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** 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 **Tutorial_Week08_EX01**
- 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:**
    - id (type: String): Stores a student's id.
    - firstName (type: String): Stores a student's first name.
    - lastName (type: String): Stores a student's last name.
3. **Constructors:**
    - Student(String id, String firstName, String lastName): This constructor allows you to create a student object and immediately provide their ID, first name, and last name. Note: The ID will typically be generated automatically, but this constructor is still useful.

4. Student(): This is a default constructor with no arguments. It's required by JAX-RS (specifically, by Jackson for JSON deserialization) and is often used to create a "blank" student object before setting its data via setter methods.

5. **Methods**

    - getId(): Returns the value of the id variable.
    - setId(String id): Sets the value of the id variable.
    - 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.
    - toString(): Overrides the default toString() method to provide a more informative string representation of the Student object. This is helpful for debugging.

CREATE STUDENTRESOURCE CLASS

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

ADDING DEPENDENCIES AND PLUGINS:

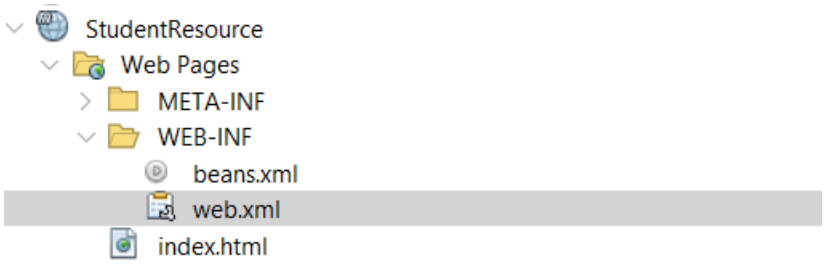
Please add the following dependencies and plugins into **pom.xml**. Please modify your pom file to include highlighted dependencies and plugins in yellow.

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>Tutorial_Week08_EX01</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>Tutorial_Week08_EX01-1.0-SNAPSHOT</name>

    <dependencies>
        <!-- JAX-RS -->
        <dependency>
            <groupId>org.glassfish.jersey.inject</groupId>
            <artifactId>jersey-hk2</artifactId>
            <version>2.32</version>
        </dependency>

        <!-- JAX-RS Implementation (Jersey) -->
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-servlet</artifactId>
```

```
            <version>2.32</version>
        </dependency>

        <!--Jackson for JSON -->


        <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>
</project>
```

CONFIGURING WEB.XML

In the previous tutorial, we created two class to add application path and configuration. This time, instead of creating those classes, we can add the application path and configurations 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 add only the following lines specified by Yellow.

```xml
<?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. Class Setup and Path Definition:

- Create a new Java class named `StudentResource`.
- Add the `@Path("/students")` annotation *before* the class definition. This sets the base URI path for all operations related to this resource.

2. Data Storage (ConcurrentHashMap and Initial Data):

- Declare the studentStore: Inside the StudentResource class, declare a static ConcurrentHashMap to store the student data

```
private static final ConcurrentHashMap<String, Student> studentStore = new
ConcurrentHashMap<>();
```

  - **`ConcurrentHashMap`:** This is a thread-safe map, crucial for web applications where multiple requests might access the data concurrently.
  - **`String` Key:** The key is the student's ID (a `String`).
  - **`Student` Value:** The value is the `Student` object itself.
- **Static Initializer Block:** Add a static initializer block to pre-populate the studentStore with some sample student data:

```
static {
    addInitialStudents();
}

private static void addInitialStudents() {
    Student student1 = new Student(UUID.randomUUID().toString(), "Alice",
"Smith");
    Student student2 = new Student(UUID.randomUUID().toString(), "Bob",
"Johnson");
    Student student3 = new Student(UUID.randomUUID().toString(), "Charlie",
"Brown");

    studentStore.put(student1.getId(), student1);
    studentStore.put(student2.getId(), student2);
    studentStore.put(student3.getId(), student3);
}
```

  - The `static { ... }` block runs *once* when the class is loaded.
  - The `addInitialStudents()` method creates `Student` objects *with unique IDs generated using* `UUID.randomUUID().toString()`. This is very important.

- The students are added to the `studentStore`.

