5COSC022W.2 Client-Server Architectures

Tutorial Week 08: RESTful web services with JAX-RS

INTRODUCTION

In this tutorial we will implement a RESTful web service using JAX-RS. You will learn how to implement all HTTP methods like GET, POST, PUT, and DELETE.

REQUIREMENTS

- Basic knowledge of Java
- NetBeans 18 or above
- Apache Tomcat server

EXERCISE 1

In this exercise, you will create a web application project that represent a simple student management system including three java classes: **Student**, **StudentApplication**, and **StudentResource**. To develop this exercise, please do the following steps:

STEP 1: CREATE PROJECT

- Create a new Maven Web Application project in NetBeans and name it as **StudentResource**
- Select the **Apache Tomcat** as a server and also select **Java EE 8** from dropdown list.
- After the project is created, please do the remaining steps as follows to complete the project.

STEP 2: CREATE STUDENT CLASS

1. Create a class called **Student**
2. **Variables:**
   - firstName (type: String): Stores a student's first name.
   - lastName (type: String): Stores a student's last name.
3. **Constructors:**
   - Student(String firstName, String lastName): This constructor allows you to create a student object and immediately provide their first and last name.
   - Student(): This is a default constructor with no arguments. It is often used to create a "blank" student object before setting its data.
4. **Methods**

- getFirstName(): Returns the value of the firstName variable.
- setFirstName(String firstName): Sets the value of the firstName variable.
- getLastName(): Returns the value of the lastName variable.
- setLastName(String lastName): Sets the value of the lastName variable.

CREATE STUDENTAPPLICATION CLASS

1. Create a class called **StudentApplication** that extends **ResourceConfig:**

This is a Java class that extends javax.ws.rs.core.Application and acts as a configuration class for your JAX-RS application. It defines which resource classes are registered with the application and which properties are set for the application. Your application class inherits from **ResourceConfig** to indicate that it's the configuration for your Jersey application.

2. Create a default constructor for the class and include the following methods:
   - register(StudentResource.class): This is where you tell Jersey about the RESTful "resources" that your application will serve. You would replace StudentResource with the actual name of your resource class.
   - property(ServerProperties.TRACING, "ALL"): This line enables detailed tracing of how Jersey processes requests, which can be helpful for debugging.
3. Save the class

CREATE STUDENTRESOURCE CLASS

Create a class called **StudentResource.** We will implement the class later.

ADDING DEPENDENCIES AND PLUGINS:

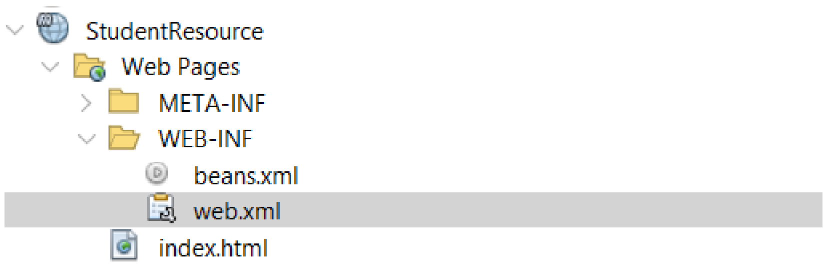
Please add the following dependencies and plugins into **pom.xml**.

```
<dependencies>

        <dependency>

            <groupId>org.glassfish.jersey.inject</groupId>

            <artifactId>jersey-hk2</artifactId>

            <version>2.32</version>

        </dependency>

        <dependency>

            <groupId>org.glassfish.jersey.containers</groupId>

            <artifactId>jersey-container-servlet</artifactId>

            <version>2.32</version>
```

```xml
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jackson</artifactId>
        <version>2.32</version> <!-- Adjust version as needed -->
    </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.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.2</version>
            <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
    </plugins>
</build>
```

CONFIGURING WEB.XML

In the previous tutorial, we created a class to add application path. This time, instead of creating that class, we can add the application path by adding Servlet and Servlet-mapping to **web.xml** file under **WEB-INF** folder in your project.



