📚 Timeout Airline - Complete Code Explanation

This document explains every file in the project, what each part does, and how they all connect together.

🏗️ PROJECT ARCHITECTURE OVERVIEW

```
┌─────────────────────────────────────────────────────────┐
│                 CLIENT (Postman/Browser)                 │
│                  ↓ HTTP Request                         │
├─────────────────────────────────────────────────────────┤
│                 CONTROLLER LAYER                         │
│          (Receives requests, returns responses)          │
│                  ↓                                      │
├─────────────────────────────────────────────────────────┤
│                 SERVICE LAYER                            │
│            (Business logic lives here)                   │
│                  ↓                                      │
├─────────────────────────────────────────────────────────┤
│                 REPOSITORY LAYER                         │
│             (Talks to the database)                     │
│                  ↓                                      │
├─────────────────────────────────────────────────────────┤
│                 DATABASE                                 │
│             (MySQL - stores data)                        │
└─────────────────────────────────────────────────────────┘
```

How a Request Flows Through the System

```
1. You send: POST /api/v1/clients with JSON body
         ↓
2. ClientController receives the request
         ↓
3. ClientController calls ClientService.createClient()
         ↓
4. ClientService calls ClientRepository.save()
         ↓
5. ClientRepository tells Hibernate to save to MySQL
         ↓
6. MySQL inserts the row and returns the ID
         ↓
7. The saved Client flows back up through all layers
         ↓
```

8. You receive JSON response with the created client

📁 FILE-BY-FILE EXPLANATION

---

1. TimeoutAirlineApplication.java (The Starting Point)

```java
package fr.epita.timeoutairline;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TimeoutairlineApplication {

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

What Each Part Does:

Code	Explanation
`package fr.epita.timeoutairline;`	This file lives in the fr/epita/timeoutairline folder
`import org.springframework.boot.SpringApplication;`	Brings in Spring's application runner
`import org.springframework.boot.autoconfigure.SpringBootApplication;`	Brings in the magic annotation
`@SpringBootApplication`	**MAGIC ANNOTATION** - Does 3 things at once: enables auto-configuration, component scanning, and configuration
`public static void main(String[] args)`	The entry point - Java starts here
`SpringApplication.run(...)`	Starts the Spring Boot application, creates the server, loads all components

What Happens When You Run This:

1. Java calls `main()`
2. Spring Boot starts up
3. Scans all packages for `@Controller`, `@Service`, `@Repository`, `@Entity`
4. Creates all the beans (objects) automatically
5. Starts Tomcat server on port 8084
6. Your app is ready to receive requests!

📦 MODEL LAYER (Entities)

These classes represent database tables. Each class = one table.

---

2. User.java (Parent Entity)

java

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "users")
@Inheritance(strategy = InheritanceType.JOINED)
public class User {

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

    private String firstname;
    private String lastname;
    private String address;
    private String email;
    private String phone;
    private LocalDate birthdate;

    // Default constructor (required by JPA)
    public User() {
    }

    // Constructor with fields
    public User(String firstname, String lastname, String address,
                String email, String phone, LocalDate birthdate) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.address = address;
        this.email = email;
        this.phone = phone;
        this.birthdate = birthdate;
    }

    // Getters and Setters...
}
```

Imports Explained:

Import	Why We Need It
`jakarta.persistence.*`	Brings in all JPA annotations (@Entity, @Id, @Table, etc.)
`java.time.LocalDate`	For the birthdate field (stores date without time)

Annotations Explained:

Annotation	What It Does
`@Entity`	Tells JPA: "This class is a database table"
`@Table(name = "users")`	Names the table "users" in MySQL (otherwise it would be "user")
`@Inheritance(strategy = InheritanceType.JOINED)`	Says "Child classes (Client, Employee) will have their own tables linked by foreign key"
`@Id`	Marks `idUser` as the PRIMARY KEY
`@GeneratedValue(strategy = GenerationType.IDENTITY)`	AUTO INCREMENT - MySQL generates the ID automatically

Why Default Constructor?

```java
public User() {
}
```

JPA needs this because when loading data from database:

1. JPA creates empty object: `new User()`
2. JPA fills fields using setters: `user.setFirstname("John")`

Without default constructor, JPA crashes!

