Test documentation for CustomerController

testRegisterCustomer

- **Rationale:** Tests the core functionality of adding a new customer to the system.
- **Inputs:** A new customer with standard attributes (ID, username, email).
- **Expected Results:** Customer successfully stored, method call tracked, and correct data returned.

testUpdateCustomer

- **Rationale:** Tests the ability to modify existing customer information.
- **Inputs:** First a new customer, then an updated version with modified email.
- **Expected Results:** Verification that update was called, data changed, and retrieving the updated record shows changes.

testGetCustomer

- **Rationale:** Tests retrieval of an existing customer.
- **Inputs:** Customer ID of a previously registered customer.
- **Expected Results:** Correct customer retrieved and method call tracked.

testGetCustomerNotFound

- **Rationale:** Tests the edge case of requesting a non-existent customer.
- **Inputs:** Invalid customer ID.
- **Expected Results:** Null return value and method call tracked.

Sufficiency of Test Coverage

The test cases are sufficient because they cover:

- **Complete CRUD operations:**
    - Create (register)
    - Read (get)
    - Update
    - (Delete is implemented in the mock but not explicitly tested in the controller)
- **Happy path and edge cases:**
    - Success scenarios (registering, updating, getting valid customers)
    - Error scenarios (getting non-existent customers)
- **Boundary of responsibility:**
    - Tests verify that the controller appropriately delegates to the database layer.
    - Tests confirm controller returns appropriate data structures.
- **Verification of behavior:**

- Each test validates both the functionality and the interaction pattern with dependencies.

CustomerControllerTest

```
package hi.verkefni.vidmot;

import hi.verkefni.vinnsla.Customer;
import hi.verkefni.vinnsla.MockCustomerDB;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class CustomerControllerTest {
    private MockCustomerDB mockDB;
    private CustomerController controller;

    @BeforeEach
    public void setUp() {
        mockDB = new MockCustomerDB();
        controller = new CustomerController(mockDB);
    }

    @Test
    public void testRegisterCustomer() {
        Customer customer = new Customer("C001", "johndoe", "john@example.com");

        Customer result = controller.registerCustomer(customer);

        assertEquals(1, mockDB.getInsertCalls(), "Insert method should be called
once");
        assertEquals(1, mockDB.getCustomerCount(), "One customer should be in the
mock database");
        assertEquals("johndoe", result.getUsername(), "Username should match");
        assertEquals("john@example.com", result.getEmail(), "Email should match");
    }

    @Test
    public void testUpdateCustomer() {
        Customer customer = new Customer("C001", "johndoe", "john@example.com");
        controller.registerCustomer(customer);

        customer.setEmail("john.doe@updated.com");
        Customer updatedCustomer = controller.updateCustomer(customer);

        assertEquals(1, mockDB.getUpdateCalls(), "Update method should be called
once");
        assertEquals("john.doe@updated.com", updatedCustomer.getEmail(), "Email
should be updated");

        Customer retrievedCustomer = controller.getCustomer("C001");
        assertEquals("john.doe@updated.com", retrievedCustomer.getEmail(),
```

```
"Retrieved customer should have updated email");
    }

    @Test
    public void testGetCustomer() {
        Customer customer = new Customer("C001", "johndoe", "john@example.com");
        controller.registerCustomer(customer);

        mockDB.reset();
        mockDB.insert(customer);

        Customer result = controller.getCustomer("C001");

        assertEquals(1, mockDB.getSelectCalls(), "Select method should be called
once");
        assertNotNull(result, "Customer should be found");
        assertEquals("johndoe", result.getUsername(), "Username should match");
    }

    @Test
    public void testGetCustomerNotFound() {
        assertNull(controller.getCustomer("NONEXISTENT"), "Should return null for
non-existent customer");
        assertEquals(1, mockDB.getSelectCalls(), "Select method should be called
once");
    }
}
```

How the Mock Object Simulates Real Behavior

The MockCustomerDB simulates a real database by:

- **Implementing the same interface:**
  - Extends the CustomerDB class.
  - Overrides all the required methods (selectById, insert, update, delete).
- **In-memory data storage:**
  - Uses a `HashMap<String, Customer>` to store and retrieve customer data.
  - Provides similar persistence behavior during a test execution.
- **Operation tracking:**
  - Counts method calls (insertCalls, updateCalls, etc.).
  - Allows tests to verify that controller methods call the expected database operations.
- **Predictable behavior:**
  - Returns consistent results based on stored data.
  - Doesn't depend on external systems that might change or be unavailable.
- **Reset capabilities:**

- Provides a `reset()` method to clear state between tests.
- Ensures test isolation by removing interference between test cases.

MockCustomerDB

```
package hi.verkefni.vinnsla;

import java.util.HashMap;
import java.util.Map;

/**
 * # Mock Object Documentation: MockCustomerDB
 *
 * ## Purpose
 * The MockCustomerDB class serves as a test double that simulates a database
component without requiring an actual database connection.
 * It implements the same interface as the real CustomerDB but stores data in
memory and tracks method invocations for verification in tests.
 *
 * ## Implementation Details
 * - Extends CustomerDB to maintain the same interface
 * - Uses a HashMap to store customer data in memory
 * - Maintains counters to track how many times each database operation is called
 * - Provides additional methods for test verification (getInsertCalls, reset,
etc.)
 *
 * ## Usage in Test Fixture
 * The MockCustomerDB is initialized in the @BeforeEach method of the test fixture
and injected into the CustomerController,
 * allowing tests to verify both the behavior of the controller and its
interactions with the database layer.
 */
public class MockCustomerDB extends CustomerDB {
    private Map<String, Customer> customers = new HashMap<>();

    // Track method calls for verification in tests
    private int insertCalls = 0;
    private int updateCalls = 0;
    private int selectCalls = 0;
    private int deleteCalls = 0;

    @Override
    public Customer selectById(String customerId) {
        selectCalls++;
        return customers.get(customerId);
    }

    @Override
    public void insert(Customer customer) {
        insertCalls++;
        customers.put(customer.getCustomerId(), customer);
    }
```

```
    @Override
    public void update(Customer customer) {
        updateCalls++;
        customers.put(customer.getCustomerId(), customer);
    }

    @Override
    public void delete(String customerId) {
        deleteCalls++;
        customers.remove(customerId);
    }

    // Methods to help with test verification
    public int getInsertCalls() {
        return insertCalls;
    }

    public int getUpdateCalls() {
        return updateCalls;
    }

    public int getSelectCalls() {
        return selectCalls;
    }

    public int getDeleteCalls() {
        return deleteCalls;
    }

    public void reset() {
        customers.clear();
        insertCalls = 0;
        updateCalls = 0;
        selectCalls = 0;
        deleteCalls = 0;
    }

    public int getCustomerCount() {
        return customers.size();
    }
}
```

Customer Model

```
package hi.verkefni.vinnsla;

public class Customer {
    private String customerId;
    private String username;
    private String email;
```

```
    public Customer(String customerId, String username, String email) {
        this.customerId = customerId;
        this.username = username;
        this.email = email;
    }

    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```