## 5COSC022W.2 Client-Server Architectures

# Tutorial Week 09: 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 CRUD operations using DAO classes and the related methods inside the resource classes. Please note that some parts of the DAO and Resource classes are highlighted in red, which means that these are independent exercises.

## 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 student management system. In this application, we will create DAO classes for each entity in order to separate business logic and data access layer. While HTTP methods will be implemented in the resource class, all CRUD operations will be handled through DAO classes.

### STEP 1: CREATE PROJECT

- Create a new Maven Web Application project in NetBeans and name it as **StudentResourceDAO.**
- Select the **Apache Tomcat** as a server and also select **Java EE 8** from dropdown list.
- Once the project is created, please remove unnecessary packages and classes under Source Packages.
- Do right click on the Source Packages and click on New and then select Java Packages. Name the package as com.example.model.
- Please create three model classes under this package called Module, Student and Teacher.
- Similarly, create other package and name it as com.example.resource. Under this package, create a three resource classes called StudentResource, TeacherResource and ModuleResource.
- Create one package more and name it as com.example.dao. Create three classes under this package and name them as ModuleDAO, StudentDAO and TeacherDAO

### STEP 2: COM.EXAMPLE.MODEL PACKAGE

#### STUDENT CLASS

- **Class Definition:**
  - Create a `public` class named `Student`.
- **Fields:**

- Declare a `private int` field named `id`.
- Declare a `private String` field named `name`.

- **Default Constructor:**

  - Create a `public` constructor with no parameters (`Student()`). Leave the body of this constructor empty (`{}`).

- **Parameterized Constructor:**

  - Create a `public` constructor that takes an `int id` and a `String name` as parameters (`Student(int id, String name)`).
  - Inside the constructor:
    - Assign the value of the `id` parameter to the `id` field of the object using `this.id = id;`.
    - Assign the value of the `name` parameter to the `name` field of the object using `this.name = name;`.

- **Getters and Setters`id`:**

  - Create getters and setters for all attributes

---

### TEACHER CLASS

- **Class Definition:**

  - Create a `public` class named `Teacher`.

- **Fields:**

  - Declare a `private int` field named `id`.
  - Declare a `private String` field named `name`.

- **Default Constructor:**

  - Create a `public` constructor with no parameters (`Teacher()`). Leave the body of this constructor empty (`{}`).

- **Parameterized Constructor:**

  - Create a `public` constructor that takes an `int id` and a `String name` as parameters (`Teacher(int id, String name)`).
  - Inside the constructor:
    - Assign the value of the `id` parameter to the `id` field using `this.id = id;`.
    - Assign the value of the `name` parameter to the `name` field using `this.name = name;`.

- **Getters and Setters:**

  - Create getters and setters for all attributes

### MODULE CLASS

- **Class Definition:**
  - Create a `public` class named `Module`.
- **Fields:**
  - Declare a `private int` field named `id`.
  - Declare a `private String` field named `name`.
  - Declare a `private Teacher` field named `teacher`.
- **Default Constructor:**
  - Create a `public` constructor with no parameters (`Module()`). Leave the body of this constructor empty (`{}`).
- **Parameterized Constructor:**
  - Create a `public` constructor that takes an `int id`, a `String name`, and a `Teacher teacher` as parameters (`Module(int id, String name, Teacher teacher)`).
  - Inside the constructor:
    - Assign the value of the `id` parameter to the `id` field using `this.id = id;`.
    - Assign the value of the `name` parameter to the `name` field using `this.name = name;`.
    - Assign the value of the `teacher` parameter to the `teacher` field using `this.teacher = teacher;`.
- **Getters and Setters:**
  - Create getters and setters for all attributes

## STEP 3: COM.EXAMPLE.DAO PACKAGE

### STUDENT DAO CLASS

- **Class Definition:**
  - Create a `public` class named `StudentDAO`.
- **Imports:**
  - Import the following classes:
    - `com.example.model.Student` (This assumes you have a `Student` class in the `com.example.model` package. You'll need to create this class separately if it doesn't exist. The Student class needs an `int id` and a `String name` field, plus getters and setters for these.)

