LS/EECS

Building E-commerce Applications

REST with Jakarta Jersey 3.x

Objectives

By the end of this lab, students should be able to:

- Understand the basic principles of RESTful web services and CRUD operations.
- Create a new Maven project and add dependencies for Jersey 3.x.
- Develop a Model class to represent student information.
- Create a DAO class to handle CRUD operations in memory.
- Set up a Controller to expose CRUD REST APIs.
- Test the APIs using tools like Postman or Curl commands.

Introduction

In this lab, we will focus on creating a Student Information System that exposes CRUD (Create, Read, List, Update, Delete) REST APIs. We will use the Jersey 2.x library to build these APIs. Unlike the previous lab, we won't implement the views; instead, we will focus on the Model and Controller components of the MVC pattern.

Understanding REST API Concepts

Before diving into the lab tasks, it's crucial to understand the concept of REST (Representational State Transfer) APIs (Application Programming Interfaces).

What is REST?

REST is an architectural style for designing networked applications. It uses a stateless, client-server communication model, meaning that each request from a client to a server must contain all the information needed to understand and process the request.

CRUD Operations in REST

REST APIs expose endpoints for CRUD operations:

- **Create**: Adds new data. Typically uses HTTP POST.
- **Read**: Retrieves data. Typically uses HTTP GET.
- **Update**: Modifies existing data. Typically uses HTTP PUT.
- **Delete**: Removes data. Typically uses HTTP DELETE.

HTTP Methods

RESTful services use standard HTTP methods:

- **GET**: Retrieve data
- **POST**: Create data
- **PUT**: Update data
- **DELETE**: Delete data

Status Codes

HTTP status codes indicate the outcome of the server's attempt to process the request. For example:

- **200 OK**: Success
- **201 Created**: Successfully created a new resource
- **400 Bad Request**: The server couldn't understand the request
- **404 Not Found**: Resource not found
- **500 Internal Server Error**: An error on the server side

Understanding JSON Format

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript language.

Basic JSON Syntax

A JSON object contains key-value pairs and looks like this:

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": {
    "subKey1": "subValue1"
  },
  "key4": ["arrayItem1", "arrayItem2"]
}
```

- **Keys** are always strings.
- **Values** can be strings, numbers, objects, arrays, true, false, or null.

JSON in REST APIs

In REST APIs, JSON is commonly used for both request and response bodies. For example, to create a new student, you might send a POST request with the following JSON payload:

```
{
  "name": "John",
  "age": 20,
  "major": "Computer Science"
}
```

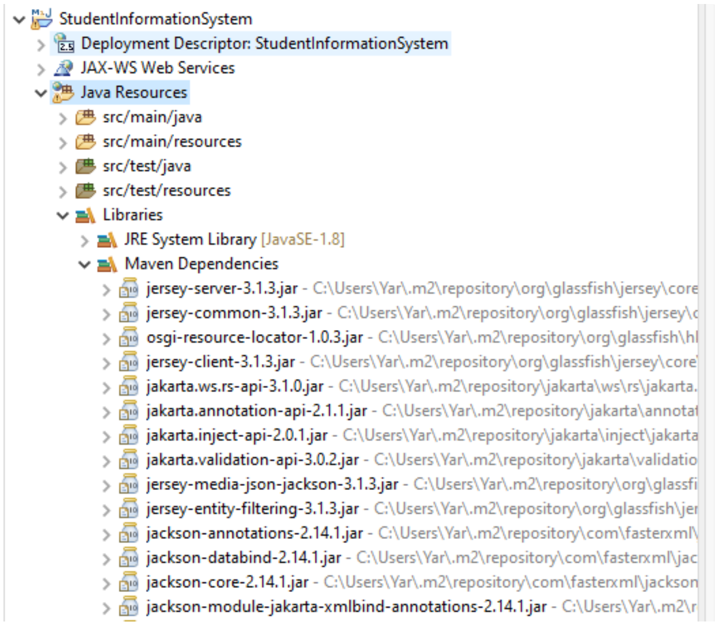
Task 1: Create a New Maven Project

1. Launch the Eclipse IDE on your computer.
2. If prompted, select a workspace where you want your new project to reside.
3. Go to File -> New -> Maven Project.
4. If a "Select a wizard" dialog appears, choose General -> Project.
5. In the "New Maven Project" dialog, check the "Create a simple project" box if you want to skip archetype selection. Otherwise, you can choose an archetype.
6. Click Next.
7. Enter Project Details:
    a. Group Id: Enter a group ID, usually in reverse domain name format (e.g., `org.yorku`).
    b. Artifact Id: Enter an artifact ID, which will be your project name (e.g., `StudentInformationSystem`).
    c. Version: Leave it as 0.0.1-SNAPSHOT or enter a version number.
    d. Packaging: Choose jar or war based on your project needs. For web projects, war is commonly used.
8. Click Finish.
9. Add Jersey Dependencies:
    a. Open the pom.xml file located in your project.
    b. Add the necessary dependencies for Jersey 3.x and any other libraries you'll use. Here's a sample snippet for adding Jersey dependencies to your pom.xml:

