        - **Name:** demo
        - **Description:** Demo project for Spring Boot CRUD API
        - **Package name:** com.example.demo
        - **Packaging:** Jar
        - **Java:** Choose your installed JDK version (e.g., 17).
    - **Dependencies:** Click "Add Dependencies..." and search for and add:
        - Spring Web (for building web, including RESTful, applications)
        - Spring Boot DevTools (optional, for faster development cycles)
3. **Generate and Download:** Click the "Generate" button. A .zip file will be downloaded to your computer.
4. **Unzip and Import into IDE:**
    - Unzip the downloaded demo.zip (or product-api.zip) file.
    - Open your preferred IDE (IntelliJ IDEA, Eclipse, VS Code).
    - **For IntelliJ IDEA:** Select File -> Open and navigate to the unzipped project folder.
    - **For Eclipse (STS):** Select File -> Import -> Maven -> Existing Maven Projects, then browse to the unzipped project folder.
    - **For VS Code:** Select File -> Open Folder and open the unzipped project folder.

Step 2: Define the Model (Product POJO)

A Plain Old Java Object (POJO) will represent our Product resource. This class will hold the data that our API will manage.

1. **Create a new package:** Inside src/main/java/com/example/demo, create a new package named model.
2. **Create Product.java:** Inside the model package, create a new Java class named Product.java and add the following code:

```
// src/main/java/com/example/demo/model/Product.java
package com.example.demo.model;

public class Product {
    private Long id;
```

```
private String name;
private double price;

// Constructors
public Product() {
    // Default constructor
}

public Product(Long id, String name, double price) {
    this.id = id;
    this.name = name;
    this.price = price;
}

// Getters and Setters for all fields
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getName() {
    return name;
}

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

```
    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", price=" + price +
                '}';
    }
}
```

- **Explanation:** This Product class has id, name, and price fields, along with constructors, getters, and setters. Spring will use these to convert JSON requests into Product objects and vice-versa.

Step 3: Implement the REST Controller (ProductController)

Now, we'll create the REST controller that will handle incoming HTTP requests for our CRUD operations. We'll use an in-memory list to simulate a database for simplicity.

1. **Create a new package:** Inside src/main/java/com/example/demo, create a new package named controller.
2. **Create ProductController.java:** Inside the controller package, create a new Java class named ProductController.java and add the following code:
   // src/main/java/com/example/demo/controller/ProductController.java
   package com.example.demo.controller;

```
import com.example.demo.model.Product; // Import our Product model
import org.springframework.http.HttpStatus; // For HTTP status codes
import org.springframework.http.ResponseEntity; // For controlling the full HTTP response
import org.springframework.web.bind.annotation.*; // For REST annotations

import java.util.ArrayList;
import java.util.List;
import java.util.Optional; // For handling potential absence of a product
import java.util.concurrent.atomic.AtomicLong; // For generating unique IDs

@RestController // Marks this class as a REST controller, combining @Controller and @ResponseBody
@RequestMapping("/api/products") // Base path for all endpoints in this controller
public class ProductController {

    // --- In-memory "database" for demonstration purposes ---
    // In a real application, this would be a Service layer interacting with a database.
    private final List<Product> products = new ArrayList<>();
    private final AtomicLong counter = new AtomicLong(); // Simple ID generator

    // --- 1. Implement CREATE (POST) ---
    // Handles HTTP POST requests to /api/products
    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product product) {
        // @RequestBody: Binds the HTTP request body (JSON) to the Product object.
        // ResponseEntity: Allows us to control the HTTP status code and response body.
```

```
        product.setId(counter.incrementAndGet()); // Assign a new unique ID
        products.add(product); // Add the new product to our list (simulated save)
        System.out.println("Created product: " + product.toString()); // Log the creation
        // Return 201 Created status with the newly created product in the response
body
        return new ResponseEntity<>(product, HttpStatus.CREATED);
    }

    // --- 2. Implement READ (GET) ---
    // Handles HTTP GET requests to /api/products (get all)
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        System.out.println("Fetching all products.");
        // Return 200 OK status with the list of all products
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // Handles HTTP GET requests to /api/products/{id} (get by ID)
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        // @PathVariable: Extracts the 'id' from the URL path.
        System.out.println("Fetching product with ID: " + id);

        // Find the product by ID in our list
        Optional<Product> foundProduct = products.stream()
                                         .filter(p -> p.getId().equals(id))
                                         .findFirst();

        if (foundProduct.isPresent()) {
            // If found, return 200 OK with the product
```

```
            return new ResponseEntity<>(foundProduct.get(), HttpStatus.OK);
        } else {
            // If not found, return 404 Not Found
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    // --- 3. Implement UPDATE (PUT) ---
    // Handles HTTP PUT requests to /api/products/{id}
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        System.out.println("Updating product with ID: " + id);

