3. **Verify ProductController.java:**
    - Located at src/main/java/com/example/demo/controller/ProductController.java.
    - It should contain @RestController, @RequestMapping("/api/products"), and the createProduct (POST), getAllProducts (GET), getProductById (GET with Path Variable), updateProduct (PUT), and deleteProduct (DELETE) methods.
    - **Crucially, ensure you have the searchProducts method (GET with Request Parameters) as shown in the "Creating REST API using Spring Boot (Putting it all together)" slide of the PPT.** If not, add it now:

```
// Add this method to your existing ProductController.java
// Make sure to add 'import java.util.stream.Collectors;' if not already present.
@GetMapping("/search")
public ResponseEntity<List<Product>> searchProducts(
    @RequestParam(required = false) String name,
    @RequestParam(required = false, defaultValue = "0.0") double minPrice) {

    System.out.println("Searching products with name: " + name + ", minPrice: " +
minPrice);

    List<Product> filteredProducts = products.stream()
        .filter(p -> (name == null ||
p.getName().toLowerCase().contains(name.toLowerCase())))
        .filter(p -> p.getPrice() >= minPrice)
        .collect(Collectors.toList());

    return new ResponseEntity<>(filteredProducts, HttpStatus.OK);
}
```

    - You may need to import the Collectors class add following line to Import block

```
import java.util.stream.Collectors;
```

    - **Explanation:** This controller defines the **Resource Identification** (e.g.,

/api/products for the collection, /api/products/{id} for a specific item, /api/products/search for a filtered collection) and leverages HTTP methods for **Resource Actions**.

Step 2: Run Your Spring Boot Application

Start your Spring Boot application so you can interact with its API endpoints.

1. **From IDE:** Right-click on your main application class (e.g., DemoApplication.java) and select "Run 'DemoApplication.main()'".
2. **From Command Line (Maven):** Open your terminal, navigate to the project root, and run:
   mvn spring-boot:run

   Confirm that the application starts successfully, typically on http://localhost:8080.

Step 3: Explore Resource Identification and Path Variables

Let's test how individual resources are identified using Path Variables.

1. **Create a Product (if you haven't already):**
   - **Purpose:** We need at least one product to retrieve by ID.
   - **Method:** POST
   - **URL:** http://localhost:8080/api/products
   - **Headers:** Content-Type: application/json
   - **Body (raw, JSON):** {"name": "Wireless Headphones", "price": 199.99}
   - Send this request. Note the id returned (e.g., 1).
2. **Retrieve a Specific Product using Path Variable:**
   - **Purpose:** Demonstrate GET with a Path Variable to identify a unique resource.
   - **Method:** GET
   - **URL:** http://localhost:8080/api/products/{id} (replace {id} with the ID from the previous step, e.g., http://localhost:8080/api/products/1)

     Using curl:curl http://localhost:8080/api/products/1
     Using Postman:
   - Set **Method** to GET.

- Enter **URL**: http://localhost:8080/api/products/1.
- Click **Send**.
- **Expected Result:** 200 OK status and the JSON representation of the product.
- **Observation:** The id is part of the URI path, making it clear we are requesting a *specific* product resource.

Step 4: Explore Passing Parameters via Request Parameters (Query Parameters)

Now, let's use query parameters to filter a collection of resources.

1. **Create More Products (Optional, for better filtering demo):**
   - Send a few more POST requests to http://localhost:8080/api/products with different product names and prices (e.g., "Gaming Mouse", 75.00; "Mechanical Keyboard", 150.00).
2. **Search Products by Name (Request Parameter):**
   - **Purpose:** Demonstrate GET with a name query parameter.
   - **Method:** GET
   - **URL:** http://localhost:8080/api/products/search?name=head (to find "Wireless Headphones")

     Using curl:curl "http://localhost:8080/api/products/search?name=head"
     Using Postman:
   - Set **Method** to GET.
   - Enter **URL**: http://localhost:8080/api/products/search?name=head.
   - Click **Send**.
   - **Expected Result:** 200 OK status and a JSON array containing products whose names include "head".
3. **Search Products by Minimum Price (Request Parameter):**
   - **Purpose:** Demonstrate GET with a minPrice query parameter.
   - **Method:** GET
   - **URL:** http://localhost:8080/api/products/search?minPrice=100

     Using curl:curl "http://localhost:8080/api/products/search?minPrice=100"

Using Postman:

- Set **Method** to GET.
- Enter **URL**: http://localhost:8080/api/products/search?minPrice=100.
- Click **Send**.
- **Expected Result:** 200 OK status and a JSON array containing products with a price greater than or equal to 100.00.
- **Observation:** Query parameters are used for filtering or providing optional criteria, appended after ?.