3. GET Method to Retrieve All Students:

- **Method Signature:** Create a method named `getAllStudents`.
- **Annotations:**
  - `@GET`: Indicates that this method handles HTTP GET requests.
  - `@Produces(MediaType.APPLICATION_JSON)`: Specifies that the method returns data in JSON format.
  - *No `@Path` annotation is needed here because the class-level `@Path` covers the base path.*
- **Return Type:** The method should return a `List<Student>`.
- **Implementation:** Return an `ArrayList` containing all the values (which are `Student` objects) from the `studentStore`.

4. GET Method to Retrieve a Student by ID:

- **Method Signature:** Create a method named `getStudentById`.
- **Annotations:**
  - `@GET`
  - `@Produces(MediaType.APPLICATION_JSON)`
  - `@Path("/{id}")`: This defines a path parameter named `id`. The `{id}` part is a placeholder that will be replaced with the actual ID in the URL.
  - `@PathParam("id") String id`: This annotation extracts the value of the `id` path parameter and makes it available as a `String` variable named `id` within the method.
- **Return Type:** The method should return a `Response` object (from `javax.ws.rs.core.Response`). This allows you to control the HTTP status code and response body.
- **Implementation:**
  1. Retrieve the student from `studentStore` using the provided `id`.

2. If the student is found, return a `Response` with status code 200 (OK) and the student object as the entity.
3. If the student is *not* found, return a `Response` with status code 404 (Not Found) and an appropriate error message.

5. POST Method to Create a New Student:

- **Method Signature:** Create a method named `createStudent`.
- **Annotations:**
  - `@POST`: Indicates that this method handles HTTP POST requests.
  - `@Consumes(MediaType.APPLICATION_JSON)`: Specifies that the method accepts data in JSON format (in the request body).
  - `@Produces(MediaType.APPLICATION_JSON)`: Specifies that the method returns data in JSON format.
  - *No `@Path` annotation is needed; the class-level `@Path` applies.*
- **Parameter:** The method should accept a `Student` object as a parameter. This `Student` object will be created from the JSON data in the request body.
- **Return Type:** Return a `Response` object.
- **Implementation:**
  1. **Validate Input:** Check if the required fields (`firstName` and `lastName`) in the input `Student` object are not null. If they are null, return a 400 (Bad Request) response with an error message.
  2. **Generate ID:** Generate a unique ID for the new student using `UUID.randomUUID().toString()`.
  3. **Set ID:** Set the ID of the input `Student` object to the generated ID.
  4. **Add to `studentStore`:** Add the `Student` object to the `studentStore`.
  5. Return a `Response` with status code 201 (Created) and the newly created `Student` object (including the generated ID) as the entity.

6. PUT Method to Update a Student:

- **Method Signature:** Create a method named `updateStudent`.
- **Annotations:**
  - `@PUT`: Indicates that this method handles HTTP PUT requests.
  - `@Consumes(MediaType.APPLICATION_JSON)`
  - `@Produces(MediaType.APPLICATION_JSON)`
  - `@Path("/{id}")`: Defines a path parameter for the student ID.
  - `@PathParam("id") String id`: Extracts the student ID from the URL.
- **Parameters:**
  - `@PathParam("id") String id`: The ID of the student to update.
  - `Student updatedStudent`: The `Student` object containing the updated data.
- **Return Type:** Return a `Response` object.
- **Implementation:**
  1. **Check if Student Exists:** Check if a student with the given `id` exists in the `studentStore`.
  2. **If Not Found:** If the student doesn't exist, return a 404 (Not Found) response.
  3. **If Found:**
     - Retrieve the existing `Student` object from the `studentStore`.
     - Update the *existing* student's fields (firstName, lastName) with the values from the `updatedStudent` object. *Only update fields that are not null in the `updatedStudent` object* (this allows for partial updates).
     - Put the *updated* (existing) `Student` object back into the `studentStore` (this overwrites the old entry).
     - Return a 200 (OK) response with the updated `Student` object.

