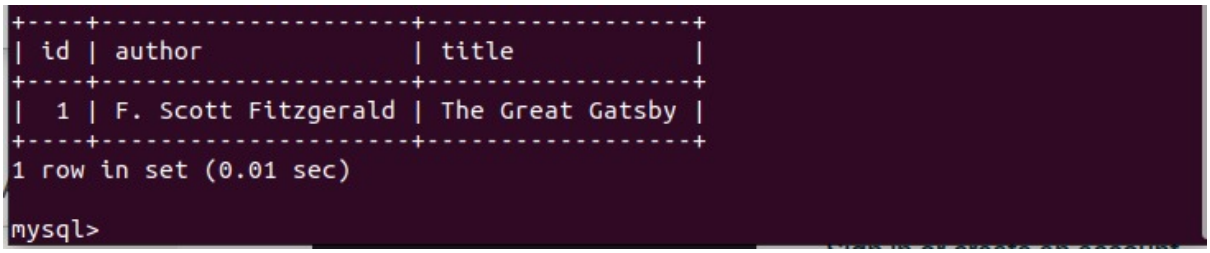

Figure 11: SQL Terminal

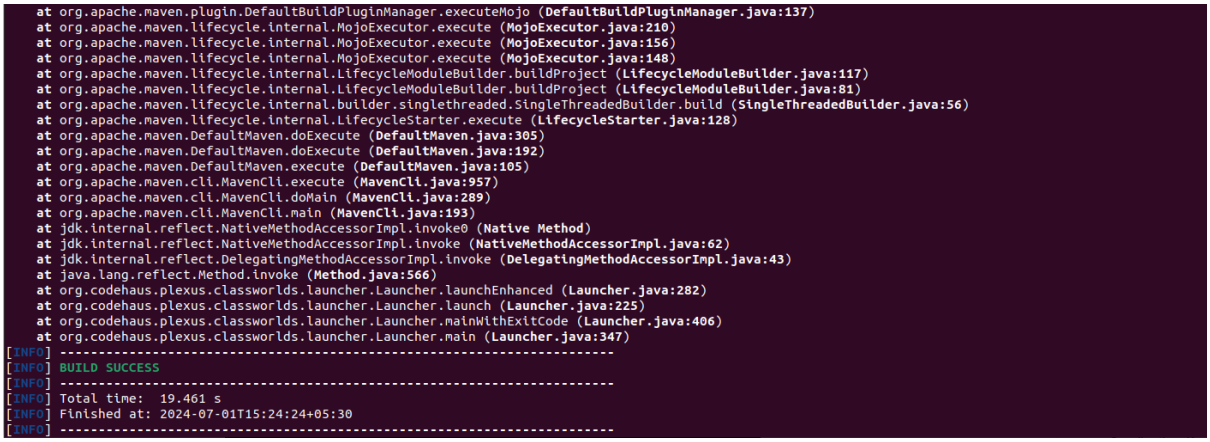

Figure 12: Output of the Hibernate Mapping Program

Experiment No. 11: Spring Transaction Management

Objective

The objective of this experiment is to implement a simple Java Spring application demonstrating transaction management using MySQL.

Introduction

Spring Transaction Management is a powerful mechanism that helps to ensure data integrity and consistency in Java applications. This experiment focuses on creating a basic Spring Boot application that manages employee data with transactions to demonstrate how to handle transactional operations efficiently.

Theory

Spring's transaction management framework provides a consistent programming model across different transaction APIs, such as JPA, JDBC, and JTA. By using declarative transaction management, developers can manage transactions with minimal boilerplate code. Spring takes care of starting, committing, or rolling back transactions using annotations or XML configuration.

Algorithm

1. Create a Spring Boot application using Spring Initializr.
2. Add dependencies for Spring Data JPA, MySQL, and Spring Web in 'pom.xml'.
3. Configure MySQL with database connection details in 'application.properties'.
4. Create an 'Employee' entity class.
5. Create a Spring Data JPA repository for 'Employee' entity.
6. Create a service class to manage transactions.
7. Create a controller class to expose REST endpoints.
8. Write methods in the service class to demonstrate transaction management, such as adding an employee and handling errors to demonstrate rollback.
9. Test the application using Postman or cURL.

Program

```java
package com.example;

import com.example.model.User;
import com.example.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringTransactionManagementApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringTransactionManagementApplication.class,
            args);
    }

    @Bean
    public CommandLineRunner demo(UserService userService) {
        return args -> {
            userService.createUser("Rupam Barui");
            userService.createUser("Oishiki Mondal");
            userService.createUser("Satyam Kumar Prasad");
            userService.createUser("Aviral Kaintura");
            userService.createUser("Kushagra Singh");
            User user1 = userService.getUser(1L);
            User user2 = userService.getUser(2L);
            User user3 = userService.getUser(3L);
            User user4 = userService.getUser(4L);
            User user5 = userService.getUser(5L);
            System.out.println("User List : ");
            System.out.println("User 1: " + user1.getName());
            System.out.println("User 2: " + user2.getName());
            System.out.println("User 3: " + user3.getName());
            System.out.println("User 4: " + user4.getName());
            System.out.println("User 5: " + user5.getName());
        };
    }
}
```

Conclusion

In this experiment, we successfully implemented a basic Spring Boot application using Spring's transaction management framework. The application demonstrates how to manage employee data with transactions, ensuring data integrity and consistency through declarative transaction management.

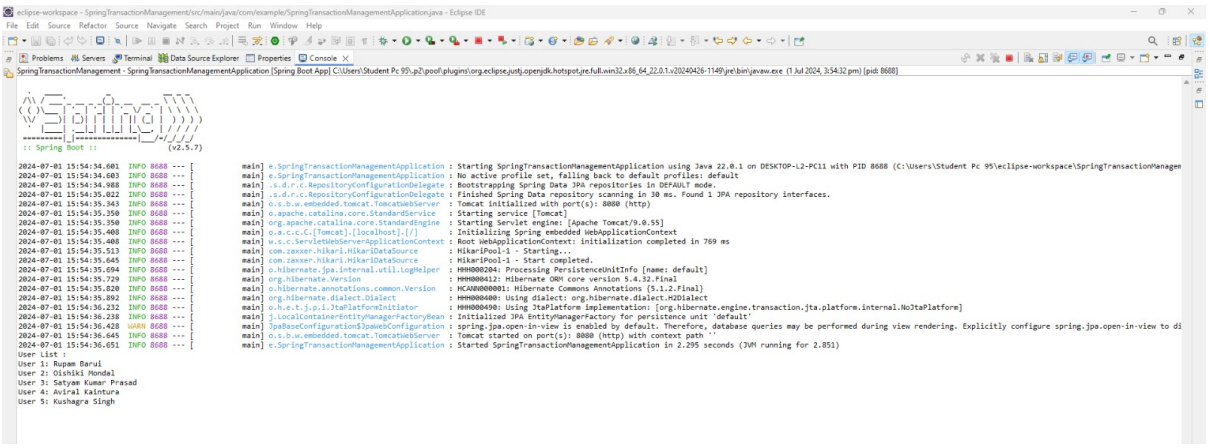

Figure 13: Output of the Spring Transaction Management Program