After locating the web.xml file, you need to only add the only following lines specified by **Blue.**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <servlet>
        <servlet-name>StudentApplication</servlet-name>
        <servlet-class>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>YOUR_PACKAGE-NAME</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
```

```
        <servlet-name>StudentApplication</servlet-name>

        <url-pattern>/rest/*</url-pattern>

    </servlet-mapping>

</web-app>
```

- Please note that you need to replace your package name with the placeholder specified by orange.
- Please make sure you have saved both **web.xml** and **pom.xml**

THE PURPOSE OF WEB.XML FILE

- The **web.xml** file is a configuration file for Java web applications.
- It defines servlets, filters, listeners, and other web-related components.
- In this specific example, web.xml specifies the following:
  - The servlet named **StudentApplication** (implemented by ServletContainer from Jersey) and its initialization parameters.
  - The URL pattern **/rest/*** mapped to the StudentApplication servlet.

The combination of servlets and the servlet container allows you to build dynamic web applications, process requests, and generate responses efficiently. The servlet container manages the lifecycle of servlets and ensures proper communication between clients and the application.

STRUCTURE THE STUDENT RESOURCE CLASS BY IMPLEMENTING THE FOLLOWING COMPONENTS:

1. **Define a Student Class:**
   - Create a new Java class named **Student**.
   - Add private fields for **id**, **firstName**, **lastName**, etc.
   - Generate constructors and getter/setter methods for these fields.
2. **Initialize Students:**
   - Create a new class called **StudentResource**
   - Before the class, define a path for the resource like **student**

- In the **StudentResource** class, create a static **Map<Integer, Student>** named **students**.
- Add some sample students to the map in a static block. For example:

```
students.put(1, new Student("John", "Doe"));
```

3. **GET Method to Retrieve All Students:**
    - Implement a method named **getAllStudents** in the **StudentResource** class.
    - Annotate the method with **@GET**, **@Produces(MediaType.APPLICATION_JSON)**, and **@Path("/all")**.
    - Return the collection of all students. The return type for the method must be Collection.
4. **GET Method to Retrieve a Student by ID:**
    - Implement a method named **getStudentById** in the **StudentResource** class.
    - Annotate the method with **@GET**, **@Produces(MediaType.APPLICATION_JSON)**, and **@Path("/{userId}")**.
    - Use **@PathParam** to retrieve the **userId** from the URL.
    - Return the student with the specified ID.
5. **PUT Method to Update a Student:**
    - Implement a method named **updateStudent** in the **StudentResource** class.
    - Annotate the method with **@PUT**, **@Consumes(MediaType.APPLICATION_JSON)**, and **@Path("/update/{userId}")**.
    - Use **@PathParam** to retrieve the **userId** from the URL.
    - Update the existing student's information based on the provided **updateRequest**.
    - Return a message indicating the success or failure of the update.
6. **POST Method to Create a New Student:**
    - Implement a method named **createStudent** in the **StudentResource** class.
    - Annotate the method with **@POST**, **@Consumes(MediaType.APPLICATION_JSON)**, and **@Path("/create")**.
    - Generate a new student ID and use it to add the new user to the **students** map.

- Return a message indicating the success of the creation.

7. **DELETE Method to Delete a Student:**

   - Implement a method named **deleteStudent** in the **StudentResource** class.
   - Annotate the method with **@DELETE** and **@Path("/delete/{userId}")**.
   - Use **@PathParam** to retrieve the **userId** from the URL.
   - Remove the student with the specified ID from the **students** map.
   - Return a message indicating the success or failure of the deletion.

TESTING THE API

TEST THE API USING CURL COMMAND

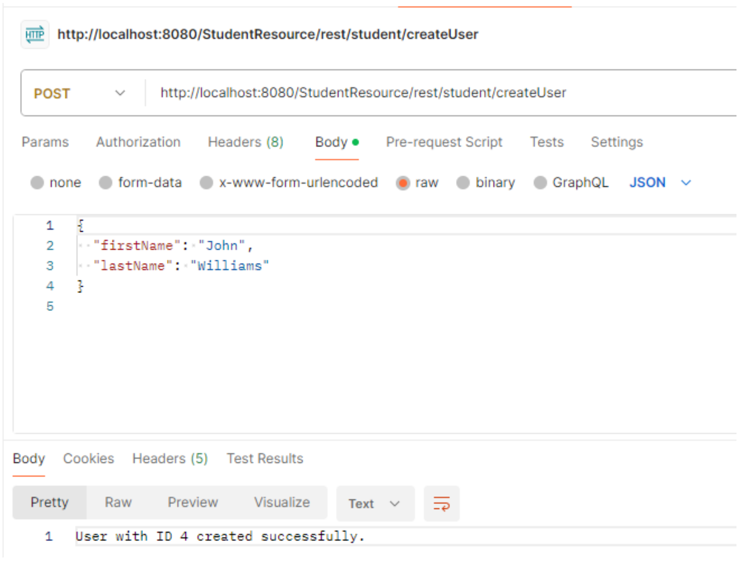
- You need to open cmd in Windows or terminal in Mac and use the following commands to send GET and POST requests.