7. DELETE Method to Delete a Student:

- **Method Signature:** Create a method named `deleteStudent`.
- **Annotations:**
  - `@DELETE`
  - `@Path("/{id}")`

- `@PathParam("id") String id`

- **Parameter:** `@PathParam("id") String id`: The ID of the student to delete.
- **Return Type:** Return a `Response` object.
- **Implementation:**
    1. **Remove from `studentStore`:** Attempt to remove the student with the given `id` from the `studentStore`. The `remove()` method returns the removed object (or null if it wasn't found).
    2. **If Removed (Not Null):** If the `remove()` method returned a non-null value (meaning the student was found and removed), return a 204 (No Content) response. This indicates success, but there's no response body.
    3. **If Not Found (Null):** If the `remove()` method returned `null` (meaning the student was not found), return a 404 (Not Found) response with an appropriate error message.

TESTING THE API

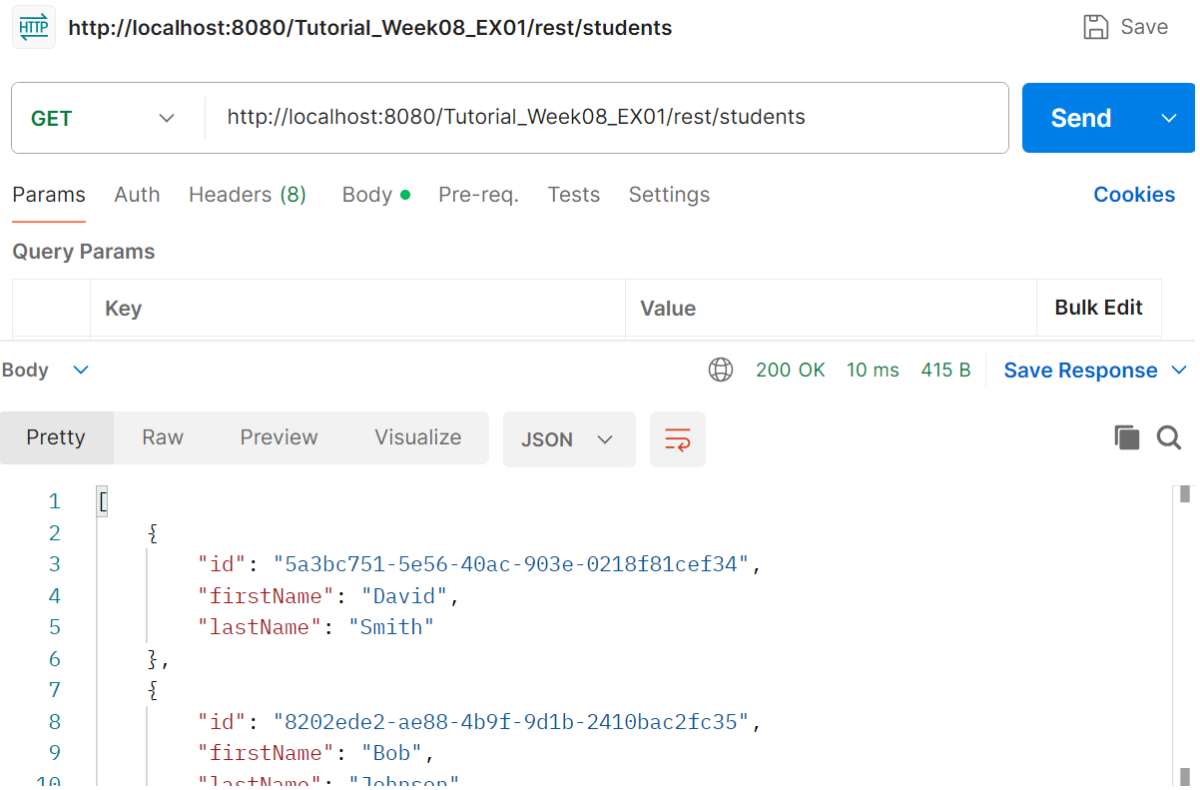
TEST THE API USING CURL COMMAND

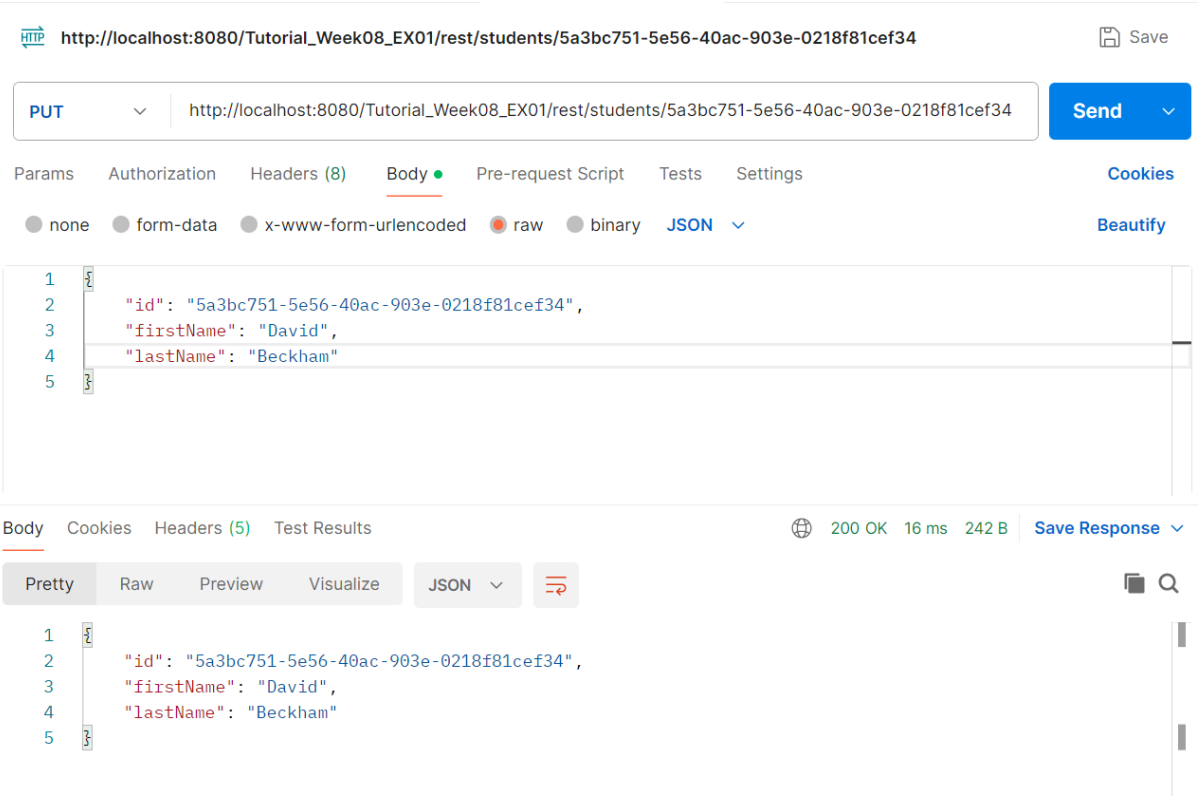
- You need to open cmd in Windows or terminal in Mac and use the following commands to send GET and POST requests.