```
<dependencies>
    <!-- Jersey 3.x dependencies -->
    <dependency>
        <groupId>org.glassfish.jersey.core</groupId>
        <artifactId>jersey-server</artifactId>
        <version>3.1.3</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jackson</artifactId>
        <version>3.1.3</version>
    </dependency>
      <!-- ... other dependencies ... -->
      <!-- ... Jersey Dependencies for injecting Java files into
Jersey Servlet ... -->

<dependency>
             <groupId>org.glassfish.jersey.containers</groupId>
             <artifactId>jersey-container-servlet</artifactId>
             <version>3.1.3</version>
      </dependency>
      <dependency>
             <groupId>org.glassfish.jersey.inject</groupId>
             <artifactId>jersey-hk2</artifactId>
             <version>3.1.3</version>
      </dependency>
</dependencies>
```

10. Save and Build: Save the pom.xml file. Maven should automatically download the dependencies. If not, right-click on the project and choose **Maven -> Update Project.**
11. You can now see the dependencies added to your project:



```
StudentInformationSystem
  Deployment Descriptor: StudentInformationSystem
  JAX-WS Web Services
  Java Resources
    src/main/java
    src/main/resources
    src/test/java
    src/test/resources
    Libraries
      JRE System Library [JavaSE-1.8]
      Maven Dependencies
        jersey-server-3.1.3.jar - C:\Users\Yar\.m2\repository\org\glassfish\jersey\core
        jersey-common-3.1.3.jar - C:\Users\Yar\.m2\repository\org\glassfish\jersey\c
        osgi-resource-locator-1.0.3.jar - C:\Users\Yar\.m2\repository\org\glassfish\hl
        jersey-client-3.1.3.jar - C:\Users\Yar\.m2\repository\org\glassfish\jersey\core
        jakarta.ws.rs-api-3.1.0.jar - C:\Users\Yar\.m2\repository\jakarta\ws\rs\jakarta.
        jakarta.annotation-api-2.1.1.jar - C:\Users\Yar\.m2\repository\jakarta\annotat
        jakarta.inject-api-2.0.1.jar - C:\Users\Yar\.m2\repository\jakarta\inject\jakarta
        jakarta.validation-api-3.0.2.jar - C:\Users\Yar\.m2\repository\jakarta\validatio
        jersey-media-json-jackson-3.1.3.jar - C:\Users\Yar\.m2\repository\org\glassfi
        jersey-entity-filtering-3.1.3.jar - C:\Users\Yar\.m2\repository\org\glassfish\jer
        jackson-annotations-2.14.1.jar - C:\Users\Yar\.m2\repository\com\fasterxml\
        jackson-databind-2.14.1.jar - C:\Users\Yar\.m2\repository\com\fasterxml\jac
        jackson-core-2.14.1.jar - C:\Users\Yar\.m2\repository\com\fasterxml\jackson
        jackson-module-jakarta-xmlbind-annotations-2.14.1.jar - C:\Users\Yar\.m2\r
```

12. Create a web.xml for your Maven Project in webapp -> WEB-INF

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
<!-- Jersey Servlet configurations -->
      <servlet>
            <servlet-name>Jersey REST Service</servlet-name>
            <servlet-
            class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
            <init-param>
                  <param-name>jersey.config.server.provider.packages</param-name>
                  <param-value>com.<your-name>.studentapp</param-value>
            </init-param>
            <load-on-startup>1</load-on-startup>
            </servlet>
            <servlet-mapping>
            <servlet-name>Jersey REST Service</servlet-name>
            <url-pattern>/*</url-pattern>
      </servlet-mapping>
</web-app>
```

This xml file will use the Jersey Servlet when running your Java files. The <init-param> will point to your java package that you will create in the next step where all your java files are located. The <url-pattern> will be the url where the client will interact with your REST service.

