**Coding Assessment: Bank Customer Management API**

**Problem Statement**

You are required to build a Spring Boot RESTful API for managing customers and their associated bank accounts. Each customer can have multiple bank accounts. The API should support CRUD operations, search functionality, and follow best practices including validation and global exception handling. The application must use an in-memory H2 database.

**Submit screenshots of all the Rest endpoints tested in Postman.** You can insert them at the end of this document.

**Important:** Your project must build and compile else the test cases on which you are being scored will not run.

**Domain Model**

**Customer**

id (auto-generated)

firstName (required, max 50 characters)

lastName

email

List<Account> accounts(One-to-Many)

**Account**

id (auto-generated)

accountNumber

accountType (e.g., SAVINGS, CHECKING)

balance

customer (Many-to-One)

**Repository Tier**

**The Repository interface may extend JpaRepository.**

### ****CustomerService Method Signatures****

| **Method Signature** | **Description** |
| --- | --- |
| Customer saveCustomer(Customer customer) | Create or update a customer |
| List<Customer> getAllCustomers() | Retrieve all customers |
| Optional<Customer> getCustomerById(Integer id) | Get customer by ID |
| void deleteCustomer(Integer id) | Delete customer by ID |
| List<Customer> search(String firstName, String lastName) | Search customers by first/last name |
|  |  |
|  |  |
|  |  |

### ****AccountService Method Signatures****

| **Method Signature** | **Description** |
| --- | --- |
| List<Account> getAccountsByCustomerId(Integer customerId) | Retrieve all accounts belonging to a customer |

**API Endpoints**

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | /customers | Create a new customer |
| GET | /customers | Retrieve all customers |
| PUT | /customers | Update an existing customer |
| DELETE | /customers/{id} | Delete a customer by ID |
| GET | /customers/{id} | Get a customer by ID. **Note:** Use customerService.getCustomerById(id) which returns an Optional<Customer>. Apply .map(ResponseEntity::ok) to transform the Customer object into a 200 OK response if present. Use .orElseThrow(() -> new ResourceNotFoundException(...)) as a lambda to throw a custom exception if the customer isn't found. |
| GET | /customers/search | Search by firstName and/or lastName |
| GET | /customers/{id}/account | Add an endpoint to retrieve all accounts for a specific customer **Note: Path Variable Usage:** The @PathVariable Integer customerId extracts the ID from the URL (e.g., /customers/1/account), so make sure your endpoint URL matches this structure.  **Service Call:** accountService.getAccountsByCustomerId(customerId) fetches a list of accounts associated with that customer. Ensure the service method handles non-existing customer IDs gracefully (e.g., returns an empty list).  **Null-Safety with Ternary:** The line accounts != null ? accounts : List.of() ensures a non-null list is returned to avoid NullPointerException.  **ResponseEntity Wrapping:** Returning ResponseEntity.ok() is a best practice to allow more flexible HTTP responses (status code, headers, etc.). |

Note: Input validation must be performed on firstName.

**Validation Requirement**

Apply validation annotations to the firstName field in the Customer entity:

Must not be blank

message = "First name is mandatory"

Maximum 50 characters

message = "First name must be at most 50 characters"

**Exception Handling**

You are required to implement a **global exception handler** for your REST API. This centralizes error handling across all controllers and ensures standardized error responses.

#### **Validation Error — 400 Bad Request**

If the client sends a request with invalid data (e.g., missing or blank firstName), your API must return a 400 Bad Request.

Use @Valid in your controller method parameters and annotate fields in the model class.

The error response should clearly indicate the specific validation failures in a field-wise format.

This exception is thrown automatically by Spring when a @Valid annotated object fails validation in a controller method (e.g., @RequestBody Customer customer).

Use ex.getBindingResult().getFieldErrors() to get a list of all field- level validation errors (like blank firstName, or exceeded max length).

Each FieldError contains:

getField() → name of the field with the issue.

getDefaultMessage() → human-readable validation error message from the annotation.

A Map<String, String> lets you neatly organize errors as field -> message pairs, which is easy for frontend clients to parse and display.

**Response Structure**

Return ResponseEntity with HttpStatus.BAD\_REQUEST (400).

Keep the error messages clear and user-friendly.

Make sure your controller method uses @Valid on the input object, or else the exception will not be triggered.

**Example Response:**

**{**

**"firstName": "First name is mandatory"**

**}**

#### Resource Not Found — 404 Not Found

When a customer with a given ID is not found, throw a custom ResourceNotFoundException.

This exception should be handled separately to return a 404 Not Found response with a clear message.

**Example Response:**

#### Resource Not Found — 404 Not Found

When a customer with a given ID is not found, throw a custom ResourceNotFoundException.