Step 5: Understand Request Body for Complex Data (HTTP/JSON)

While covered in the CRUD activity, it's essential to reiterate the role of the Request Body for sending complex data as **Representations**.

1. **Revisit Create (POST) or Update (PUT):**
   - **Purpose:** Observe how JSON data is sent in the request body to represent a resource.
   - **Method:** POST or PUT
   - **URL:** http://localhost:8080/api/products (for POST) or http://localhost:8080/api/products/{id} (for PUT)
   - **Headers: Crucially, ensure Content-Type: application/json is set.** This header tells the server that the body contains JSON data.
   - **Body (raw, JSON):** Provide a JSON object representing the product.

Observation: The request body allows you to send a full, structured representation of the resource, which Spring Boot automatically converts into your Product Java object using @RequestBody.

Step 6: Reflect on Statelessness

Your current in-memory ProductController inherently demonstrates statelessness.

1. **Restart your Spring Boot application.**
2. **Try to retrieve products again:**

    - **Method:** GET
    - **URL:** http://localhost:8080/api/products
    - **Expected Result:** An empty array [].
    - **Observation:** Since our "database" is just an in-memory list, restarting the server clears all data. This is a direct consequence of statelessness: the server doesn't remember previous interactions (like products you added before restarting). Each request is treated independently. In a real application, data would persist in a database, making the *API itself* stateless, even if the underlying data store is stateful.
3. **Consider a "stateful" scenario (mental exercise):**
    - Imagine if, after adding a product, the server remembered *your specific session* and only showed *your* products. This would be stateful.
    - In a RESTful API, if you wanted to see products *only you* added, you would send an authentication token with each request. The server would then use this token to identify you and retrieve *your* products from the database, without storing your session state on the server.

You have successfully explored key aspects of RESTful API design and implementation in Spring Boot:

- **Resource Identification:** Using URIs and Path Variables to pinpoint specific resources.
- **Parameter Passing:** Employing Request Parameters for filtering and Request Body for complex data representations (JSON).
- **HTTP Methods for Actions:** Reinforcing how GET, POST, PUT, DELETE convey intent.
- **Statelessness:** Observing how the API operates without server-side session state, contributing to scalability and reliability.

This activity reinforces that REST is about interacting with resources in a uniform, stateless manner using standard HTTP semantics, making your APIs robust and easy to integrate.

Activity 8.1: Spring Boot Database & JDBC with Oracle DB

This guide will walk you through connecting your existing Spring Boot application to an Oracle Database 23ai instance using JDBC and Spring's JdbcTemplate. You will learn to configure the database connection, initialize the schema, and perform CRUD operations against the actual database.

Objective

By the end of this activity, you will have a Spring Boot application that:

- Connects to your Oracle Database 23ai instance.
- Uses JdbcTemplate to perform CRUD operations on a PRODUCTS table.
- Demonstrates database schema and data initialization on startup.
- Utilizes HikariCP for efficient connection pooling.

Prerequisites

- **Completed the "Spring Boot AOP Activity Guide."** You should have a working Spring Boot project (e.g., demo or product-api) with Product model, ProductService, LoggingAspect, and ProductController.
- **Oracle Database 23ai running in your VM.** Ensure you know its IP address, listener port (default 1521), and the service name of your Pluggable Database (PDB) (e.g., XEPDB1).
- **Java Development Kit (JDK) 21.**
- **Apache Maven 3.6+** (or Gradle).
- An Integrated Development Environment (IDE) like **IntelliJ IDEA**, **Eclipse (with STS)**, or **VS Code (with Spring Boot extensions)**.
- An API testing tool like **Postman** or **curl**.