Quick Maven Project Run-through

Right-click your Maven Project (*StudentInformationSystem*) -> go to Run-as. You will see multiple actions you can do with your project.

Maven clean - used when you want to remove files generated at build-time in a project's directory. *(If you used Maven Install or Maven build, you will notice your target folder deleted or empty)*

Maven install – part of the maven lifecycle, compiles the source code and packages the project. It is the equivalent of running *mvn install* on the command line if you have maven installed. Once you run this through eclipse, you will see the target folder containing your compiled application and WAR files.

Maven build – Not part of the maven lifecycle, you can use Maven build to run custom run configurations. When you select this, try adding *package* to the Goals field to get Maven build to work. Just like Maven install, this will create a WAR file in the target folder.

However, to just deploy your project to your Tomcat server, you can **Run-as -> Run on Server** as done in the previous labs.

Task 2: Develop the Model Class

Create a new Java class named Student.java in a package `com.<your-name>.studentapp;`. <your-name> should be a shortened version of your full name without any spaces, ex. jsmith

1. Add fields for student ID, name, age, and major.
2. Generate getters and setters for these fields.

Here's a sample code snippet for the Model class:

```
package com.<your-name>.studentapp; //This line should be on the top
public class Student {
    private int id;
    private String name;
    private int age;
    private String major;

    // Getters
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getMajor() {
        return major;
    }

    // Setters
    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setMajor(String major) {
        this.major = major;
    }
}
```

Task 3: Create the DAO Class

1. Create a new Java class named `StudentDAO.java`.
2. Add a static list variable that will act as an in-memory database for storing student objects.
3. Implement methods for CRUD operations: `create`, `read`, `readAll`, `update`, and `delete`.

Here's a sample code snippet for the DAO class:

```
package com.<your-name>.studentapp;
import java.util.ArrayList;
import java.util.List;

public class StudentDAO {
    private static List<Student> students = new ArrayList<>();

    public Student create(Student student) {
        students.add(student);
        return student;
    }

    public List<Student> readAll() {
        return students;
    }

    public Student read(int id) {
        return students.stream().filter(s -> s.getId() ==
id).findFirst().orElse(null);
    }

    public Student update(int id, Student student) {
        // Implement update logic
        // ...
        return updatedStudent;
    }

    public void delete(int id) {
        students.removeIf(s -> s.getId() == id);
    }
}
```

Task 4: Set Up the Controller

1. Create a new Java class named StudentController.java.
2. Use Jersey annotations to define REST endpoints for CRUD operations.
3. Call corresponding DAO methods in each REST API endpoint.

Here's a sample code snippet for the Controller class:

```
package com.<your-name>.studentapp;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import java.util.List;

@Path("/students")
public class StudentController {
    private StudentDAO studentDAO = new StudentDAO();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Student> getAllStudents() {
        return studentDAO.readAll();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Student createStudent(Student student) {
        return studentDAO.create(student);
    }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Student getStudent(@PathParam("id") int id) {
        return studentDAO.read(id);
    }

    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Student updateStudent(@PathParam("id") int id, Student student) {
        return studentDAO.update(id, student);
    }

    @DELETE
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public void deleteStudent(@PathParam("id") int id) {
        studentDAO.delete(id);
    }
}
```

Testing with Curl

Run your application and use Postman or Curl commands to test the CRUD operations.

Here are some example Curl commands to test the different scenarios:

Create a Student

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"John", "age":20,
"major":"Computer Science"}' http://localhost:8080/YourApp/students
```

> Windows CMD can’t read single quotes, so you may have to use \” to replace the single and quotes in your data payload

```
curl -X POST -H "Content-Type: application/json" -d
"{\"name\":\"John\", \"age\":20, \"major\":\"Computer Science\"}"
http://localhost:8080/YourApp/students
```

List All Students

```
curl -X GET http://localhost:8080/YourApp/students
```

Get a Specific Student