This exception should be handled separately to return a 404 Not Found response with a clear message.

**Example Response:**

**{**

**"message": "Internal server error"**

**}**

**Database Configuration** (application.properties)

spring.datasource.url=jdbc:h2:mem:bankdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

**Project Setup Instructions**

1. Visit <https://start.spring.io>
2. Select the following settings:

* **Project**: Maven
* **Language**: Java
* **Spring Boot**: (use the latest stable version)
* **Group**: com.example
* **Artifact**: bank-customer-api
* **Package name: com.example.bank**
* **Packaging**: Jar
* **Java**: 17 (or 21, 24 according to you local JDK installation)

1. Add the following dependencies:

* Spring Web
* Spring Data JPA
* H2 Database
* Spring Boot DevTools
* Validation (Jakarta Bean Validation)

1. Generate the project and import it into your IDE (IntelliJ, Eclipse, etc.)

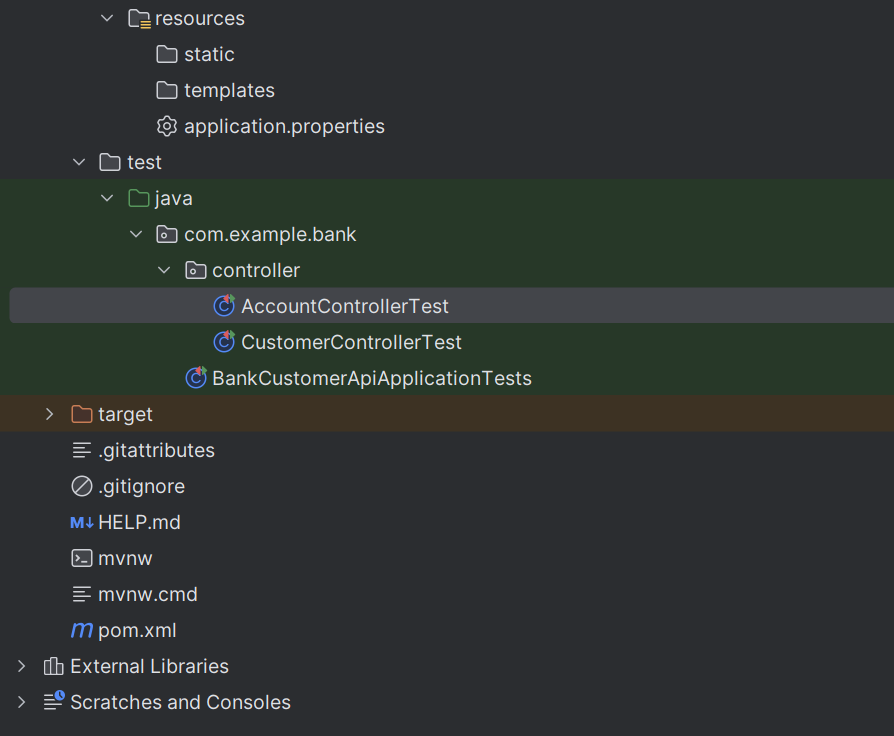
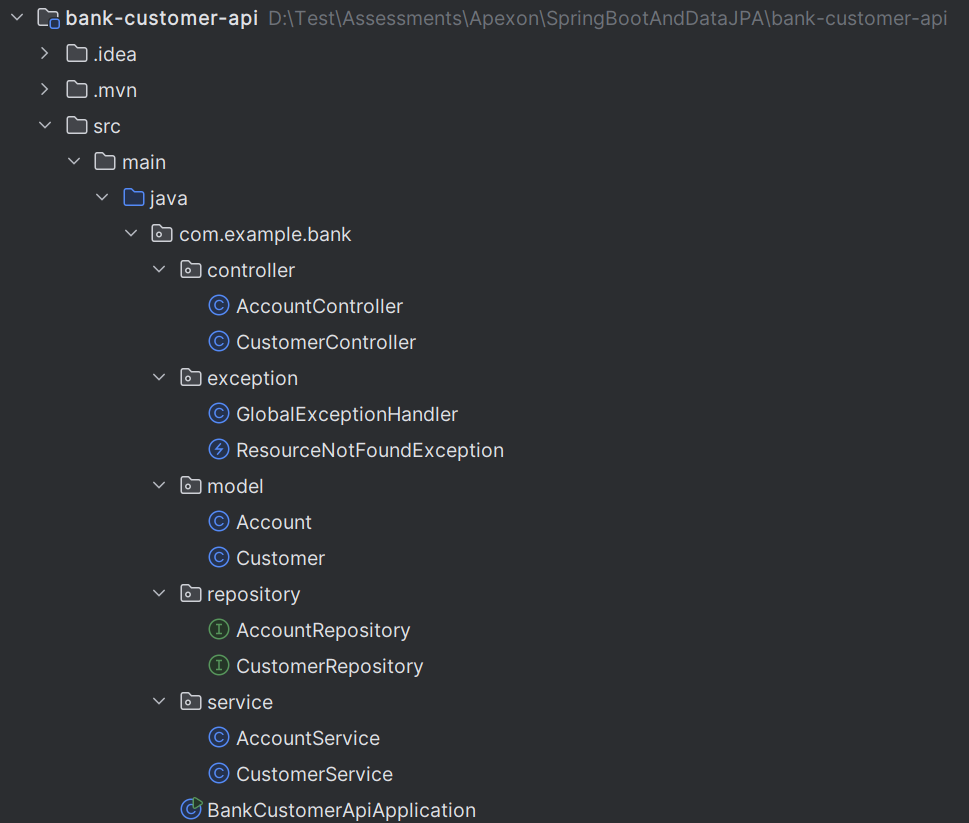
**Deliverables**