        Optional<Product> existingProductOptional = products.stream()
                                                .filter(p -> p.getId().equals(id))
                                                .findFirst();

        if (existingProductOptional.isPresent()) {
            Product existingProduct = existingProductOptional.get();
            // Update the existing product's details with the new data
            existingProduct.setName(productDetails.getName());
            existingProduct.setPrice(productDetails.getPrice());
            System.out.println("Updated product: " + existingProduct.toString());
            // Return 200 OK with the updated product
            return new ResponseEntity<>(existingProduct, HttpStatus.OK);
        } else {
            // If product not found, return 404 Not Found
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
```

```
    // --- 4. Implement DELETE (DELETE) ---
    // Handles HTTP DELETE requests to /api/products/{id}
    @DeleteMapping("/{id}")
    public ResponseEntity<HttpStatus> deleteProduct(@PathVariable Long id) {
        System.out.println("Attempting to delete product with ID: " + id);
        // Remove the product from the list if its ID matches
        boolean removed = products.removeIf(p -> p.getId().equals(id));

        if (removed) {
            // If successfully removed, return 204 No Content
            System.out.println("Product with ID " + id + " deleted.");
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        } else {
            // If product was not found, return 404 Not Found
            System.out.println("Product with ID " + id + " not found for deletion.");
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

- **Explanation:**
  - @RestController: Combines @Controller and @ResponseBody. It tells Spring that this class handles web requests and that the return value of methods should be bound directly to the web response body.
  - @RequestMapping("/api/products"): Sets the base URI path for all methods in this controller.
  - List<Product> products: A simple ArrayList to act as our in-memory data store. Data will be lost when the application restarts.
  - AtomicLong counter: Used to generate simple, unique IDs for new products.
  - **@PostMapping**: Maps HTTP POST requests. Used for creating new

resources.

- **@GetMapping**: Maps HTTP GET requests. Used for retrieving resources.
- **@PutMapping**: Maps HTTP PUT requests. Used for updating existing resources.
- **@DeleteMapping**: Maps HTTP DELETE requests. Used for deleting resources.
- **@RequestBody**: Used with POST and PUT to bind the JSON (or XML) payload from the request body to a Java object (Product in this case).
- **@PathVariable**: Used with GET, PUT, DELETE to extract values from the URI path (e.g., id from /products/{id}).
- **ResponseEntity<T>**: Allows you to return a specific HTTP status code (HttpStatus) along with the response body (T). This gives you full control over the HTTP response.

Step 4: Run the Spring Boot Application

Now that the code is written, let's run the application.

1. **Locate the main application class:** In your IDE, find src/main/java/com/example/demo/DemoApplication.java (or ProductApiApplication.java if you named it differently).
2. **Run the application:**
   - **From IDE:** Right-click on the DemoApplication.java file and select "Run 'DemoApplication.main()'" (or similar option).
   - **From Command Line (Maven):** Open your terminal or command prompt, navigate to the root directory of your project (where pom.xml is located), and run:
     mvn spring-boot:run

You should see output indicating that Spring Boot is starting and an embedded Tomcat server is running on port 8080.

Step 5: Test the API Endpoints

Now, let's use curl (command-line) or Postman (GUI) to interact with your API. Ensure your Spring Boot application is running before testing.

Test 1: CREATE a Product (POST)

- **Endpoint:** http://localhost:8080/api/products
- **Method:** POST
- **Request Body (JSON):**

```
{
    "name": "Smartphone",
    "price": 699.99
}
```

Using curl:

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"Smartphone", "price":699.99}' http://localhost:8080/api/products
```

Expected Response: 201 Created status code, and a JSON response similar to:

```
{
    "id": 1,
    "name": "Smartphone",
    "price": 699.99
}
```

Using Postman:

1. Set **Method** to POST.
2. Enter **URL**: http://localhost:8080/api/products.
3. Go to **Headers** tab: Add Content-Type with value application/json.
4. Go to **Body** tab: Select raw and JSON. Paste the JSON request body.

5. Click **Send**.

Test 2: READ All Products (GET)

- **Endpoint:** http://localhost:8080/api/products
- **Method:** GET

Using curl:

```
curl http://localhost:8080/api/products
```

Expected Response: 200 OK status code, and a JSON array containing the product(s) you created:

```
[
   {
      "id": 1,
      "name": "Smartphone",
      "price": 699.99
   }
]
```

Using Postman:

1. Set **Method** to GET.
2. Enter **URL**: http://localhost:8080/api/products.
3. Click **Send**.

Test 3: READ a Product by ID (GET)

- **Endpoint:** http://localhost:8080/api/products/1 (assuming ID 1 was created)
- **Method:** GET

Using curl:

```
curl http://localhost:8080/api/products/1
```

Expected Response: 200 OK status code, and a JSON object for the product with ID 1.
If you try an ID that doesn't exist (e.g., /products/99), you should get 404 Not Found with an empty response body.

Using Postman:

1. Set **Method** to GET.
2. Enter **URL**: http://localhost:8080/api/products/1.
3. Click **Send**.

Test 4: UPDATE a Product (PUT)

- **Endpoint:** http://localhost:8080/api/products/1 (assuming ID 1 exists)
- **Method:** PUT
- **Request Body (JSON):**