```
curl -X GET http://localhost:8080/YourApp/students/1
```

Update a Student

```
curl -X PUT -H "Content-Type: application/json" -d '{"name":"John Doe", "age":21,
"major":"Software Engineering"}' http://localhost:8080/YourApp/students/1
```

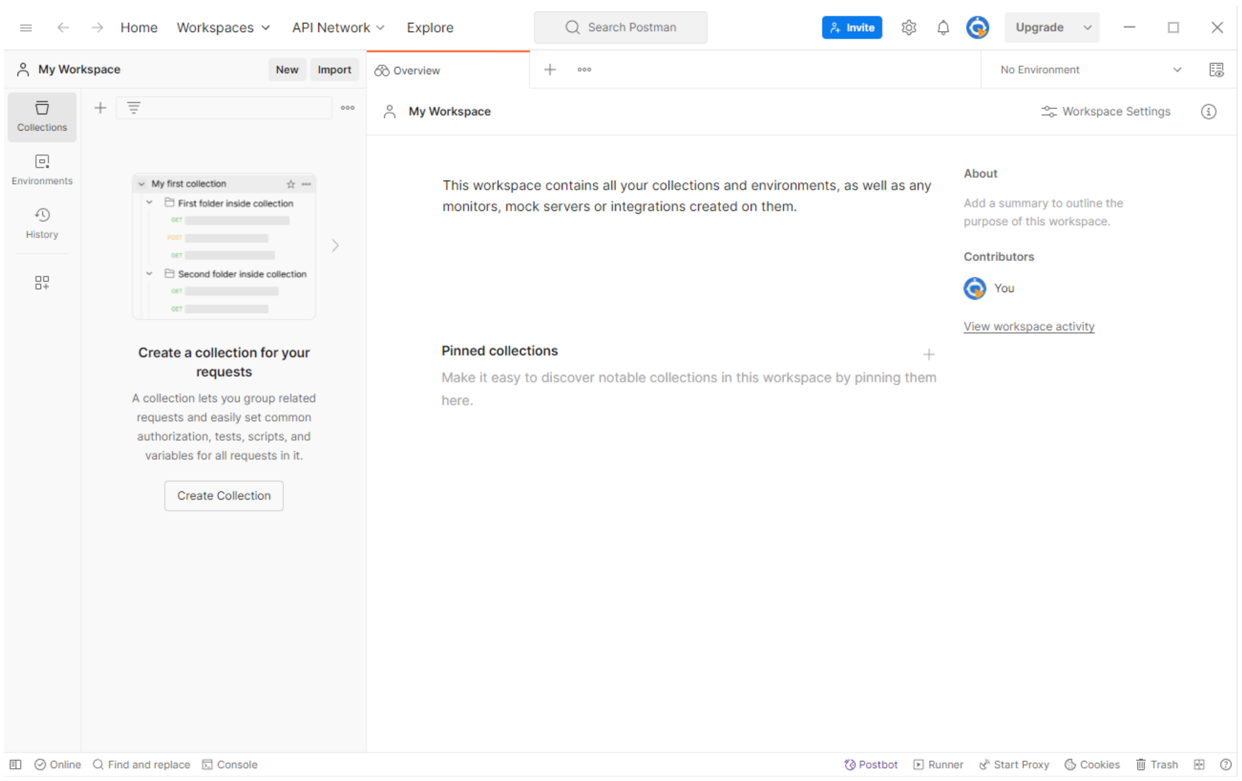
Delete a Student