A fully working Spring Boot application

Domain models, controllers, and repository interfaces

H2 database setup and schema auto-generation

Exception handling and input validation

Here's a clean and standard folder structure for your **Spring Boot Bank Customer API Project**:

**Postman Screenshots**

|  |  |  |
| --- | --- | --- |
| GET | /customers | Retrieve all customers |

<insert screenshot>

|  |  |  |
| --- | --- | --- |
| PUT | /customers | Update an existing customer |

<insert screenshot>

|  |  |  |
| --- | --- | --- |
| DELETE | /customers/{id} | Delete a customer by ID |

<insert screenshot>

|  |  |  |
| --- | --- | --- |
| GET | /customers/{id} | Get a customer by ID |

<insert screenshot>

|  |  |  |
| --- | --- | --- |
| GET | /customers/search | Search by firstName and/or lastName |

<insert screenshot>

|  |  |  |
| --- | --- | --- |
| GET | /customers/{id}/account | Add an endpoint to retrieve all accounts for a specific customer |

<insert screenshot>

We have defined 10 test scenarios, each worth 10 points for a total of 100. These tests will cover core functionality, validation, and exception handling.

Here’s a breakdown of the 10 test cases (you’ll get the code for each too):

### Test Scenarios (10 points each)

### ****Controller Test Case Summary****

| **#** | **Test Case Description** | **Expected Outcome** |
| --- | --- | --- |
| 1 | Create a new customer (valid input) | 200 OK with JSON response containing the created customer |
| 2 | Retrieve all customers | 200 OK with a list of all customers |
| 3 | Get customer by valid ID | 200 OK with customer JSON |
| 4 | Return 404 for customer not found by ID | 404 Not Found with meaningful error message |
| 5 | Update an existing customer | 200 OK with updated customer data |
| 6 | Delete a customer by ID | 200 OK or 204 No Content if deletion successful |
| 7 | Search customers by firstName and/or lastName | 200 OK with filtered list |
| 8 | Retrieve accounts of a customer | 200 OK with customer’s account list |
| 9 | Validation error when firstName is blank | 400 Bad Request with validation error message |
| 10 | Global exception handler returns 500 on unexpected error | 500 Internal Server Error with generic error response |

### Final Checklist

**Entities**: Customer and Account are created with proper JPA annotations and validation.

**Repositories**: CustomerRepository and AccountRepository extend JpaRepository.

**Services**: Business logic is implemented in CustomerService and AccountService.

**Controllers**: All required REST endpoints are implemented in CustomerController and AccountController.

**Global Exception Handler**: @ControllerAdvice is set up to handle validation and general errors.

application.properties is configured for in-memory H2 DB.

**Spring Boot App Class** (BankCustomerApiApplication) has @SpringBootApplication.

#### **Run the Application**

mvn spring-boot:run

Or

./mvnw spring-boot:run

Or use your IDE (IntelliJ, Eclipse) to run BankCustomerApiApplication.java.

**Access H2 console at:** http://localhost:8080/h2-console  
JDBC URL: jdbc:h2:mem:bankdb

### **Test the API**

Use Postman or any REST client. Sample body for POST /customers:

{

“firstName”: “John”,

“lastName”: “Doe”,

“email”: ["john.doe@example.com](mailto:\"john.doe@example.com\")”

}

Here’s complete CustomerControllerTest and AccountControllerTest classes using **JUnit 5**, **Spring Boot Test**, and **MockMvc**, covering all 10 of your test cases:

