Test Documentation for FlightController

Hópur 7F

- **Mikael Sigurður Kristinsson** (msk14@hi.is)
- **Anton Benediktsson** (anb59@hi.is)
- **Valur Ingi Sigurðarson** (vis45@hi.is)
- **Benedikt Arnar Davíðsson** (bad9@hi.is)

Table of Contents

Test Fixture Implementation

```
package hi.verkefni.vidmot;

import hi.verkefni.vinnsla.Flight;
import hi.verkefni.vinnsla.MockFlightDB;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.List;

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

public class FlightControllerTest {
    private MockFlightDB mockDB;
    private FlightController controller;
    private DateTimeFormatter formatter;

    @BeforeEach
    public void setUp() {
        // Initialize the mock database and controller before each test
        mockDB = new MockFlightDB();
        controller = new FlightController(mockDB);
        formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    }

    @AfterEach
    public void tearDown() {
        // Clean up after each test to prevent state bleeding between tests
        mockDB.reset();
        mockDB = null;
        controller = null;
    }

    @Test
    public void testGetFlightByNumber() {
        // Act - Access the controller method being tested
        Flight flight = controller.getFlightByNumber("FI101");

        // Assert - Verify expected behavior
        assertNotNull(flight, "Flight should be found");
        assertEquals("FI101", flight.getFlightNumber(), "Flight number should
match");
    }

    @Test
    public void testGetFlightByNumberNotFound() {
```

```
        // Act - Try to retrieve a non-existent flight
        Flight flight = controller.getFlightByNumber("NONEXISTENT");

        // Assert - Verify it returns null as expected
        assertNull(flight, "Flight should not be found");
    }

    @Test
    public void testSearchFlights() {
        // Arrange - Set up search parameters
        LocalDateTime searchDate = LocalDateTime.parse("2020-11-01 00:00:00",
formatter);

        // Act - Call the search method with all parameters
        List<Flight> flights = controller.searchFlights("KEF", "JFK",
searchDate);

        // Assert - Verify correct results are returned
        assertEquals(1, flights.size(), "One flight should be found");
        assertEquals("FI101", flights.get(0).getFlightNumber(), "Should find
the correct flight");
    }

    @Test
    public void testSearchFlightsNoResults() {
        // Arrange - Set up search parameters that won't match any flights
        LocalDateTime searchDate = LocalDateTime.parse("2020-11-10 00:00:00",
formatter);

        // Act - Search with parameters that don't match any flights
        List<Flight> flights = controller.searchFlights("KEF", "JFK",
searchDate);

        // Assert - Verify empty results
        assertTrue(flights.isEmpty(), "No flights should be found");
    }

    @Test
    public void testSearchFlightsByOriginOnly() {
        // Act - Search with only origin specified
        List<Flight> flights = controller.searchFlights("KEF", null, null);

        // Assert - Verify correct filtering
        assertEquals(1, flights.size(), "One flight should be found");
        assertEquals("FI101", flights.get(0).getFlightNumber(), "Should find
flight from KEF");
    }

    @Test
    public void testSearchFlightsByDestinationOnly() {
        // Act - Search with only destination specified
        List<Flight> flights = controller.searchFlights(null, "KEF", null);
```

```
        // Assert - Verify correct filtering
        assertEquals(1, flights.size(), "One flight should be found");
        assertEquals("FI102", flights.get(0).getFlightNumber(), "Should find
flight to KEF");
    }

    @Test
    public void testGetAllFlights() {
        // Act - Get all flights from the controller
        List<Flight> flights = controller.getAllFlights();

        // Assert - Verify all test flights are returned
        assertEquals(2, flights.size(), "Two flights should be found");
        assertTrue(
            flights.stream().map(Flight::getFlightNumber).anyMatch(num ->
num.equals("FI101")) &&
            flights.stream().map(Flight::getFlightNumber).anyMatch(num ->
num.equals("FI102")),
            "Both test flights should be in the results"
        );
    }

    @Test
    public void testAddFlightAndRetrieve() {
        // Arrange - Create a new flight
        Flight newFlight = new Flight(
            "FI103",
            "KEF",
            "CPH",
            LocalDateTime.parse("2020-11-03 10:00:00", formatter),
            LocalDateTime.parse("2020-11-03 13:30:00", formatter)
        );
        mockDB.addFlight(newFlight);

        // Act - Try to retrieve the newly added flight
        Flight retrievedFlight = controller.getFlightByNumber("FI103");

        // Assert - Verify it was stored and retrieved correctly
        assertNotNull(retrievedFlight, "Added flight should be retrievable");
        assertEquals("CPH", retrievedFlight.getDestination(), "Destination
should be preserved");
    }
}
```