    - `java.util.ArrayList`
    - `java.util.Collections` (Although not directly used in the provided code, it's a common utility class when working with collections, so it's good to include it.)
    - `java.util.List`
    - `java.util.Map`
    - `java.util.HashMap`

- **`students` Field:**
    - Declare a `private static List<Student>` named `students`. Initialize it with a new `ArrayList<>()`. This list will store `Student` objects. The `static` keyword means this list is shared by all instances of the `StudentDAO` class (it's a class-level variable, not an instance-level variable).

- **Static Initialization Block:**
    - Create a `static` block (using the `static { ... }` syntax). This block of code will be executed *once* when the class is loaded.
    - Inside the static block:
        - Add at least two `Student` objects to the `students` list using `students.add()`.
        - For each `Student`, create a new `Student` object using the constructor (e.g., `new Student(1, "John Doe")`). Use different IDs and names for each student.
        - You can add more `Student` objects here if you want to initialize the list with more data.

- **`getAllStudents` Method:**
    - Create a `public` method named `getAllStudents` that returns a `List<Student>`.
    - Inside the method, simply `return` the `students` list.

- **`getStudentById` Method:**
    - Create a `public` method named `getStudentById` that takes an `int id` as a parameter and returns a `Student` object (or `null` if no student with that ID is found).
    - **Iterate:** Use a `for-each` loop to iterate through the `students` list.
    - **Check ID:** Inside the loop, check if the current `Student` object's ID (obtained using `student.getId()`) is equal to the `id` parameter.
    - **Return Student:** If the IDs match, `return` the current `Student` object.
    - **Return Null:** If the loop completes without finding a matching student, `return null`.

- **`addStudent` Method:**
    - Create a `public` method named `addStudent` that takes a `Student` object as a parameter and returns `void`.
    - **Get the next user id**
        - invoke `getNextUserId()` method and store the result in an integer called `newUserId`.
    - **Set ID:** Call the `setId` method of the input `student` object, passing in the `newUserId`.
    - **Add to List:** Add the `student` object to the `students` list using `students.add()`.

- **`updateStudent` Method:**

- Create a `public` method named `updateStudent` that takes a `Student` object (representing the updated student information) as a parameter and has a `void` return type.
- **Iterate:** Use a `for` loop with an index (e.g., `for (int i = 0; i < students.size(); i++)`) to iterate through the `students` list. You need the index to be able to modify the list.
- **Get Student:** Inside the loop, get the current `Student` object using `students.get(i)`.
- **Check ID:** Compare the ID of the current `Student` object with the ID of the `updatedStudent` object.
- **Update:** If the IDs match:
    - Replace the `Student` object at the current index `i` in the `students` list with the `updatedStudent` object using `students.set(i, updatedStudent)`.
    - Print a message "Student Updated: " plus the `updatedStudent` object.
    - `return` from the method (since you've found and updated the student).

- **`deleteStudent` Method:**

    - Create a `public` method named `deleteStudent` that takes an `int id` as a parameter and has a `void` return type.
    - **Remove:** Use the `removeIf` method of the `students` list to remove the student with the matching ID. Use a lambda expression: `students.removeIf(student -> student.getId() == id);`

- **`getNextUserId` Method:**

    - Create a public method that does not accept parameters and return an integer.
    - Create a variable named `maxUserId` and assign `Integer.MIN_VALUE`.
    - Iterate through all elements of the students list.
        - get the id of the iterated user, and store it inside a variable called `userId`.
        - create an if statement, check if the `userId` is greater than `maxUserId`, then assign `userId` value to `maxUserId`.
    - return `maxUserId + 1`.

## TEACHER DAO CLASS

- **Class Definition:**

    - Create a `public` class named `TeacherDAO`.

