Implementing and Testing Flight Redirection with @Transactional Annotation

This guide provides a step-by-step solution and description for implementing a method to redirect all flights from one airport to another using the `@Transactional` annotation in a Spring Boot application. It also includes writing a test case to verify the functionality.

Overview

We aim to:

1. Implement the `redirectFlights` method in the `FlightService` class.
2. Write a corresponding test case in `FlightServiceTest` to verify that flights are correctly redirected.

Step 1: Implementing the `redirectFlights` Method in `FlightService`

1.0 Create FlightService Class

Create a new method in the `FlightService` class to redirect flights from one airport to another. It should be in the `com.luxoft.flights.services` package.

1.1 Import Necessary Packages

Ensure you have the required imports in your `FlightService` class:

```java
import com.luxoft.flights.model.Airport;
import com.luxoft.flights.model.Flight;
import com.luxoft.flights.repositories.AirportRepository;
import com.luxoft.flights.repositories.FlightRepository;
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Service;
import java.util.List;
```

1.2 Annotate the Class with `@Service`

Annotate the `FlightService` class to make it a Spring service component:

```java
@Service
public class FlightService {
    // Class content...
}
```

1.3 Inject Required Repositories

Use constructor injection to inject `FlightRepository` and `AirportRepository`:

```java
private final FlightRepository flightRepository;
private final AirportRepository airportRepository;

public FlightService(FlightRepository flightRepository,
                     AirportRepository airportRepository) {
    this.flightRepository = flightRepository;
    this.airportRepository = airportRepository;
}
```

1.4 Implement the `redirectFlights` Method

Add the `redirectFlights` method and annotate it with `@Transactional`:

```java
@Transactional
public void redirectFlights(String oldAirportCode, String newAirportCode) {
    // Fetch the new airport entity
    Airport newAirport = airportRepository.findByAirportCode(newAirportCode)
            .orElseThrow(() -> new IllegalArgumentException("New airport not found"));

    // Redirect flights originating from the old airport
    List<Flight> flightsFromOldOrigin = flightRepository.findByOriginAirportCode
(oldAirportCode);
    for (Flight flight : flightsFromOldOrigin) {
        flight.setOrigin(newAirport);
    }
    flightRepository.saveAll(flightsFromOldOrigin);

    // Redirect flights destined for the old airport
    List<Flight> flightsToOldDestination = flightRepository
.findByDestinationAirportCode(oldAirportCode);
    for (Flight flight : flightsToOldDestination) {
        flight.setDestination(newAirport);
    }
    flightRepository.saveAll(flightsToOldDestination);
}
```

Explanation

Transactional Annotation: The `@Transactional` annotation ensures that all operations within the method are executed within a single transaction. If any exception occurs, the entire transaction is rolled back.

Fetching New Airport: We retrieve the `Airport` entity for the `newAirportCode`. If it doesn't exist, we throw an `IllegalArgumentException`.

Redirecting Origin Flights:

- We fetch all flights where the origin airport code matches `oldAirportCode`.
- We update the origin airport of each flight to the new airport.
- We save all updated flights using `saveAll`.

Redirecting Destination Flights:

- Similar to the origin flights, but we update the destination airport.

Step 2: Writing the Test Case in `FlightServiceTest`

2.1 Set Up the Test Class

Annotate the test class with `@SpringBootTest` and specify the application class. Use `@ActiveProfiles` to exclude components not needed during testing.

```
import com.luxoft.flights.FlightsApplication;
// Other imports...

@ActiveProfiles("test")
@SpringBootTest(classes = FlightsApplication.class)
public class FlightServiceTest {
    // Test content...
}
```

2.2 Inject Required Beans

Inject `FlightService`, `FlightRepository`, `AirportRepository`, and `StateRepository`:

```
@Autowired
private FlightService flightService;
```

```
@Autowired
private FlightRepository flightRepository;

@Autowired
private AirportRepository airportRepository;

@Autowired
private StateRepository stateRepository;
```

