1. Creating the Customer Management API

Introduction

The Customer Management API is a Spring Boot project designed to manage customer data. This project offers a RESTful interface for performing CRUD (Create, Read, Update, Delete) operations on customer data. The primary goal is to provide a backend service that stores customer information in memory and can be easily accessed and manipulated through HTTP requests.

Code Implementation Overview

Project Setup

1. **Create the Project**:
    - Visit Spring Initializer.
    - Configure the project with the following settings:
        - Project: Maven
        - Language: Java
        - Spring Boot: Choose the latest stable version
        - Group: com.example
        - Artifact: customer-management-api
        - Name: customer-management-api
        - Description: API for managing customers
        - Package name: com.example.demo
        - Packaging: Jar
        - Java: 11 (or your preferred version)
    - Add the dependency: Spring Web.
    - Click "Generate" to download the project as a ZIP file.
    - Extract the ZIP file to your desired location.
    - Open your IDE (VS Code or Eclipse) and import the project as a Maven project.

Customer Entity

1. **Create the Customer Entity**:
    - In the com.example.demo package, create a new Java class named `Customer`.
    - Define attributes `id` (Long) and `name` (String).
    - Implement constructors, getters, and setters for these attributes.

Customer Service

1. **Create the Customer Service**:
    - In the com.example.demo package, create a new Java class named `CustomerService`.
    - Manage an in-memory list of customers using a `List<Customer>`.
    - Initialize the list with three customers: John Doe, Jane Adams, and Alice Jones.
    - Provide methods for:
        - Getting all customers
        - Getting a customer by ID
        - Creating a new customer
        - Updating an existing customer
        - Deleting a customer

Customer Controller

1. **Create the Customer Controller**:
    - In the com.example.demo package, create a new Java class named `CustomerController`.
    - Annotate the class with `@RestController` and `@RequestMapping("/customers")`.
    - Inject the `CustomerService` into the controller.
    - Map methods to HTTP verbs:
        - GET for retrieving all customers and retrieving a customer by ID
        - POST for creating a new customer
        - PUT for updating an existing customer
        - DELETE for deleting a customer

Application Configuration

1. **Create the Main Application Class**:
    - In the com.example.demo package, create a new Java class named `DemoApplication`.
    - Annotate the class with `@SpringBootApplication`.
    - Implement the main method to bootstrap the Spring Boot application.

Testing

Test the API:

- Use tools like Postman or cURL to perform HTTP requests.
- Perform basic CRUD operations to ensure the endpoints are functioning as expected.

2. Testing CRUD Operations Using a Browser

To test CRUD operations for the `@RestController`, you typically use REST clients or tools like Postman, but some operations can be tested using a browser (mostly GET operations).

Steps to Test Using a Browser

GET All Customers:

- Open a browser and go to `http://localhost:8080/customers` to retrieve all customers.

GET Customer by ID:

- Open a browser and go to `http://localhost:8080/customers/{id}` (replace `{id}` with an actual customer ID) to retrieve a specific customer.

3. Testing CRUD Operations Using Postman

Steps to Test Using Postman

Start Spring Boot Application:

- Run your Spring Boot application from your IDE or using the command line: `mvn spring-boot:run`.

Open Postman:

- Open the Postman application.

Create a New Request:

GET All Customers:

- Method: GET
- URL: `http://localhost:8080/customers`
- Click "Send".

GET Customer by ID:

- Method: GET
- URL: `http://localhost:8080/customers/{id}` (replace `{id}` with actual ID)
- Click "Send".

POST Create Customer:

- Method: POST
- URL: `http://localhost:8080/customers`
- Body: JSON (e.g., `{"name": "John Doe"}`)
- Click "Send".

PUT Update Customer:

- Method: PUT
- URL: `http://localhost:8080/customers/{id}` (replace `{id}` with actual ID)
- Body: JSON (e.g., `{"name": "John Doe Updated"}`)
- Click "Send".

DELETE Customer:

- Method: DELETE
- URL: `http://localhost:8080/customers/{id}` (replace `{id}` with actual ID)
- Click "Send".

4. Create a Web Client Project

Introduction

The Customer Management Web Client is a separate Spring Boot project designed to provide a web-based interface for interacting with the Customer Management API. This project aims to create a user-friendly front-end application where users can perform CRUD operations on customer data through a web browser.