- **Imports:**

    - Import the following classes:
        - `com.example.model.Teacher` (This assumes you have a `Teacher` class in the `com.example.model` package. This class needs an `int id` and a `String name` field, along with corresponding getter and setter methods.)
        - `java.util.ArrayList`
        - `java.util.List`

- **`teachers` Field:**
    - Declare a `private static List<Teacher>` named `teachers`. Initialize it with a new `ArrayList<>()`. This list will hold `Teacher` objects. The `static` keyword makes it a class-level variable.
- **Static Initialization Block:**
    - Create a `static` block (`static { ... }`). This code will run once when the class is loaded.
    - Inside the static block:
        - Add at least two `Teacher` objects to the `teachers` list using `teachers.add()`.
        - For each `Teacher`, create a new `Teacher` object (e.g., `new Teacher(1, "Mr. Smith")`). Use different IDs and names.
        - Add more `Teacher` objects as desired.
- **`getAllTeachers` Method:**
    - Create a `public` method named `getAllTeachers` that returns a `List<Teacher>`.
    - Inside the method, `return` the `teachers` list.
- **`getTeacherById` Method:**
    - Create a `public` method named `getTeacherById` that takes an `int id` as input and returns a `Teacher` object (or `null` if no teacher with that ID is found).
    - **Iterate:** Use a `for-each` loop to iterate through the `teachers` list.
    - **Check ID:** Inside the loop, check if the current `Teacher` object's ID (using `teacher.getId()`) is equal to the input `id`.
    - **Return Teacher:** If the IDs match, `return` the current `Teacher` object.
    - **Return Null:** If the loop finishes without finding a match, `return null`.
- **`addTeacher` Method:**
    - Create a `public` method named `addTeacher` that takes a `Teacher` object as a parameter and has a `void` return type.
    - **Add to List:** Add the input `teacher` object to the `teachers` list using `teachers.add()`.
- **`updateTeacher` Method:**
    - Create a `public` method named `updateTeacher` that takes a `Teacher` object (containing the updated information) as input and has a `void` return type.
    - **Iterate:** Use a `for` loop with an index (e.g., `for (int i = 0; i < teachers.size(); i++)`) to iterate through the `teachers` list. You need the index to modify the list.
    - **Get Teacher:** Inside the loop, get the current `Teacher` object using `teachers.get(i)`.
    - **Check ID:** Compare the ID of the current `Teacher` object with the ID of the `updatedTeacher` object.
    - **Update:** If the IDs match:
        - Replace the `Teacher` object at the current index `i` in the `teachers` list with the `updatedTeacher` object using `teachers.set(i, updatedTeacher)`.
        - `return` from the method (you've found and updated the teacher).

- **`deleteTeacher` Method:**
  - Create a `public` method named `deleteTeacher` that takes an `int id` as input and has a `void` return type.
  - **Remove:** Use the `removeIf` method of the `teachers` list to remove the teacher with the matching ID. Use a lambda expression: `teachers.removeIf(teacher -> teacher.getId() == id);`

---

## MODULE DAO CLASS

- **Class Definition:**
  - Create a `public` class named `ModuleDAO`.
- **Imports:**
  - Import the following classes:
    - `com.example.model.Module` (This assumes a `Module` class in the `com.example.model` package. This class should have fields for `int id`, `String name`, and `Teacher teacher`, with associated getters and setters).
    - `com.example.model.Teacher` (You should have already created the Teacher class and TeacherDAO in the previous steps).
    - `java.util.ArrayList`
    - `java.util.List`
- **`modules` Field:**
  - Declare a `private static List<Module>` named `modules`. Initialize it with a new `ArrayList<>()`. This list will store `Module` objects.
- **Static Initialization Block:**
  - Create a `static` block (`static { ... }`).
  - Inside the static block:
    - **Create `TeacherDAO` Instance:** Create a new instance of the `TeacherDAO` class (e.g., `TeacherDAO teacherDAO = new TeacherDAO();`).
    - **Get Teachers:** Call the `getAllTeachers()` method on the `teacherDAO` object and store the result (a `List<Teacher>`) in a variable (e.g., `List<Teacher> teachers = teacherDAO.getAllTeachers();`).
    - **Add Modules:**
      - Add at least two `Module` objects to the `modules` list using `modules.add()`.
      - For each `Module`, create a new `Module` object using its constructor (e.g., `new Module(1, "Math", teachers.get(0))`).
        - Use different IDs and names for each module.
        - Use the `teachers` list obtained earlier to assign a `Teacher` object to each `Module`. You can access elements of the `teachers` list using `teachers.get(index)`. Make sure the indexes you use are valid (e.g., 0 and 1 if you have at least two teachers).

