Implementing JPA Annotations in Flight Application

JPA Feature Description

The Java Persistence API (JPA) is a specification that defines how Java objects can be mapped to relational database tables. It provides a standardized approach for object-relational mapping (ORM), allowing developers to manage relational data in applications using Java objects. By using JPA annotations, we can define the mapping between our model classes and the database schema directly within the code, facilitating seamless data persistence and retrieval.

Why Do We Need It in Flight Application?

In the current Flight Application, data is stored in-memory using lists and sets within the `InMemoryDataService` class. This approach limits the application's ability to handle large datasets and doesn't persist data between application restarts. By introducing JPA annotations, we can:

- Persist data in a relational database.
- Leverage the power of SQL databases for querying and managing data.
- Improve scalability and data integrity.
- Utilize Spring Data JPA repositories for simplified data access.

Refactoring Application

We will refactor the model classes to include JPA annotations and configure the application to use a JPA provider like Hibernate. The steps include:

1. **Add Dependencies to `pom.xml`**

Add Spring Data JPA and H2 Database dependencies:

```xml
<dependencies>
<!-- Other dependencies -->

   <!-- Spring Data JPA -->
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>

   <!-- H2 Database -->
   <dependency>
       <groupId>com.h2database</groupId>
       <artifactId>h2</artifactId>
   </dependency>
```

```
</dependencies>
```

This adds the necessary libraries to use JPA and an in-memory H2 database for development and testing purposes.

2. **Add JPA Annotations to Model Classes**

Update the `Airport` class:

```java
package com.luxoft.flights.model;

import jakarta.persistence.*;

@Entity
public class Airport {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int airportID;

   private String airportCode;

   @ManyToOne
   @JoinColumn(name = "state_name")
   private State state;

   public Airport(int airportID, String airportCode, State state) {
       this.airportID = airportID;
       this.airportCode = airportCode;
       this.state = state;
   }

   public Airport() {
   }

   // Getters and setters...

   // equals, hashCode, toString...
}
```

Update the `Carrier` class:

```java
package com.luxoft.flights.model;

import jakarta.persistence.*;

@Entity
public class Carrier {
   @Id
   private String code;
```

```
    private String name;

    public Carrier() {
    }

    public Carrier(String code) {
        this.code = code;
        if (code.equals("AA")) {
            this.name = "America Airlines";
        } else if (code.equals("WN")) {
            this.name = "Southwest Airlines";
        }
    }

    // Getters and setters...

    // equals, hashCode, toString...
}
```

Update the `Flight` class:

```
package com.luxoft.flights.model;

import jakarta.persistence.*;
import java.util.Date;
import java.util.Objects;

@Entity
public class Flight {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String tailNum;
    private Date flightDate;

    @ManyToOne
    private Airport origin;

    @ManyToOne
    private Airport destination;

    @ManyToOne
    private Carrier carrier;

    private int depDelay;
    private int arrDelay;
    private int cancelled;

    public Flight(String tailNum, Date flightDate, Airport origin,
                  Airport destination, Carrier carrier,
                  int depDelay, int arrDelay,
                  int cancelled) {
        this.tailNum = tailNum;
        this.flightDate = flightDate;
        this.origin = origin;
        this.destination = destination;
        this.carrier = carrier;
        this.cancelled = cancelled;
        this.depDelay = depDelay;
        this.arrDelay = arrDelay;
    }

    public Flight() {
    }

    // Getters and setters...

    // equals, hashCode, toString...
}
```

Update the `State` class:

```java
package com.luxoft.flights.model;

import jakarta.persistence.*;

@Entity
public class State {
    @Id
    private String name;

    public State() {
    }

    public State(String name) {
        this.name = name;
    }

    // Getters and setters...

    // equals, hashCode, toString...
}
```

Explanation:

- Annotate each model class with `@Entity` to mark them as JPA entities.
- Use `@Id` to denote the primary key of the entity.

- For generated IDs, use `@GeneratedValue(strategy = GenerationType.IDENTITY)`.
- Establish relationships using `@ManyToOne` and `@JoinColumn` annotations.

3. **Define Relationships Between Entities**
    - **Airport to State**: Many airports belong to one state.

```
@ManyToOne
@JoinColumn(name = "state_name")
private State state;
```

- **Flight to Airport**: A flight has one origin and one destination airport.

```
@ManyToOne
private Airport origin;

@ManyToOne
private Airport destination;
```

- **Flight to Carrier**: Many flights are operated by one carrier.

```
@ManyToOne
private Carrier carrier;
```

4. **Create Repository Interfaces**

Create repository interfaces for each entity to handle CRUD operations.

FlightRepository:

```
package com.luxoft.flights.repositories;

import com.luxoft.flights.model.Flight;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Date;
import java.util.Optional;

@Repository
public interface FlightRepository extends JpaRepository<Flight, Long> {
   Optional<Flight> findByTailNumAndFlightDate(String tailNum, Date flightDate);
}
```

AirportRepository:

```java
package com.luxoft.flights.repositories;

import com.luxoft.flights.model.Airport;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface AirportRepository extends JpaRepository<Airport, Integer> {
   Optional<Airport> findByAirportCode(String airportCode);
}
```

CarrierRepository:

```java
package com.luxoft.flights.repositories;

import com.luxoft.flights.model.Carrier;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CarrierRepository extends JpaRepository<Carrier, String> {
}
```

StateRepository:

```java
package com.luxoft.flights.repositories;

import com.luxoft.flights.model.State;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StateRepository extends JpaRepository<State, String> {
}
```