```
curl -X DELETE http://localhost:8080/YourApp/students/1
```

Testing with Postman

We can also use Postman, a widely used API platform, to help us test our REST API endpoints of our application.

- Download on Windows, MAC or Linux: https://www.postman.com/downloads/
- Go to your Workspaces -> My Workspace *(the default workspace, you can also create a new workspace for other projects*)



This workspace contains all your collections and environments, as well as any monitors, mock servers or integrations created on them.

Add a summary to outline the purpose of this workspace.

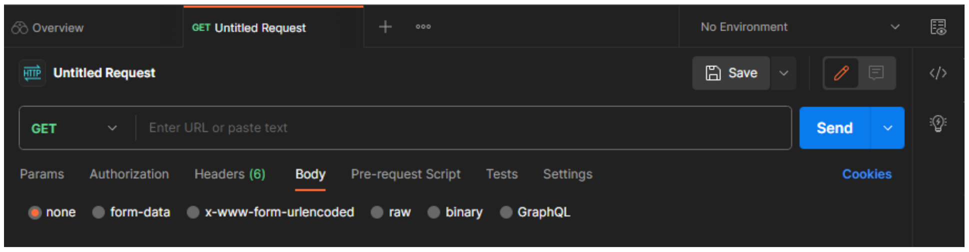
Create a collection for your requests

A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.

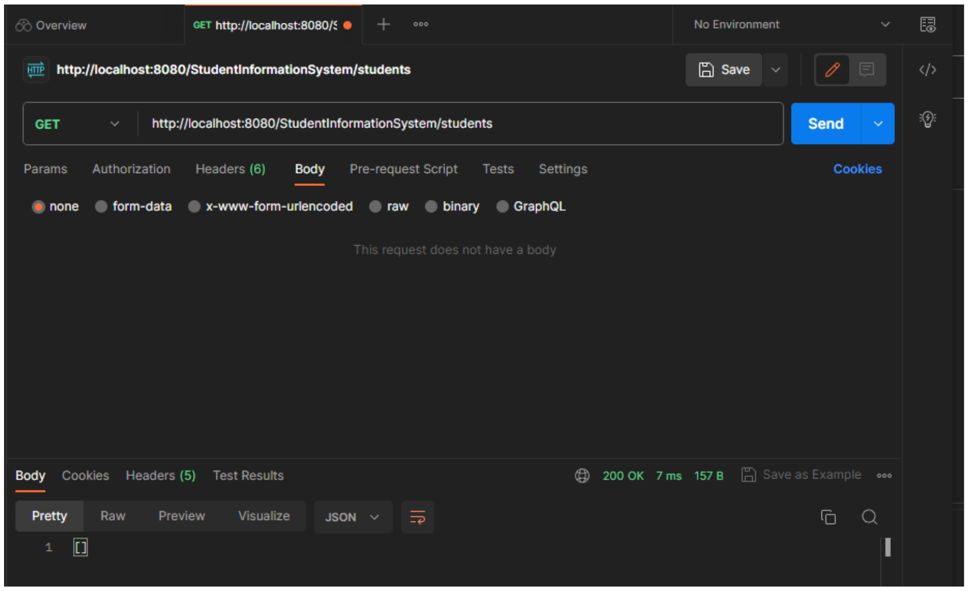
Make it easy to discover notable collections in this workspace by pinning them here.

(To change to Dark Mode, Go to Settings - > Themes)

- Click the **+** beside *Overview* in the top center of the screen to create a **new Request tab** in Postman

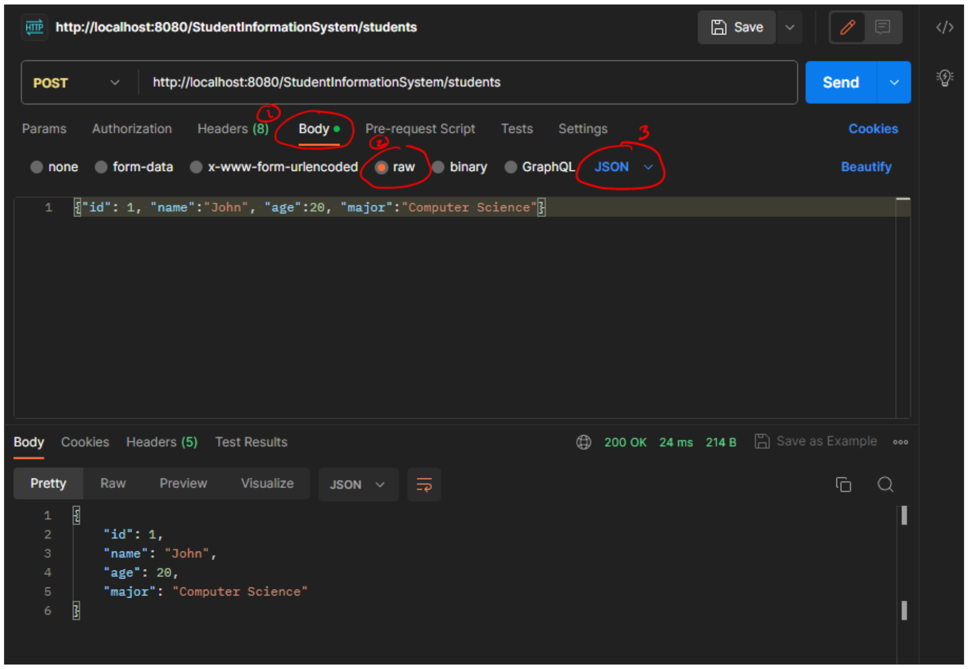


- You can change the HTTP methods (GET, POST, PUT, DELETE) in the dropdown box in your new Request Tab. Add the URL of your Maven application and try the **Get** Request:


Overview
GET http://localhost:8080/S
No Environment
http://localhost:8080/StudentInformationSystem/students
Save
GET
http://localhost:8080/StudentInformationSystem/students
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies
none form-data x-www-form-urlencoded raw binary GraphQL
This request does not have a body
Body Cookies Headers (5) Test Results
200 OK 7 ms 157 B Save as Example
Pretty Raw Preview Visualize JSON