        - Add more `Module` objects if you want.

- **`getAllModules` Method:**
    - Create a `public` method named `getAllModules` that returns a `List<Module>`.
    - Inside the method, `return` the `modules` list.

- **`getModuleById` Method:**
    - Create a `public` method named `getModuleById` that takes an `int id` as a parameter and returns a `Module` object (or `null`).
    - **Iterate:** Use a `for-each` loop to iterate through the `modules` list.
    - **Check ID:** Inside the loop, check if the current `Module` object's ID (using `module.getId()`) matches the input `id`.
    - **Return Module:** If the IDs match, `return` the current `Module` object.
    - **Return Null:** If the loop completes without a match, `return null`.

- **`addModule` Method:**
    - Create a `public` method named `addModule` that takes a `Module` object as input and has a `void` return type.
    - **Add to List:** Add the input `module` to the `modules` list using `modules.add()`.

- **`updateModule` Method:**
    - Create a `public` method named `updateModule` that takes a `Module` object (with updated data) as input and has a `void` return type.
    - **Iterate:** Use a `for` loop with an index (e.g., `for (int i = 0; i < modules.size(); i++)`) to go through the `modules` list. You need the index to modify the list.
    - **Get Module:** Inside the loop, get the current `Module` object using `modules.get(i)`.
    - **Check ID:** Compare the ID of the current `Module` object with the ID of the `updatedModule`.
    - **Update:** If the IDs match:
        - Replace the `Module` object at index `i` in the `modules` list with the `updatedModule` using `modules.set(i, updatedModule)`.
        - `return` from the method (you've found and updated).

- **`deleteModule` Method:**
    - Create a `public` method named `deleteModule` that takes an `int id` as input and has a `void` return type.
    - **Remove:** Use the `removeIf` method of the `modules` list, with a lambda: `modules.removeIf(module -> module.getId() == id);`

## STEP 4: COM.EXAMPLE.RESOURCE PACKAGE

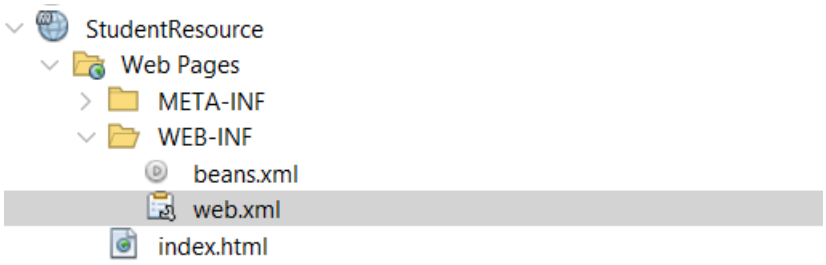
### ADDING DEPENDENCIES AND PLUGINS:

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>
        </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>8</source>
                    <target>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

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 only add the only 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>com.example.resource</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**
- **Please note that in the following resource classes for Teacher and Module, some section header are highlighted by RED showing that those parts must be done by yourself as independent exercises.**

## STUDENT RESOURCE CLASS

- **Class Definition:**
  - Create a `public` class named `StudentResource`.
- **Imports:**
  - Import the following classes:
    - `com.example.model.Module`
    - `com.example.model.Teacher`
    - `com.example.model.Student`
    - `com.example.dao.TeacherDAO`
    - `com.example.dao.ModuleDAO`
    - `com.example.dao.StudentDAO`
    - `java.util.ArrayList` (While not directly used in the provided code for existing methods, it's good practice to include since you're adding a method that might use it, and it's common with list handling)

    - `javax.ws.rs.*` (Imports all classes from `javax.ws.rs` for JAX-RS annotations)
    - `javax.ws.rs.core.MediaType`
    - `java.util.List`

- **Class-Level Annotations:**
    - Add the `@Path("/students")` annotation above the class definition.