```
curl -X GET http://localhost:8080/StudentResource/rest/student/allUsers
```

```
curl -X POST -H "Content-Type: application/json" -d "{\"firstName\":
\"John\", \"lastName\": \"Klich\"}"
http://localhost:8080/StudentResource/rest/student/createUser
```

- Please note that in the POST request, you need to send a JSON payload to the server using firstName and lastname.

TEST THE API USING POSTMAN



http://localhost:8080/StudentResource/rest/student/createUser

POST http://localhost:8080/StudentResource/rest/student/createUser

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
  "firstName": "John",
  "lastName": "Williams"
}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text

```
User with ID 4 created successfully.
```

EXERCISE 2

In this exercise, you will improve the exercise one by adding logging functionality. To do this:

1. please add the following dependencies to the pom.xml file.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version> <!-- Adjust the version as needed -->
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.6</version> <!-- Adjust the version as needed -->
</dependency>
```

2. Import logging classes to **StudentResource** class

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

3. You can now use loggers in each method
4. Run the application to see the effects

APPENDIX

1. **Servlet**:

    - A **servlet** is a Java class that handles HTTP requests and generates HTTP responses.
    - It plays a central role in web applications by processing client requests (e.g., from browsers or mobile apps) and producing appropriate responses.
    - Servlets are part of the **Java EE (Enterprise Edition)** framework and are used for web development.
    - In the given example, the servlet named StudentApplication is specified with its fully qualified class name (org.glassfish.jersey.servlet.ServletContainer).
    - The servlet class is responsible for handling incoming requests related to the /rest/* URL pattern.

2. **Servlet Container**:

    - A **servlet container** (also known as a **web container** or **servlet engine**) is a runtime environment that manages the execution of servlets.
    - It provides services such as request handling, thread management, and lifecycle management for servlets.
    - Common servlet containers include **Apache Tomcat**, **Jetty**, and **WildFly**.
    - Here's how the servlet container interacts with servlets:
        - **Initialization**: When the container starts, it creates an instance of ServletContext. This object serves as the container's memory and keeps track of all servlets, filters, and listeners associated with the web application (as defined in web.xml or equivalent annotations).
        - **Lifecycle Management**: The container initializes servlets based on their load-on-startup parameter. If this parameter has a value greater than zero, the server initializes the servlet during startup. Otherwise, the servlet's init() method is called when the first request hits it.
        - **Request Handling**: When a client sends an HTTP request (e.g., via a browser), the container creates HttpServletRequest and HttpServletResponse objects. These objects encapsulate the request data and provide methods for generating the response.
        - **Dispatching**: The container directs incoming requests to the appropriate servlet's service() method. Based on the HTTP request type (e.g., GET, POST), the container further delegates the request to the corresponding doGet() or doPost() method within the servlet.

- **Shutdown**: When the container stops or terminates, it tears down the servlets using their destroy() method. After this point, the servlets can no longer accept incoming requests.