**CustomerControllerTest.java**

package com.example.bank.controller;  
  
import com.example.bank.model.Customer;  
import com.example.bank.service.CustomerService;  
import com.example.bank.service.AccountService;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.http.MediaType;  
import org.springframework.test.web.servlet.MockMvc;  
  
import java.util.List;  
import java.util.Optional;  
  
import static org.mockito.ArgumentMatchers.\*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  
  
@WebMvcTest(CustomerController.class)  
class CustomerControllerTest {  
  
 @Autowired  
 private MockMvc mockMvc;  
  
 @MockBean  
 private CustomerService customerService;  
  
 @MockBean  
 private AccountService accountService;  
  
 private final ObjectMapper mapper = new ObjectMapper();  
  
 @Test  
 void createCustomer\_validInput\_returnsCreated() throws Exception {  
 Customer customer = new Customer(1, "John", "Doe", "john@example.com", null);  
 Mockito.*when*(customerService.saveCustomer(*any*(Customer.class))).thenReturn(customer);  
  
 mockMvc.perform(*post*("/customers")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content(mapper.writeValueAsString(customer)))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.id").value(1));  
 }  
  
 @Test  
 void getAllCustomers\_returnsList() throws Exception {  
 Mockito.*when*(customerService.getAllCustomers()).thenReturn(List.*of*(new Customer()));  
  
 mockMvc.perform(*get*("/customers"))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$").isArray());  
 }  
  
 @Test  
 void getCustomerById\_validId\_returnsCustomer() throws Exception {  
 Mockito.*when*(customerService.getCustomerById(1)).thenReturn(Optional.*of*(new Customer(1, "Jane", "Doe", "jane@example.com", null)));  
  
 mockMvc.perform(*get*("/customers/1"))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.firstName").value("Jane"));  
 }  
  
 @Test  
 void getCustomerById\_notFound\_returns404() throws Exception {  
 Mockito.*when*(customerService.getCustomerById(999)).thenReturn(Optional.*empty*());  
  
 mockMvc.perform(*get*("/customers/999"))  
 .andExpect(*status*().isNotFound());  
 }  
  
 @Test  
 void updateCustomer\_validInput\_returnsUpdated() throws Exception {  
 Customer customer = new Customer(1, "Updated", "User", "updated@example.com", null);  
 Mockito.*when*(customerService.saveCustomer(*any*(Customer.class))).thenReturn(customer);  
  
 mockMvc.perform(*put*("/customers")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content(mapper.writeValueAsString(customer)))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.firstName").value("Updated"));  
 }  
  
 @Test  
 void deleteCustomer\_validId\_returns200() throws Exception {  
 mockMvc.perform(*delete*("/customers/1"))  
 .andExpect(*status*().isOk());  
 }  
  
 @Test  
 void searchCustomer\_byFirstName\_returnsMatch() throws Exception {  
 Mockito.*when*(customerService.search(*eq*("Alice"), *any*()))  
 .thenReturn(List.*of*(new Customer(1, "Alice", "Smith", "alice@example.com", null)));  
  
 mockMvc.perform(*get*("/customers/search?firstName=Alice"))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$[0].firstName").value("Alice"));  
 }  
  
 @Test  
 void createCustomer\_blankFirstName\_returns400() throws Exception {  
 Customer customer = new Customer(null, "", "Doe", "email@example.com", null);  
  
 mockMvc.perform(*post*("/customers")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content(mapper.writeValueAsString(customer)))  
 .andExpect(*status*().isBadRequest())  
 .andExpect(*jsonPath*("$.firstName").value("First name is mandatory"));  
 }  
  
  
 @Test  
 void handleUnhandledException\_returns500() throws Exception {  
 Mockito.*when*(customerService.getAllCustomers()).thenThrow(new RuntimeException("Unexpected"));  
  
 mockMvc.perform(*get*("/customers"))  
 .andExpect(*status*().isInternalServerError())  
 .andExpect(*jsonPath*("$.message").value("Internal server error"));  
 }  
}

**AccountControllerTest.java**

package com.example.bank.controller;  
  
import com.example.bank.service.AccountService;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.test.web.servlet.MockMvc;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.http.MediaType;  
  
import java.util.List;  
  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  
  
@WebMvcTest(AccountController.class)  
class AccountControllerTest {  
  
 @Autowired  
 private MockMvc mockMvc;  
  
 @MockBean  
 private AccountService accountService;  
  
 @Test  
 void getAccountsForCustomer\_returnsEmptyList() throws Exception {  
 Mockito.*when*(accountService.getAccountsByCustomerId(1)).thenReturn(List.*of*());  
  
 mockMvc.perform(*get*("/customers/1/account"))  
 .andExpect(*status*().isOk())  
 .andExpect(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*jsonPath*("$").isArray())  
 .andExpect(*jsonPath*("$.length()").value(0));  
 }  
}

**Best Of Luck!**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*