```
[]
```

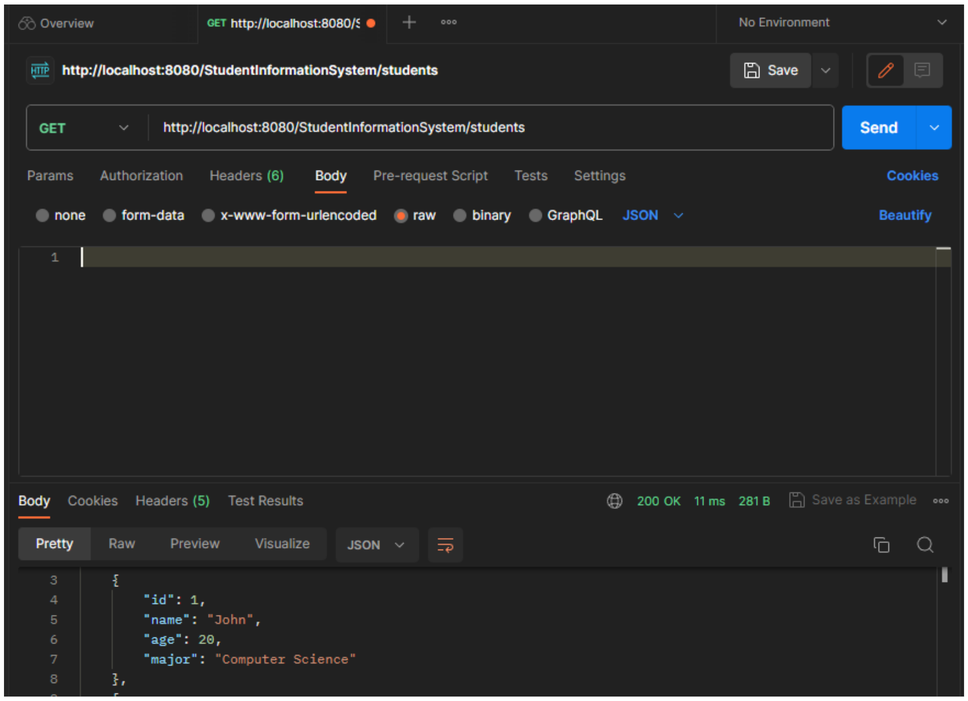
- To do the POST Request, go to the dropdown box and change it to POST
- Then under the URL Field, click **Body -> Raw.** Since you are sending a JSON to the web application, change text to JSON *(You can find the dropdown right beside the GraphQL checkbox).* There should now be a textbox now where you can send the JSON to your web application.


http://localhost:8080/StudentInformationSystem/students
Save
POST
http://localhost:8080/StudentInformationSystem/students
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies
1
2
3
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (5) Test Results
200 OK 24 ms 214 B Save as Example
Pretty Raw Preview Visualize JSON