1. Create a New Spring Boot Project

Using Spring Initializer

1. **Go to Spring Initializer**:
   - Visit Spring Initializer.
2. **Configure the Project**:
   - Project: Maven.
   - Language: Java.
   - Spring Boot: Choose the latest stable version.
   - Group: com.example.
   - Artifact: customer-web-client.
   - Name: customer-web-client.
   - Description: Web client for customer management.
   - Package name: com.example.webclient.
   - Packaging: Jar.
   - Java: 11 (or your preferred version).
3. **Add Dependencies**:
   - Add the following dependencies:
     - Spring Web.
     - Thymeleaf.
4. **Generate and Download**:
   - Click "Generate" to download the project as a ZIP file.
   - Extract the ZIP file to your desired location.
5. **Import the Project**:
   - Open your preferred IDE (e.g., VS Code, Eclipse).
   - Import the project as a Maven project.

2. Configure the Web Client Project

Controller

1. **Create the Controller Class**:
   - In the com.example.webclient package, create a new Java class named `CustomerWebController`.
   - Annotate the class with `@Controller` and `@RequestMapping("/web/customers")`.
   - Inject a `RestTemplate` bean to facilitate HTTP communication with the Customer Management API.
2. **Implement Methods**:
   - `getAllCustomers`: Use `RestTemplate` to retrieve all customers from the API and add them to the model to be displayed in the `customers.html` template.
   - `createCustomerForm`: Add a new `Customer` object to the model to be used in the `customer-form.html` template.
   - `createCustomer`: Use `RestTemplate` to send a POST request to the API to create a new customer, then redirect to the customer list.
   - `editCustomerForm`: Use `RestTemplate` to retrieve a specific customer by ID from the API and add it to the model to be used in the `customer-form.html` template.
   - `updateCustomer`: Use `RestTemplate` to send a PUT request to the API to update an existing customer, then redirect to the customer list.
   - `deleteCustomer`: Use `RestTemplate` to send a DELETE request to the API to delete a customer, then redirect to the customer list.

Customer Model

1. **Create the Customer Class**:
   - In the com.example.webclient package, create a new Java class named `Customer`.
   - Define attributes `id` (Long) and `name` (String).
   - Implement constructors, getters, and setters for these attributes.

Main Application Class

1. **Create the Main Application Class**:
   - In the com.example.webclient package, create a new Java class named `CustomerWebClientApplication`.
   - Annotate the class with `@SpringBootApplication`.
   - Implement the main method to bootstrap the Spring Boot application.
   - Define a `RestTemplate` bean method to facilitate HTTP communication.

Thymeleaf Templates

1. **Create `customers.html`**:
   - Define an HTML template to display a table of customers.
   - Include a link to add a new customer, and options to edit or delete each customer.
2. **Create `customer-form.html`**:
   - Define an HTML template to display a form for creating or updating a customer.
   - Include fields for id and name, and a submit button.

Summary

- Create a new Spring Boot project using Spring Initializer with dependencies Spring Web and Thymeleaf.
- Implement the controller (CustomerWebController) to interact with the Customer API using RestTemplate.
- Define the Customer model to match the structure of the customer data.
- Create Thymeleaf templates to display and manage customers.

With these steps, you have a web client project that interacts with the Customer project, providing an HTML interface with separate buttons for Create, Read, Update, and Delete operations.

5. Converting from List of Customers to H2 Database

Introduction

To convert the in-memory list of customers to an H2 database, you will use Spring Data JPA. This will allow you to persist customer data in a relational database and provide more robust data management capabilities.

Steps to Convert to H2 Database

Update Dependencies

1. **Add H2 Database Dependency**:
   - Open the `pom.xml` file.
   - Add the following dependency for the H2 database:

```xml
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

2. **Add Spring Data JPA Dependency**:
   - Ensure that the Spring Data JPA dependency is added:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Configure Database

1. **Update `application.properties`**:
   - In the `src/main/resources` directory, open or create the `application.properties` file.
   - Add the following properties to configure the H2 database:

```properties
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
```

Database Initialization

1. **Create `schema.sql` and `data.sql` Files**:
   - In the `src/main/resources` directory, create a file named `schema.sql` to define the database schema.
   - Create another file named `data.sql` to populate the database with initial data.

`schema.sql`:

```sql
CREATE TABLE customer (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