5. **Implement `DBDataService` to Use Repositories**

Create a new `DBDataService` class that implements the `DataService` interface and uses the repositories for data access.

```java
package com.luxoft.flights.services;

import com.luxoft.flights.model.*;
import com.luxoft.flights.repositories.*;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;
import java.util.Set;

@Service
@Primary
public class DBDataService implements DataService {

   private final AirportRepository airportRepository;
   private final CarrierRepository carrierRepository;
   private final FlightRepository flightRepository;
   private final StateRepository stateRepository;

   public DBDataService(AirportRepository airportRepository,
                        CarrierRepository carrierRepository,
                        FlightRepository flightRepository,
                        StateRepository stateRepository) {
       this.airportRepository = airportRepository;
       this.carrierRepository = carrierRepository;
       this.flightRepository = flightRepository;
       this.stateRepository = stateRepository;
   }

   @Override
   public Flight saveFlight(Flight flight) {
       Optional<Flight> existingFlight = flightRepository.findByTailNumAndFlightDate(
               flight.getTailNum(), flight.getFlightDate());
       return existingFlight.orElseGet(() -> flightRepository.save(flight));
   }

   @Override
   public Airport saveAirport(Airport airport) {
       Optional<Airport> existingAirport = airportRepository.findByAirportCode(
airport.getAirportCode());
       return existingAirport.orElseGet(() -> airportRepository.save(airport));
   }

   @Override
   public Carrier saveCarrier(Carrier carrier) {
       Optional<Carrier> existingCarrier = carrierRepository.findById(carrier.getCode
());
       return existingCarrier.orElseGet(() -> carrierRepository.save(carrier));
   }

   @Override
   public State saveState(State state) {
       Optional<State> existingState = stateRepository.findById(state.getName());
       return existingState.orElseGet(() -> stateRepository.save(state));
   }
```

```
    @Override
    public List<Flight> getFlights() {
        return flightRepository.findAll();
    }

    @Override
    public Set<Airport> getAirports() {
        List<Airport> airportList = airportRepository.findAll();
        return Set.copyOf(airportList);
    }

    @Override
    public Set<Carrier> getCarriers() {
        List<Carrier> carrierList = carrierRepository.findAll();
        return Set.copyOf(carrierList);
    }

    @Override
    public Set<State> getStates() {
        List<State> stateList = stateRepository.findAll();
        return Set.copyOf(stateList);
    }

    @Override
    public List<Flight> getFlightsByOrigin(String airportCode) {
        // Implement method using repository queries
        return List.of();
    }

    @Override
    public List<Flight> getFlightsByCarrier(String carrierCode) {
        // Implement method using repository queries
        return List.of();
    }

    @Override
    public List<Flight> getFlightsByState(String stateName) {
        // Implement method using repository queries
        return List.of();
    }
}
```

Explanation:

- Annotate the class with `@Service` and `@Primary` to indicate it’s the primary implementation of `DataService`.
- Inject the repositories via constructor injection.
- Implement methods to save and retrieve entities using the repositories.

6. **Configure Database Connection**

Add database configuration to `application.properties`:

```
# Use H2 in-memory database
spring.datasource.url=jdbc:h2:mem:flightdb;DB_CLOSE_DELAY=-1
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# H2 Console settings
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Explanation:

- Configures Spring Boot to use the H2 in-memory database.
- Sets Hibernate to automatically update the database schema (`ddl-auto=update`).
- Enables the H2 console for viewing the database in a browser.

7. **Implement `DataService` Interface**

Ensure that the `DataService` interface reflects the methods used in `DBDataService`.

```java
package com.luxoft.flights.services;

import com.luxoft.flights.model.*;

import java.util.List;
import java.util.Set;

public interface DataService {
   Flight saveFlight(Flight flight);

   Airport saveAirport(Airport airport);

   Carrier saveCarrier(Carrier carrier);

   State saveState(State state);

   List<Flight> getFlights();

   Set<Airport> getAirports();
```

```
    Set<Carrier> getCarriers();

    Set<State> getStates();

    List<Flight> getFlightsByOrigin(String airportCode);

    List<Flight> getFlightsByCarrier(String carrierCode);

    List<Flight> getFlightsByState(String stateName);
}
```

8. **Update `DataLoader` Implementations**

Ensure that `CSVDataLoader` or `DBDataLoader` uses the `DBDataService` to save entities, which now persists them to the database.

9. **Test the Application**

Run the application and verify that data is being saved to and retrieved from the H2 database. You can access the H2 console at `http://localhost:8080/h2-console` to inspect the database.

Benefits of Using JPA Annotations

- **Persistent Data Storage**: Data is stored in a database, persisting beyond application restarts.
- **Scalability**: Efficiently handle large datasets with database management systems.
- **Simplified Data Access**: Use Spring Data JPA repositories for CRUD operations without boilerplate code.
- **Maintainability**: Centralized data definitions make it easier to manage and update the data model.
- **Transaction Management**: Spring handles transactions, ensuring data integrity.

Further Advices

- **Use Proper Fetch Types**: Be mindful of `FetchType.LAZY` and `FetchType.EAGER` to optimize performance and prevent `LazyInitializationException`.
- **Handle Relationships Carefully**: Define cascading operations and orphan removal where appropriate.
- **Validate Entities**: Use validation annotations (e.g., `@NotNull`, `@Size`) to ensure data integrity.
- **Exception Handling**: Implement exception handling for database operations to manage `DataAccessException` and others.
- **Database Indexing**: Consider adding indexes to frequently queried columns for performance optimization.
- **Migration Tools**: Use tools like Flyway or Liquibase for database schema migrations in production environments.