```
{
    "name": "Latest Smartphone Model",
    "price": 749.99
}
```

Using curl:

curl -X PUT -H "Content-Type: application/json" -d '{"name":"Latest Smartphone Model", "price":749.99}' http://localhost:8080/api/products/1

Expected Response: 200 OK status code, and a JSON response with the updated product details:

```
{
    "id": 1,
    "name": "Latest Smartphone Model",
    "price": 749.99
}
```

Using Postman:

1. Set **Method** to PUT.
2. Enter **URL**: http://localhost:8080/api/products/1.
3. Go to **Headers** tab: Add Content-Type with value application/json.
4. Go to **Body** tab: Select raw and JSON. Paste the JSON request body.
5. Click **Send**.

Test 5: DELETE a Product (DELETE)

- **Endpoint:** http://localhost:8080/api/products/1 (assuming ID 1 exists)
- **Method:** DELETE

Using curl:

curl -X DELETE http://localhost:8080/api/products/1

Expected Response: 204 No Content status code, with an empty response body.
If you try to delete an ID that doesn't exist, you should get 404 Not Found.

Using Postman:

1. Set **Method** to DELETE.
2. Enter **URL**: http://localhost:8080/api/products/1.
3. Click **Send**.

Step 6: Basic Error Handling (Optional Enhancement)

To make your API more robust, you can implement custom error handling.

1. Create ProductNotFoundException.java:
   Inside src/main/java/com/example/demo/controller (or a new exception package), create this class:
   // src/main/java/com/example/demo/controller/ProductNotFoundException.java
   package com.example.demo.controller; // Or com.example.demo.exception;

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND) // This annotation automatically sets
the HTTP status to 404
public class ProductNotFoundException extends RuntimeException {
    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

2. Modify getProductById and updateProduct and deleteProduct in ProductController.java:
   Change the else blocks to throw this exception:

```
// In ProductController.java, modify getProductById:
@GetMapping("/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    System.out.println("Fetching product with ID: " + id);
    return products.stream()
                .filter(p -> p.getId().equals(id))
                .findFirst()
                .map(p -> new ResponseEntity<>(p, HttpStatus.OK))
                .orElseThrow(() -> new ProductNotFoundException("Product not found
with ID: " + id));
}

// In ProductController.java, modify updateProduct:
@PutMapping("/{id}")
public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
    System.out.println("Updating product with ID: " + id);
    return products.stream()
                .filter(p -> p.getId().equals(id))
                .findFirst()
                .map(existingProduct -> {
                    existingProduct.setName(productDetails.getName());
```

```
                    existingProduct.setPrice(productDetails.getPrice());
                    System.out.println("Updated product: " + existingProduct.toString());
                    return new ResponseEntity<>(existingProduct, HttpStatus.OK);
                })
                .orElseThrow(() -> new ProductNotFoundException("Product not found
with ID: " + id));
}

// In ProductController.java, modify deleteProduct:
@DeleteMapping("/{id}")
public ResponseEntity<HttpStatus> deleteProduct(@PathVariable Long id) {
    System.out.println("Attempting to delete product with ID: " + id);
    boolean removed = products.removeIf(p -> p.getId().equals(id));

    if (removed) {
        System.out.println("Product with ID " + id + " deleted.");
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    } else {
        throw new ProductNotFoundException("Product not found with ID: " + id + " for
deletion.");
    }
}
```

- **Explanation:** Now, if a product is not found, a ProductNotFoundException is thrown. Because this exception is annotated with @ResponseStatus(HttpStatus.NOT_FOUND), Spring Boot will automatically convert it into an HTTP 404 Not Found response, along with a default error body containing the exception message.

You have successfully created a Spring Boot application, defined a simple data model, and implemented RESTful API endpoints for all CRUD operations (Create, Read, Update, Delete) using POST, GET, PUT, and DELETE HTTP methods. You've also learned how to use ResponseEntity for fine-grained control over HTTP responses and implemented basic error handling.

Activity 6.1: RESTful API Overview & Implementation

This guide will walk you through exploring the core concepts of RESTful APIs, focusing on resource identification, various parameter passing mechanisms (Path Variables, Request Parameters, Request Body), and the principle of statelessness, all within a Spring Boot context.

Objective

By the end of this activity, you will have a deeper understanding of RESTful API design principles and how to implement them effectively using Spring Boot, particularly regarding how clients interact with resources and pass data.

Prerequisites

- **Completed the "Implementing CRUD operations using HTTP methods" activity.** This guide builds upon the Product model and ProductController you created there.
- **Java Development Kit (JDK) 17 or higher**
- **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 a command-line tool like **curl**

Step 1: Review Your Existing Spring Boot Project

Ensure your Spring Boot project from the previous CRUD activity is set up and ready. We will be using the Product model and the ProductController you previously created.

1. **Open your Project:** Open the Spring Boot project you used for the CRUD operations activity in your IDE.
2. **Verify Product.java:**
    - Located at src/main/java/com/example/demo/model/Product.java.
    - It should have id, name, and price fields with constructors, getters, and setters. This represents our **Resource Representation** (how the data looks in JSON).