Key Components of the Test Fixture

1. **Setup Method**: The `@BeforeEach` annotated `setUp()` method establishes a clean test environment before each test by:
    - Creating a fresh instance of the mock database
    - Initializing the controller with the mock database
    - Setting up a date formatter for test data creation
2. **Teardown Method**: The `@AfterEach` annotated `tearDown()` method cleans up resources after each test by:
    - Resetting the mock database to clear any changes
    - Setting references to null to aid garbage collection
3. **Test Methods**: Each test focuses on a specific aspect of the controller's functionality:
    - `testGetFlightByNumber`: Tests retrieving a flight by its unique identifier
    - `testGetFlightByNumberNotFound`: Tests handling of non-existent flight lookups
    - `testSearchFlights`: Tests searching with multiple criteria
    - `testSearchFlightsByOriginOnly`, `testSearchFlightsByDestinationOnly`: Test partial search scenarios
    - `testAddFlightAndRetrieve`: Tests the full cycle of adding and retrieving a flight
4. **Testing Patterns**:
    - Arrange-Act-Assert pattern for test structure
    - Explicit failure messages in assertions
    - Test isolation through setup and teardown

Mock Object Implementation

To support testing the `FlightController` without relying on an actual database, a mock implementation of the `FlightDB` class was created. This mock simulates database operations in memory.

```
package hi.verkefni.vinnsla;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * Mock implementation of FlightDB for testing purposes.
 * This class simulates flight database operations without requiring an
actual database.
 */
public class MockFlightDB extends FlightDB {
    private Map<String, Flight> flights = new HashMap<>();

    /**
     * Constructs a MockFlightDB pre-populated with two test flights:
     * - FI101: KEF to JFK on 2020-11-01
     * - FI102: JFK to KEF on 2020-11-02
     */
    public MockFlightDB() {
        // Initialize with some test flights
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");

        addFlight(new Flight(
            "FI101",
            "KEF",
            "JFK",
            LocalDateTime.parse("2020-11-01 12:00:00", formatter),
            LocalDateTime.parse("2020-11-01 18:00:00", formatter)
        ));

        addFlight(new Flight(
            "FI102",
            "JFK",
            "KEF",
            LocalDateTime.parse("2020-11-02 12:00:00", formatter),
            LocalDateTime.parse("2020-11-02 18:00:00", formatter)
        ));
    }
```

```
    /**
     * Add a flight to the mock database.
     * This method has no equivalent in the real FlightDB as it directly
     * manipulates the in-memory collection.
     *
     * @param flight The flight to add
     */
    public void addFlight(Flight flight) {
        flights.put(flight.getFlightNumber(), flight);
    }

    /**
     * Retrieve a flight by its flight number.
     *
     * @param flightNumber The flight number to look up
     * @return The flight if found, null otherwise
     */
    @Override
    public Flight selectByFlightNumber(String flightNumber) {
        return flights.get(flightNumber);
    }

    /**
     * Search for flights based on origin, destination, and date from the
mock database.
     * Unlike the real database implementation, this mock performs simple in-
memory filtering
     * and only matches exact dates (not time ranges).
     *
     * @param origin The origin airport code (can be null)
     * @param destination The destination airport code (can be null)
     * @param date The departure date (can be null)
     * @return List of matching flights
     */
    @Override
    public List<Flight> searchFlights(String origin, String destination,
LocalDateTime date) {
        return flights.values().stream()
            .filter(f -> (origin == null || f.getOrigin().equals(origin)) &&
                   (destination == null ||
f.getDestination().equals(destination)) &&
                   (date == null ||
f.getDepartureTime().toLocalDate().equals(date.toLocalDate())))
            .collect(Collectors.toList());
    }

    /**
     * Get all flights in the mock database.
     *
     * @return List of all flights
     */
    @Override
```

```
    public List<Flight> getAllFlights() {
        return new ArrayList<>(flights.values());
    }

    /**
     * Reset the mock database, clearing all flights.
     * This method is specifically for testing purposes to ensure test
isolation.
     */
    public void reset() {
        flights.clear();
    }

    /**
     * Get the number of flights in the mock database.
     * This method has no equivalent in the real FlightDB class.
     *
     * @return The number of flights
     */
    public int getFlightCount() {
        return flights.size();
    }
}
```

Key Components of the Mock Object

1. **In-Memory Storage**: Uses a `HashMap` to store flights in memory, with flight numbers as keys
2. **Pre-populated Test Data**: Constructor initializes the mock with predefined test flights
3. **Overridden Methods**: Implements all required methods from the parent class:
    - `selectByFlightNumber`: Retrieves flights from the in-memory map
    - `searchFlights`: Filters flights based on multiple criteria
    - `getAllFlights`: Returns all flights in the system
4. **Testing Helpers**: Additional methods specific to testing:
    - `addFlight`: Adds flights directly to the in-memory collection
    - `reset`: Clears all data to provide a clean state between tests
    - `getFlightCount`: Returns the number of flights currently stored
5. **Documentation**: Thorough Javadoc comments explain:
    - The purpose of the mock implementation
    - Behavior differences from the real database
    - Testing-specific methods not present in the real implementation
    - Initial state of the mock object

This mock object successfully isolates the FlightController tests from any external dependencies, allowing tests to run quickly and deterministically without requiring database access.