```
{"id": 1, "name":"John", "age":20, "major":"Computer Science"}
```

```
{
    "id": 1,
    "name": "John",
    "age": 20,
    "major": "Computer Science"
}
```

- The JSON payload has been sent to your web application. Now when you send the GET request through Postman, you should be able to see the Student you just created:



http://localhost:8080/StudentInformationSystem/students

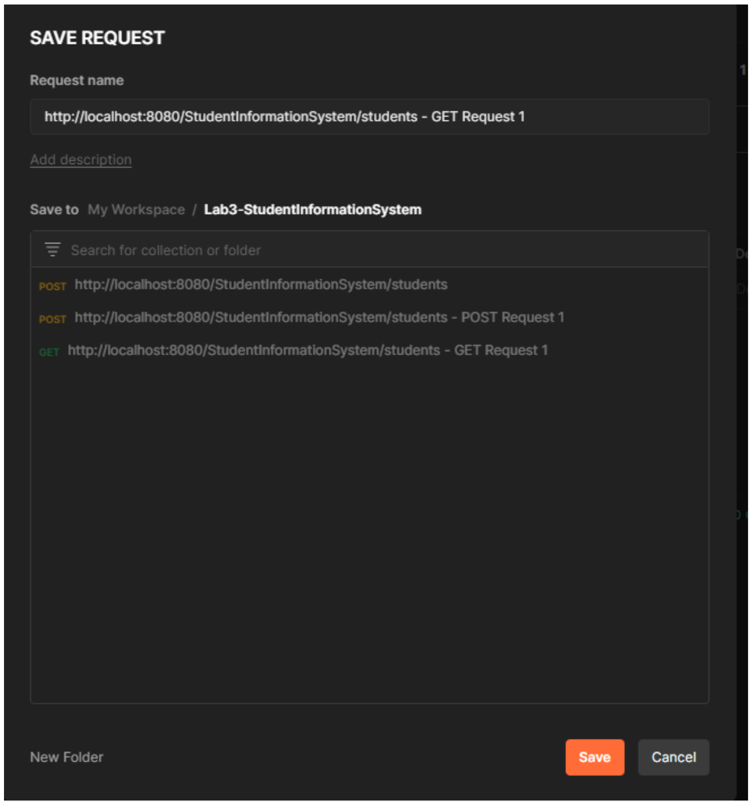
GET http://localhost:8080/StudentInformationSystem/students

Body: raw, JSON

200 OK 11 ms 281 B

```
    {
        "id": 1,
        "name": "John",
        "age": 20,
        "major": "Computer Science"
    },
```

- You can also save all your API Requests. Go to Save **or Dropdown - > Save As …** (If you are saving a new API call, and don't want to overwrite the request you already saved), create a new Collection (example **collection name**: Lab3-StudentInformationSystem), and save your requests.



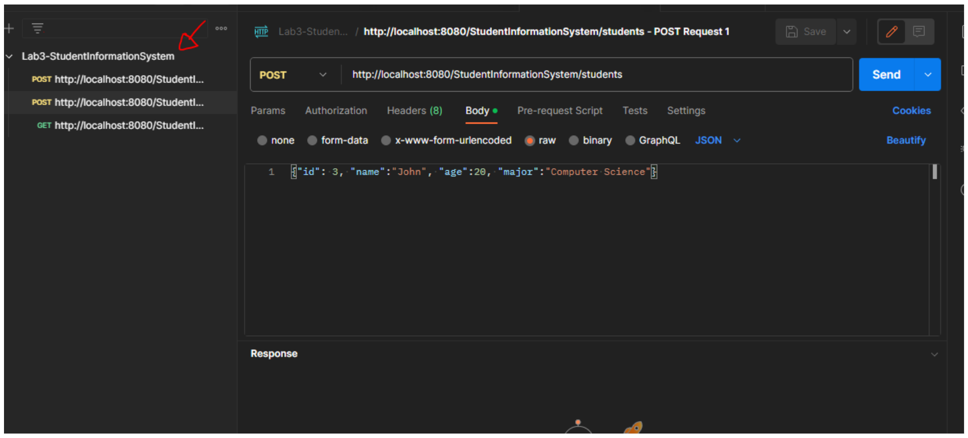
SAVE REQUEST

Request name: http://localhost:8080/StudentInformationSystem/students - GET Request 1

Save to My Workspace / Lab3-StudentInformationSystem

POST http://localhost:8080/StudentInformationSystem/students
POST http://localhost:8080/StudentInformationSystem/students - POST Request 1
GET http://localhost:8080/StudentInformationSystem/students - GET Request 1

- You can now easily scroll go through your API requests on the left box of Postman:



Lab3-StudentInformationSystem

POST http://localhost:8080/StudentI...

POST http://localhost:8080/StudentI...

GET http://localhost:8080/StudentI...

Lab3-Studen... / http://localhost:8080/StudentInformationSystem/students - POST Request 1

POST http://localhost:8080/StudentInformationSystem/students

Body: raw JSON

```
{"id": 3, "name":"John", "age":20, "major":"Computer Science"}
```

- You can also export your collection of requests to share with another developer or to import it to another workspace. Right-click your Collection (*On the above image, the red arrow is pointing at your collection*) -> Export and save it on your computer.

Postman is useful to test if parts of your REST APIs for your application are working as intended. Use it as you develop your web application!

Feel free to reach out if you have any questions or run into issues. Good luck!