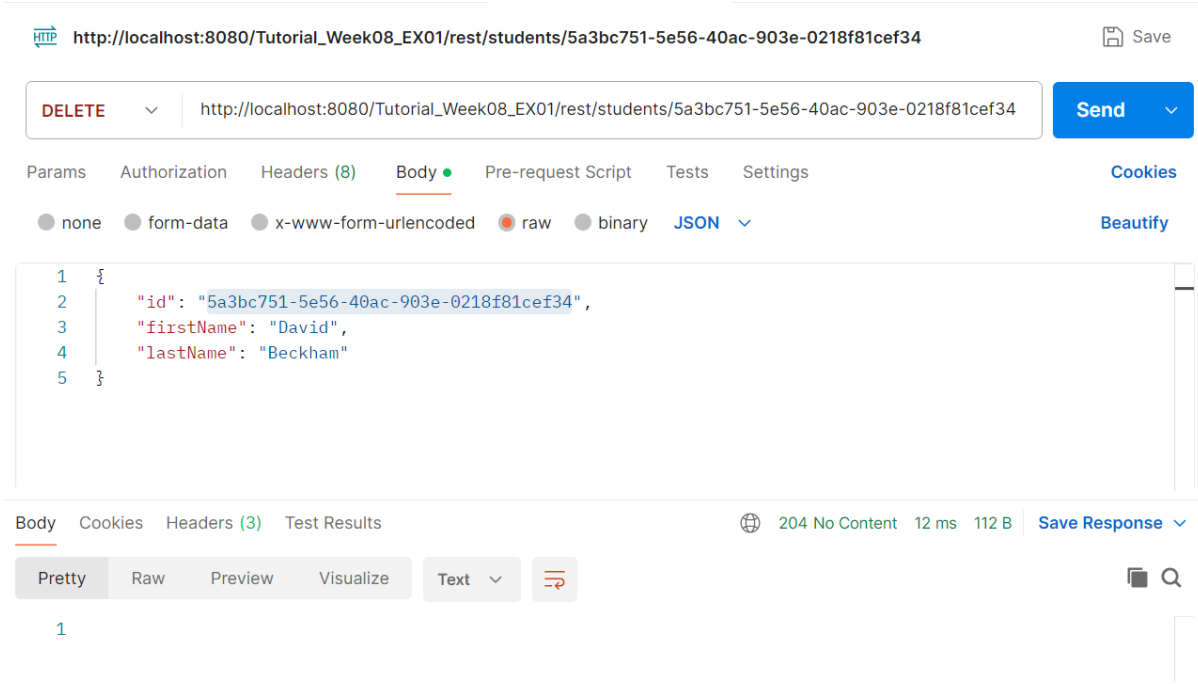
```
curl -X GET http://localhost:8080/ Tutorial_Week08_EX01/rest/students

curl -X POST -H "Content-Type: application/json" -d "{\"firstName\": \"John\", \"lastName\": \"Klich\"}" http://localhost:8080// Tutorial_Week08_EX01/rest/students

curl -X DELETE http://localhost:8080// Tutorial_Week08_EX01/rest/students/{userId}

curl -X PUT -H "Content-Type: application/json" -d "{\"firstName\":\"Joo\",\"lastName\":\"Smith\"}" http://localhost:8080// Tutorial_Week08_EX01/rest/students/<REPLACE_WITH_STUDENT_ID>
```

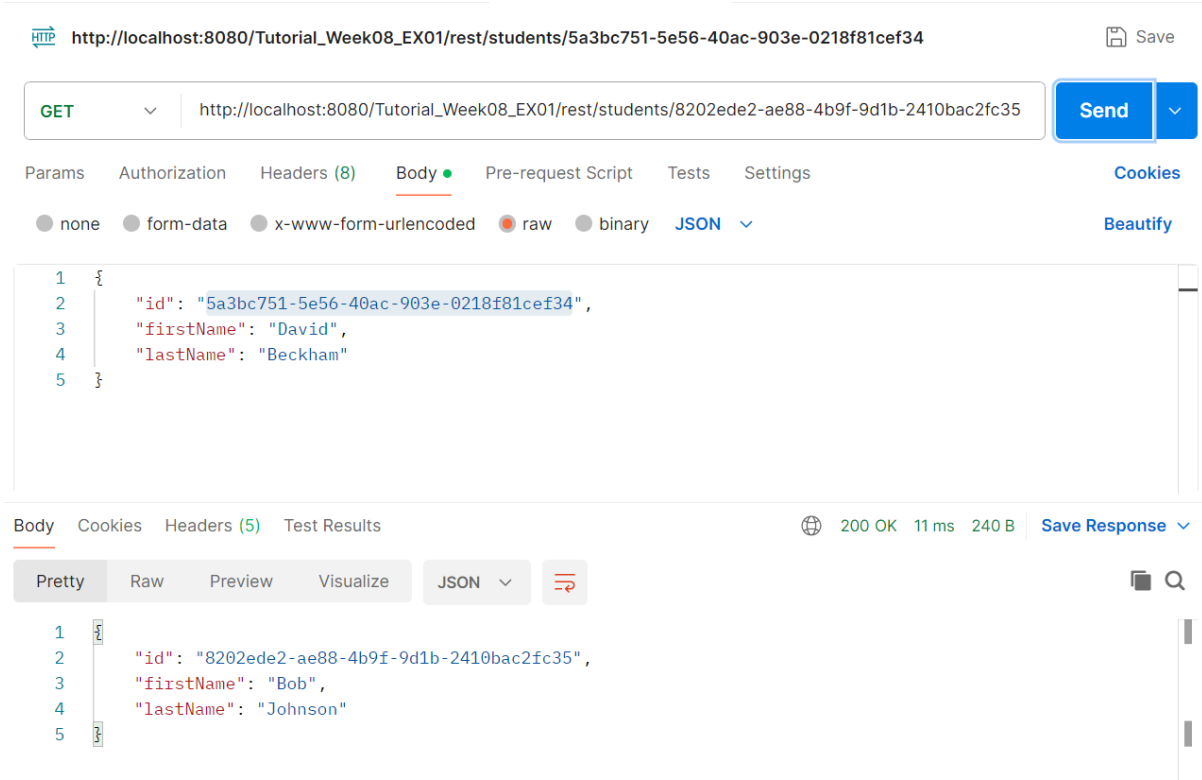
- Please note that in the POST request, you need to send a JSON payload to the server using firstName and lastname.