- **DAO Fields:**
    - Declare `private` instance variables:
        - `StudentDAO studentDAO = new StudentDAO();`
        - `ModuleDAO moduleDAO = new ModuleDAO();`

- **`getAllStudents` Method:**
    - Create a `public` method named `getAllStudents` that returns a `List<Student>`.
    - **Annotations:**
        - `@GET`
        - `@Produces(MediaType.APPLICATION_JSON)`
    - **Method Body:**
        - `return` the result of calling `studentDAO.getAllStudents()`.

- **`getStudentById` Method:**
    - Create a `public` method named `getStudentById` that takes an `int studentId` and returns a `Student`.
    - **Annotations:**
        - `@GET`
        - `@Path("/{studentId}")`
        - `@Produces(MediaType.APPLICATION_JSON)`
        - `@PathParam("studentId")`
    - **Method Body:**
        - `return` the result of calling `studentDAO.getStudentById(studentId)`.

- **`addStudent` Method:**
    - Create a `public` method named `addStudent` that takes a `Student` object and has a `void` return type.
    - **Annotations:**
        - `@POST`
        - `@Consumes(MediaType.APPLICATION_JSON)`
    - **Method Body:**
        - Call `studentDAO.addStudent(student)`.

- **`updateStudent` Method:**
    - Create a `public` method named `updateStudent` that takes an `int studentId` and a `Student updatedStudent` as parameters and has a `void` return type.
    - **Annotations:**
        - `@PUT`
        - `@Path("/{studentId}")`

    - `@Consumes(MediaType.APPLICATION_JSON)`
    - `@PathParam("studentId")`
- **Method Body:**
    - Get existing student: Call `studentDAO.getStudentById(studentId)` and store result in variable called `existingStudent`.
    - Check if `existingStudent` is not `null`.
        - Set ID of updateStudent.
        - Call `studentDAO.updateStudent(updatedStudent)`.

- **`deleteStudent` Method:**

- Create a `public` method named `deleteStudent` that takes an `int studentId` and has a `void` return type.
- **Annotations:**
    - `@DELETE`
    - `@Path("/{studentId}")`
    - `@PathParam("studentId")`
- **Method Body:**
    - Call `studentDAO.deleteStudent(studentId)`.

- **`getModulesForStudent` Method:**

- Create a `public` method named `getModulesForStudent` that takes an `int studentId` as a parameter and returns a `String`. This method will return a JSON string.
- **Annotations:**
    - `@GET`
    - `@Path("/{studentId}/modules")`: This defines a *nested resource path*. It handles requests like `/students/5/modules`, where `5` is the `studentId`.
    - `@Produces(MediaType.APPLICATION_JSON)`: Indicates that this method returns JSON.
    - `@PathParam("studentId")`: Extracts the `studentId` from the URL.
- **Method Body:**
    - **Get Student:** Call `studentDAO.getStudentById(studentId)` and store the result in a `Student` variable (e.g., `student`).
    - **Check if Student Exists:** Use an `if` statement to check if `student` is not `null`.
        - **If Student Exists:**
            - Declare an integer variable named `moduleId` and assign 1 as its value.
            - call `moduleDAO.getModuleById(moduleId)` and store it in variable called `selectedModule`.
            - Create an inner `if` statement to check if selected module is not null. * create a JSON String. Return a JSON string containing the module's name and the teacher's name. The JSON should have this structure: `{"module": "moduleName", "teacher": "teacherName"}`. You'll use string concatenation to build this string.
        - **If Student Does Not Exist (or Module Not Found):**
            - `return` a JSON string indicating an error: `{"error": "Student or module not found"}`.

