Example: Defining a REST Controller in the Flight Service

Below is an example of a basic `FlightController` in the Flight Service. It exposes a REST endpoint to retrieve flight details by ID.

```java
// FlightController.java
package com.example.flight.controller;

import com.example.flight.model.Flight;
import com.example.flight.service.FlightService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/flights")
public class FlightController {

    @Autowired
    private FlightService flightService;

    // GET all flights
    @GetMapping
    public List<Flight> getAllFlights() {
        return flightService.getAllFlights();
    }

    // GET flight by ID
    @GetMapping("/{id}")
    public Flight getFlightById(@PathVariable String id) {
        return flightService.getFlightById(id);
    }

    // POST new flight
    @PostMapping
    public void addFlight(@RequestBody Flight flight) {
        flightService.addFlight(flight);
    }
}
```

Explanation:

- The `@RestController` annotation marks this class as a controller where every method returns a domain object instead of a view name.
- `@RequestMapping("/flights")` defines the base URL for all endpoints in this controller.
- Each method corresponds to an HTTP verb and route:
    - `GET /flights`: Retrieve all flights.
    - `GET /flights/{id}`: Retrieve a specific flight by its ID.
    - `POST /flights`: Add a new flight.

Lab Exercise

Objective:

Implement a simple version of the **Booking Service**, which interacts with the **Flight Service** via REST to fetch flight details before creating a booking.

You will:

1. Create a new Spring Boot project for the Booking Service.
2. Define a Booking entity and repository.
3. Use `RestTemplate` to call the Flight Service's `/flights/{id}` endpoint.
4. Expose a REST endpoint to create a booking only if the flight exists.

Prerequisites

Ensure the following are installed:

- Java 17+
- Maven 3.x
- Spring Boot CLI (optional)
- IDE (IntelliJ, Eclipse, or VSCode)

Also, ensure the **Flight Service** is running on port `8081`.

Step-by-Step Instructions

Step 1: Generate the Booking Service Project

Use Spring Initializr or run:

```
spring init \
--groupId=com.example \
--artifactId=booking-service \
--name="BookingService" \
--dependencies=web,data-jpa,h2 \
booking-service
```

Extract and open the project in your IDE.

Step 2: Configure `application.properties`

Edit `src/main/resources/application.properties`:

```
server.port=8082
spring.application.name=booking-service

# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:bookings
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
```

Step 3: Define the Booking Entity

Create `com/example/booking/model/Booking.java`:

```java
package com.example.booking.model;

import jakarta.persistence.*;

@Entity
public class Booking {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String userId;
    private String flightId;
    private String status;

    // Constructors, Getters, and Setters
    public Booking() {}

    public Booking(String userId, String flightId, String status) {
        this.userId = userId;
        this.flightId = flightId;
        this.status = status;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
```

```
    public String getUserId() { return userId; }
    public void setUserId(String userId) { this.userId = userId; }

    public String getFlightId() { return flightId; }
    public void setFlightId(String flightId) { this.flightId = flightId; }

    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }
}
```

Step 4: Create the Booking Repository

Create `com/example/booking/repository/BookingRepository.java`:

```
package com.example.booking.repository;

import com.example.booking.model.Booking;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookingRepository extends JpaRepository<Booking, Long> {}
```

Step 5: Implement the Booking Service

Create `com/example/booking/service/BookingService.java`:

```
package com.example.booking.service;

import com.example.booking.model.Booking;
import com.example.booking.repository.BookingRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class BookingService {

    @Autowired
    private BookingRepository bookingRepository;

    private final RestTemplate restTemplate = new RestTemplate();
```

```
public Booking bookFlight(String userId, String flightId) {
        String flightUrl = "http://localhost:8081/flights/" + flightId;

        try {
            ResponseEntity<String> response =
restTemplate.getForEntity(flightUrl, String.class);

            if (response.getStatusCode().is2xxSuccessful()) {
                Booking booking = new Booking(userId, flightId, "CONFIRMED");
                return bookingRepository.save(booking);
            } else {
                throw new RuntimeException("Flight not found");
            }
        } catch (Exception e) {
            throw new RuntimeException("Error checking flight availability: "
+ e.getMessage());
        }
    }
}
```

Step 6: Create the Booking Controller

Create `com/example/booking/controller/BookingController.java`:

```
package com.example.booking.controller;

import com.example.booking.model.Booking;
import com.example.booking.service.BookingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/bookings")
public class BookingController {

    @Autowired
    private BookingService bookingService;

    @PostMapping
    public Booking createBooking(@RequestParam String userId, @RequestParam
String flightId) {
        return bookingService.bookFlight(userId, flightId);
    }
}
```

Step 7: Enable RestTemplate

In `BookingServiceApplication.java`, enable `RestTemplate` bean:

```
package com.example.bookingservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class BookingServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(BookingServiceApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

☑ Expected Output

Start the Flight Service first on port `8081`. Then start the Booking Service on `8082`.

Send a POST request to:

```
POST http://localhost:8082/bookings?userId=user123&flightId=FL101
```

If the flight ID exists in the Flight Service, you'll receive a response like:

```
{
  "id": 1,
  "userId": "user123",
  "flightId": "FL101",
  "status": "CONFIRMED"
}
```

If the flight doesn't exist, you'll get a runtime exception thrown by the service.

☑ Complete Implementation

The complete source code for the lab can be found below:

📁 Directory Structure

```
src/
├── main/
│   ├── java/
│   │   └── com.example.booking/
│   │       ├── controller/
│   │       │   └── BookingController.java
│   │       ├── model/
│   │       │   └── Booking.java
│   │       ├── repository/
│   │       │   └── BookingRepository.java
│   │       ├── service/
│   │       │   └── BookingService.java
│   │       └── BookingServiceApplication.java
│   └── resources/
│       └── application.properties
```

📁 Files Recap:

- `Booking.java`: JPA entity representing a booking.
- `BookingRepository.java`: Interface for CRUD operations.
- `BookingService.java`: Business logic including integration with Flight Service.
- `BookingController.java`: REST API for creating bookings.
- `BookingServiceApplication.java`: Main class with `main()` method.
- `application.properties`: Configuration file for server, DB, and app name.