TEST THE API USING POSTMAN


http://localhost:8080/Tutorial_Week08_EX01/rest/students
Save
GET
http://localhost:8080/Tutorial_Week08_EX01/rest/students
Send
Params
Auth
Headers (8)
Body
Pre-req.
Tests
Settings
Cookies
Query Params
Key
Value
Bulk Edit
Body
200 OK
10 ms
415 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": "5a3bc751-5e56-40ac-903e-0218f81cef34",
"firstName": "David",
"lastName": "Smith"
},
{
"id": "8202ede2-ae88-4b9f-9d1b-2410bac2fc35",
"firstName": "Bob",


http://localhost:8080/Tutorial_Week08_EX01/rest/students/5a3bc751-5e56-40ac-903e-0218f81cef34
Save
PUT
http://localhost:8080/Tutorial_Week08_EX01/rest/students/5a3bc751-5e56-40ac-903e-0218f81cef34
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
{
"id": "5a3bc751-5e56-40ac-903e-0218f81cef34",
"firstName": "David",
"lastName": "Beckham"
}
Body
Cookies
Headers (5)
Test Results
200 OK
16 ms
242 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"id": "5a3bc751-5e56-40ac-903e-0218f81cef34",
"firstName": "David",
"lastName": "Beckham"
}


http://localhost:8080/Tutorial_Week08_EX01/rest/students/5a3bc751-5e56-40ac-903e-0218f81cef34
Save
DELETE
http://localhost:8080/Tutorial_Week08_EX01/rest/students/5a3bc751-5e56-40ac-903e-0218f81cef34
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
{
"id": "5a3bc751-5e56-40ac-903e-0218f81cef34",
"firstName": "David",
"lastName": "Beckham"
}
Body
Cookies
Headers (3)
Test Results
204 No Content
12 ms
112 B
Save Response
Pretty
Raw
Preview
Visualize
Text
1


http://localhost:8080/Tutorial_Week08_EX01/rest/students/5a3bc751-5e56-40ac-903e-0218f81cef34
Save
GET
http://localhost:8080/Tutorial_Week08_EX01/rest/students/8202ede2-ae88-4b9f-9d1b-2410bac2fc35
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
{
"id": "5a3bc751-5e56-40ac-903e-0218f81cef34",
"firstName": "David",
"lastName": "Beckham"
}
Body
Cookies
Headers (5)
Test Results
200 OK
11 ms
240 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"id": "8202ede2-ae88-4b9f-9d1b-2410bac2fc35",
"firstName": "Bob",
"lastName": "Johnson"
}

APPENDIX: **UNDERSTANDING JAVAX.WS.RS.CORE.RESPONSE IN JAX-RS**

In JAX-RS, the javax.ws.rs.core.Response class is a fundamental part of handling HTTP responses. Instead of directly manipulating low-level servlet responses, JAX-RS provides the Response class as an abstraction, offering a more convenient and flexible way to construct and send responses to clients.

Purpose of Response:

- **Encapsulates HTTP Response Details:** A Response object represents the complete HTTP response, including:
  - **Status Code:** The standard HTTP status code (e.g., 200 OK, 201 Created, 404 Not Found, 500 Internal Server Error).
  - **Headers:** HTTP headers (e.g., Content-Type, Location).
  - **Entity Body:** The actual content of the response (e.g., a JSON object, an HTML page, an image).
- **Provides a Builder Pattern:** Response uses a builder pattern, allowing you to chain method calls to construct the response step-by-step. This makes the code more readable and easier to maintain.
- **Abstraction from Servlet API:** Using Response shields you from the complexities of the underlying servlet API, making your JAX-RS code more portable and easier to test.