2.3 Set Up Test Data in `@BeforeEach` Method

Use the `@BeforeEach` annotation to set up the data before each test:

```
@BeforeEach
public void setUp() {
    // Create test states
    State newYork = new State("New York");
    stateRepository.save(newYork);
    State california = new State("California");
    stateRepository.save(california);

    // Create test airports
    Airport jfk = new Airport(1, "JFK", newYork);
    airportRepository.save(jfk);
    Airport lax = new Airport(2, "LAX", california);
    airportRepository.save(lax);

    // Create sample flights
    Flight flight1 = new Flight("AA123", new Date(), jfk, lax, null, 0, 0, 0);
    Flight flight2 = new Flight("AA456", new Date(), lax, jfk, null, 0, 0, 0);

    // Save flights in the repository
    flightRepository.save(flight1);
    flightRepository.save(flight2);
}
```

Explanation

- **Data Cleanup**: We delete all existing data to ensure a clean state for each test.
- **Creating States and Airports**: We create `State` and `Airport` entities for "New York" (`JFK`) and "California" (`LAX`).
- **Creating Flights**: Two flights are created:
- `flight1`: Originating from `JFK` to `LAX`.
- `flight2`: Originating from `LAX` to `JFK`.
- **Saving Data**: We save the states, airports, and flights to the repositories.

2.4 Write the Test Method `testRedirectFlights`

```
@Test
public void testRedirectFlights() {
    // Redirect flights from JFK to LAX
    flightService.redirectFlights("JFK", "LAX");

    // Retrieve all flights and verify redirection
    List<Flight> flights = flightRepository.findAll();
    for (Flight flight : flights) {
        assertNotEquals("JFK", flight.getOrigin().getAirportCode());
        assertNotEquals("JFK", flight.getDestination().getAirportCode());

        // Verify that the origin or destination is now LAX
        if ("LAX".equals(flight.getOrigin().getAirportCode()) || "LAX".equals(flight
.getDestination().getAirportCode())) {
            assertEquals("LAX", flight.getOrigin().getAirportCode());
            assertEquals("LAX", flight.getDestination().getAirportCode());
        }
    }
}
```

Explanation

- **Invoke Redirect Method**: We call `redirectFlights("JFK", "LAX")` to redirect flights from `JFK` to `LAX`.
- **Retrieve and Assert**:
- We fetch all flights from the repository.
- We assert that none of the flights have `JFK` as their origin or destination airport.
- We check that flights have been correctly updated to have `LAX` as their origin or destination where appropriate.

2.5 Additional Assertions (Optional)

You may add more assertions to ensure data integrity:

```
assertEquals(2, flights.size()); // Ensure the number of flights remains the same

// Additional checks for specific flight details
Flight flight1 = flights.get(0);
Flight flight2 = flights.get(1);

// Verify that flights have been updated correctly
assertEquals("LAX", flight1.getOrigin().getAirportCode());
assertEquals("LAX", flight1.getDestination().getAirportCode());
assertEquals("LAX", flight2.getOrigin().getAirportCode());
assertEquals("LAX", flight2.getDestination().getAirportCode());
```

Step 3: Configuring the Application for Testing

3.1 Exclude `CommandLineRunner` During Tests

To prevent the `CommandLineRunner` from executing during tests, use the `@ActiveProfiles` annotation and set up profiles in your application.

- **In `FlightsApplication` or configuration class**:

```
import org.springframework.context.annotation.Profile;

@Bean
@Profile("!test")
public CommandLineRunner run(DataService dataService, DataLoader dataLoader) throws
Exception {
    return args -> {
        // CommandLineRunner logic here
    };
}
```

- **In your test class**:

```
@ActiveProfiles("test")
@SpringBootTest(classes = FlightsApplication.class)
public class FlightServiceTest {
    // Test content...
}
```

3.2 Application Properties for Test Profile (Optional)

Create an `application-test.properties` file in `src/test/resources` to define properties specific to the test profile.

Conclusion

By following these steps, you have successfully implemented the `redirectFlights` method with transactional behavior and written a test case to verify its functionality. This ensures that flight redirections are handled atomically, maintaining data consistency in your application.

Key Takeaways

- **Transactional Integrity**: Using `@Transactional` ensures that multiple database operations either all succeed or all fail, maintaining data integrity.
- **Test Isolation**: Setting up test data in `@BeforeEach` and cleaning up ensures that tests are independent and repeatable.
- **Profiles in Spring Boot**: Utilizing profiles allows you to control bean loading and configuration based on the environment (e.g., development, testing, production).