## TEACHER RESOURCE CLASS

- **Class Definition:**
  - Create a `public` class named `TeacherResource`.
- **Imports:**
  - Import the following classes:
    - `com.example.dao.TeacherDAO`
    - `com.example.model.Teacher`
    - `javax.ws.rs.*` (This imports all classes from the `javax.ws.rs` package, for JAX-RS annotations.)
    - `javax.ws.rs.core.MediaType`
    - `java.util.List`
- **Class-Level Annotations:**
  - Add the `@Path("/teachers")` annotation *before* the class definition. This sets the base URI path for all operations on this resource.
- **DAO Field:**
  - Declare a `private` instance variable:
    - `TeacherDAO teacherDAO = new TeacherDAO();` This creates an instance of the `TeacherDAO` class, which will be used to interact with the data layer.
- **`getAllTeachers` Method:**
  - Create a `public` method named `getAllTeachers` that returns a `List<Teacher>`.
  - **Annotations:**
    - `@GET`: Handles HTTP GET requests.
    - `@Produces(MediaType.APPLICATION_JSON)`: Returns data in JSON format.
  - **Method Body:**
    - `return` the result of calling `teacherDAO.getAllTeachers()`.
- **`getTeacherById` Method:**
  - Create a `public` method named `getTeacherById` that takes an `int teacherId` as a parameter and returns a `Teacher` object.
  - **Annotations:**
    - `@GET`
    - `@Path("/{teacherId}")`: Handles requests to paths like `/teachers/123`, where `123` is the `teacherId`.
    - `@Produces(MediaType.APPLICATION_JSON)`
    - `@PathParam("teacherId")`: Extracts the `teacherId` from the URL.
  - **Method Body:**
    - `return` the result of calling `teacherDAO.getTeacherById(teacherId)`.
- **`addTeacher` Method:**

- Create a `public` method named `addTeacher` that takes a `Teacher` object as a parameter and has a `void` return type.
- **Annotations:**
  - `@POST`: Handles POST requests.
  - `@Consumes(MediaType.APPLICATION_JSON)`: Accepts JSON data in the request body.
- **Method Body:**
  - Call `teacherDAO.addTeacher(teacher)`.

- **`updateTeacher` Method:**

  - Create a `public` method named `updateTeacher` that takes an `int teacherId` and a `Teacher updatedTeacher` as parameters, and has a `void` return type.
  - **Annotations:**
    - `@PUT`: Handles PUT requests.
    - `@Path("/{teacherId}")`
    - `@Consumes(MediaType.APPLICATION_JSON)`
    - `@PathParam("teacherId")`: Extracts the `teacherId` from the URL.
  - **Method Body:**
    - Get existing Teacher by calling the `getTeacherById` of the teacherDAO object and pass `teacherId`, store the returned result inside `existingTeacher` object.
    - Create an if statement and check if the `existingTeacher` object is not equal to null. If it is true
      - set the Id of the `updatedTeacher` object using the value of the `teacherId`.
      - Call the `teacherDAO.updateTeacher(updatedTeacher)` method to save the updates.

- **`deleteTeacher` Method:**

  - Create a `public` method named `deleteTeacher` that takes an `int teacherId` as a parameter and has a `void` return type.
  - **Annotations:**
    - `@DELETE`
    - `@Path("/{teacherId}")`
    - `@PathParam("teacherId")`
  - **Method Body:**
    - Call `teacherDAO.deleteTeacher(teacherId)`.

---

## MODULE RESOURCE CLASS

- **Class Definition:**

  - Create a `public` class named `ModuleResource`.

- **Imports:**

  - Import the following classes:
    - `com.example.dao.ModuleDAO`

    - `com.example.model.Module`
    - `java.util.ArrayList`
    - `javax.ws.rs.*` (Imports all classes from `javax.ws.rs` for JAX-RS annotations)
    - `javax.ws.rs.core.MediaType`
    - `java.util.List`

- **Class-Level Annotations:**
    - Add the `@Path("/modules")` annotation above the class definition. This sets the base path for all operations on this resource.