`data.sql`:

```sql
INSERT INTO customer (name) VALUES ('John Doe');
INSERT INTO customer (name) VALUES ('Jane Adams');
INSERT INTO customer (name) VALUES ('Alice Jones');
```

Update Code

1. **Update the Customer Entity**:
   - In the `Customer` class, ensure it is annotated with `@Entity`.
2. **Create the Customer Repository**:
   - In the `com.example.demo` package, create a new Java interface named `CustomerRepository`.
   - Extend `JpaRepository<Customer, Long>` to provide CRUD operations for the `Customer` entity.
3. **Update the Customer Service**:
   - Inject the `CustomerRepository` into the `CustomerService`.
   - Update the methods to use the repository for database operations instead of the in-memory list.
4. **Update the Customer Controller**:
   - Ensure the `CustomerController` uses the updated `CustomerService` methods.

With these updates, your application will now use an H2 database to persist customer data instead of an in-memory list.

6. Creating JUnit Tests with Mockito

Introduction

To ensure the functionality of your application, you should write unit tests. You can use JUnit for writing tests and Mockito for mocking dependencies. This section will guide you through writing JUnit tests for the `CustomerService` and `CustomerController` classes, using Mockito to mock the database queries.

Steps to Create JUnit Tests with Mockito

Update Dependencies

1. **Add JUnit and Mockito Dependencies**:
   - Open the `pom.xml` file.
   - Add the following dependencies for JUnit and Mockito:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
```

Create Unit Tests

1. **Create Unit Tests for `CustomerService`**:
   - In the `src/test/java/com/example/demo` directory, create a new Java class named `CustomerServiceTest`.
   - Annotate the class with `@ExtendWith(MockitoExtension.class)`.
   - Mock the `CustomerRepository` and inject it into the `CustomerService`.
   - Write test methods for each service method, using Mockito to define the behavior of the mocked repository.
     - `testGetAllCustomers`: Mock the `findAll()` method of `CustomerRepository` to return a list of customers. Verify that `getAllCustomers()` calls `findAll()` and returns the correct list.
     - `testGetCustomerById`: Mock the `findById()` method of `CustomerRepository` to return an `Optional` of a customer. Verify that `getCustomerById()` calls `findById()` and returns the correct customer.

- `testCreateCustomer`: Mock the `save()` method of `CustomerRepository` to return a customer. Verify that `createCustomer()` calls `save()` and returns the correct customer.
- `testUpdateCustomer`: Mock the `findById()` and `save()` methods of `CustomerRepository` to return a customer. Verify that `updateCustomer()` calls `findById()` and `save()`, and returns the correct customer.
- `testDeleteCustomer`: Mock the `findById()` and `deleteById()` methods of `CustomerRepository`. Verify that `deleteCustomer()` calls `findById()` and `deleteById()`, and returns the correct result.

2. Create Unit Tests for `CustomerController`:
   - In the `src/test/java/com/example/demo` directory, create a new Java class named `CustomerControllerTest`.
   - Annotate the class with `@ExtendWith(MockitoExtension.class)`.
   - Mock the `CustomerService` and inject it into the `CustomerController`.
   - Write test methods for each controller method, using Mockito to define the behavior of the mocked service.
     - `testGetAllCustomers`: Mock the `getAllCustomers()` method of `CustomerService` to return a list of customers. Verify that `getAllCustomers()` calls `getAllCustomers()` and returns the correct list.
     - `testGetCustomerById`: Mock the `getCustomerById()` method of `CustomerService` to return an `Optional` of a customer. Verify that `getCustomerById()` calls `getCustomerById()` and returns the correct customer.
     - `testCreateCustomer`: Mock the `createCustomer()` method of `CustomerService` to return a customer. Verify that `createCustomer()` calls `createCustomer()` and returns the correct customer.
     - `testUpdateCustomer`: Mock the `updateCustomer()` method of `CustomerService` to return an `Optional` of a customer. Verify that `updateCustomer()` calls `updateCustomer()` and returns the correct customer.
     - `testDeleteCustomer`: Mock the `deleteCustomer()` method of `CustomerService` to return true. Verify that `deleteCustomer()` calls `deleteCustomer()` and returns the correct result.

With these unit tests, you can ensure that your `CustomerService` and `CustomerController` classes are functioning correctly and that the database interactions are properly mocked using Mockito.