Key Methods and Usage:

1. **Response.status(int status) / Response.status(Response.Status status):**
   - Sets the HTTP status code of the response.
   - You can use either an integer (e.g., 200) or an enum constant from Response.Status (e.g., Response.Status.OK). Using the Response.Status enum is generally preferred for readability and type safety.
   - This method *returns a Response.ResponseBuilder*, allowing you to chain further method calls.

```
// Using integer status code
Response.status(200);

// Using Response.Status enum (recommended)
Response.status(Response.Status.OK);
```

2. **Response.ok() / Response.ok(Object entity) / Response.ok(Object entity, MediaType mediaType):**
   - Creates a ResponseBuilder with a status code of 200 (OK).
   - The no-argument version (Response.ok()) creates an OK response with no entity body.
   - The version with an entity parameter sets the response body.
   - The version with entity and mediaType sets both the body and the Content-Type header.

```
// OK response with no body
Response.ok().build();

// OK response with a String entity
Response.ok("Success").build();

// OK response with a Student object and JSON content type
Student student = ...;
Response.ok(student, MediaType.APPLICATION_JSON).build();
```

3. **Response.created(URI location):**

- Creates a ResponseBuilder with a status code of 201 (Created).
- Sets the Location header to the provided URI (typically the URI of the newly created resource).

```
URI location =
UriBuilder.fromPath("/students/{id}").build(newStudent.getId());
Response.created(location).entity(newStudent).build();
```

4. **Response.noContent():**
   - Creates a ResponseBuilder with a status code of 204 (No Content). This is often used for successful DELETE operations or PUT operations where no response body is needed.

```
Response.noContent().build();
```

5. **Response.notModified():**

- Creates a ResponseBuilder with status 304 (Not Modified)

```
Response.notModified().build();
```

6. **Response.seeOther(URI location):**

- Creates a ResponseBuilder with status 303 (See Other)

```
URI location =
UriBuilder.fromPath("/students/{id}").build(newStudent.getId());
Response.seeOther(location).build();
```

7. **Response.temporaryRedirect(URI location):**
   - Creates a ResponseBuilder with status code 307 (Temporary Redirect).
   - Sets the Location header for the redirection.

```
URI redirectLocation = ...;
Response.temporaryRedirect(redirectLocation).build();
```

8. **Response.status(Response.Status.NOT_FOUND) / Response.status(Response.Status.BAD_REQUEST) / etc.:**
   - Use Response.Status enum constants for other common status codes.

```
Response.status(Response.Status.NOT_FOUND).entity("Resource not found").build();
Response.status(Response.Status.BAD_REQUEST).entity("Invalid input").build();
```

9. **Response.entity(Object entity):**
   - Sets the entity body of the response. This is the data that will be sent to the client.
   - Returns a ResponseBuilder.

```
Response.status(Response.Status.OK).entity("Hello, world!").build();
```

10. **Response.header(String name, Object value):**
    - Adds a custom HTTP header to the response.
    - Returns a ResponseBuilder.

```
 Response.status(Response.Status.OK)
         .entity(student)
         .header("X-Custom-Header", "MyValue")
         .build();
```

11. **Response.build():**
    - **Crucially**, this method *builds* the final Response object. You *must* call build() at the end of the chain of method calls. Without build(), you won't have a complete Response object.

```
 Response response = Response.status(Response.Status.OK)
         .entity(student)
         .header("X-Custom-Header", "MyValue")
```

```
        .build(); // build() is essential!
```

Example (Combining Multiple Methods):

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getStudentById(@PathParam("id") String id) {
    Student student = studentStore.get(id);
    if (student != null) {
        return Response.ok(student) // Status 200, entity is the
student object
                .header("X-Student-Id", student.getId()) // Add a
custom header
                .build(); // Build the final Response object
    } else {
        return Response.status(Response.Status.NOT_FOUND) // Status
404
                .entity("Student with ID " + id + " not found") //
Set the error message
                .build(); // Build the final Response object
    }
}
```