- **DAO Field:**
    - Declare a `private` instance variable:
        - `ModuleDAO moduleDAO = new ModuleDAO();` This creates an instance of the `ModuleDAO` for data access.

- **`getAllModules` Method:**
    - Create a `public` method named `getAllModules` that returns a `List<Module>`.
    - **Annotations:**
        - `@GET`
        - `@Produces(MediaType.APPLICATION_JSON)`
    - **Method Body:**
        - `return` the result of calling `moduleDAO.getAllModules()`.

- **`getModuleById` Method:**
    - Create a `public` method named `getModuleById` that takes an `int moduleId` as a parameter and returns a `Module` object.
    - **Annotations:**
        - `@GET`
        - `@Path("/{moduleId}")`
        - `@Produces(MediaType.APPLICATION_JSON)`
        - `@PathParam("moduleId")`
    - **Method Body:**
        - `return` the result of calling `moduleDAO.getModuleById(moduleId)`.

- **`addModule` Method:**
    - Create a `public` method named `addModule` that takes a `Module` object as input and has a `void` return type.
    - **Annotations:**
        - `@POST`
        - `@Consumes(MediaType.APPLICATION_JSON)`
    - **Method Body:**
        - Call `moduleDAO.addModule(module)`.

- **`updateModule` Method:**

- Create a `public` method named `updateModule` that takes an `int moduleId` and a `Module updatedModule` as parameters and has a `void` return type.
- **Annotations:**
  - `@PUT`
  - `@Path("/{moduleId}")`
  - `@Consumes(MediaType.APPLICATION_JSON)`
  - `@PathParam("moduleId")`
- **Method Body:**
  - Get `existingModule` by calling `getModuleById()` method of the `moduleDAO` object and passing `moduleId` argument.
  - Use an `if` statement to Check if `existingModule` is not null.
    - Set the ID of the `updatedModule` to the value of `moduleId`.
    - Call `moduleDAO.updateModule(updatedModule)`.

- **`deleteModule` Method:**

  - Create a `public` method named `deleteModule` that takes an `int moduleId` as input and has a `void` return type.
  - **Annotations:**
    - `@DELETE`
    - `@Path("/{moduleId}")`
    - `@PathParam("moduleId")`
  - **Method Body:**
    - Call `moduleDAO.deleteModule(moduleId)`.

- **`getModulesByTeacher` Method:**

  - Create a `public` method named `getModulesByTeacher` that takes an `int teacherId` as a parameter and returns a `List<Module>`.
  - **Annotations:**
    - `@GET`
    - `@Path("/teachers/{teacherId}")`: This defines a nested resource path. It handles requests to URLs like `/modules/teachers/5` (where 5 is the `teacherId`).
    - `@Produces(MediaType.APPLICATION_JSON)`
    - `@PathParam("teacherId")`
  - **Method Body:**
    - **Create Result List:** Create a new `ArrayList` to store the modules taught by the specified teacher (e.g., `List<Module> modulesByTeacher = new ArrayList<>();`).
    - **Get All Modules:** Call `moduleDAO.getAllModules()` and store the result in a `List<Module>` variable (e.g., `List<Module> allModules = moduleDAO.getAllModules();`).
    - **Iterate and Filter:**
      - Use a `for-each` loop to iterate through the `allModules` list.
      - Inside the loop, for each `Module` object:
        - Get the `Teacher` object associated with the module (using `module.getTeacher()`).
        - Get the ID of the `Teacher` object (using `getTeacher().getId()`).
        - Compare the teacher's ID with the `teacherId` parameter.
        - If the IDs match, add the current `Module` object to the `modulesByTeacher` list.
    - **Return Result:** After the loop finishes, `return` the `modulesByTeacher` list.

## TESTING THE API

Please test the application using POSTMAN and share your results with your instructor.

## INTEGRATE DATA OPERATIONS IN RESOURCE CLASSES (NEXT WEEK'S TUTORIAL PREPARATION)

After doing this tutorial and implementing all DAO classes, try to remove all DAO classes and integrate CRUD operations in resource classes. Therefore, you will need to remove DAO classes.