What MySQL Creates:

```sql
CREATE TABLE users (
    id_user BIGINT PRIMARY KEY AUTO_INCREMENT,
    firstname VARCHAR(255),
    lastname VARCHAR(255),
    address VARCHAR(255),
    email VARCHAR(255),
    phone VARCHAR(255),
    birthdate DATE
);
```

---

3. Client.java (Child of User)

java

```
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;

@Entity
@Table(name = "clients")
public class Client extends User {

    @Column(unique = true)
    private String numPassport;

    public Client() {
    }

    public Client(String firstname, String lastname, String address,
                  String email, String phone, java.time.LocalDate birthdate,
                  String numPassport) {
        super(firstname, lastname, address, email, phone, birthdate);
        this.numPassport = numPassport;
    }

    // Getters and Setters...
}
```

Key Concepts:

Code	Explanation
`extends User`	Client inherits ALL fields from User (firstname, lastname, etc.)
`@Column(unique = true)`	No two clients can have the same passport number
`super(...)`	Calls the parent (User) constructor to set inherited fields

What MySQL Creates:

```sql
CREATE TABLE clients (
    id_user BIGINT PRIMARY KEY,  -- Foreign key to users table
    num_passport VARCHAR(255) UNIQUE,
    FOREIGN KEY (id_user) REFERENCES users(id_user)
);
```

How Inheritance Works in Database:

```
When you create a Client:
1. INSERT INTO users (firstname, lastname, ...) → gets id_user = 1
2. INSERT INTO clients (id_user, num_passport) → uses id_user = 1

When you GET a Client:
1. SELECT * FROM users u JOIN clients c ON u.id_user = c.id_user WHERE u.id_user = 1
2. You get ALL fields (User + Client combined)
```

---

4. Employee.java (Child of User)

```java
```

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "employees")
public class Employee extends User {

    @Column(unique = true)
    private String numEmp;

    private String profession;
    private String title;

    public Employee() {
    }

    public Employee(String firstname, String lastname, String address,
                    String email, String phone, LocalDate birthdate,
                    String numEmp, String profession, String title) {
        super(firstname, lastname, address, email, phone, birthdate);
        this.numEmp = numEmp;
        this.profession = profession;
        this.title = title;
    }

    // Getters and Setters...
}
```

Same pattern as Client:

- Extends User
- Has its own unique fields (numEmp, profession, title)
- Creates separate `employees` table linked to `users`

---

5. Plane.java (Simple Entity)

```
java
```

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;

@Entity
@Table(name = "planes")
public class Plane {

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

    private String brand;
    private String model;
    private Integer manufacturingYear;

    public Plane() {
    }

    public Plane(String brand, String model, Integer manufacturingYear) {
        this.brand = brand;
        this.model = model;
        this.manufacturingYear = manufacturingYear;
    }

    // Getters and Setters...
}
```

Why Integer instead of int?

java

```java
private Integer manufacturingYear;  // Can be null
private int manufacturingYear;      // Cannot be null - causes JSON parse error!
```

When receiving JSON like `{"idPlane": 1}`, the other fields are null. Primitive `int` can't be null, but `Integer` can!

---

6. Airport.java (Simple Entity)

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;

@Entity
@Table(name = "airports")
public class Airport {

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

    private String nameAirport;
    private String countryAirport;
    private String cityAirport;

    public Airport() {
    }

    public Airport(String nameAirport, String countryAirport, String cityAirport) {
        this.nameAirport = nameAirport;
        this.countryAirport = countryAirport;
        this.cityAirport = cityAirport;
    }

    // Getters and Setters...
}
```

Nothing new here - same pattern as Plane.

---

7. Flight.java (Entity with Relationships)

```java
```

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDate;

@Entity
@Table(name = "flights")
public class Flight {

    @Id
    private String flightNumber;  // Not auto-generated!

    private String departureCity;
    private String arrivalCity;
    private String departureHour;
    private String arrivalHour;
    private LocalDate departureDate;
    private Integer numberOfSeat;
    private BigDecimal firstClassSeatPrice;
    private BigDecimal premiumSeatPrice;
    private BigDecimal businessClassPrice;
    private BigDecimal economicsClassPrice;

    @ManyToOne
    @JoinColumn(name = "id_plane")
    private Plane plane;

    @ManyToOne
    @JoinColumn(name = "departure_airport_id")
    private Airport departureAirport;

    @ManyToOne
    @JoinColumn(name = "arrival_airport_id")
    private Airport arrivalAirport;

    // Constructors, Getters, Setters...
}
```

New Concepts:

Code	Explanation
`@Id private String flightNumber`	Primary key is a String (like "TA101"), NOT auto-generated
`BigDecimal`	Used for money - more precise than double
`@ManyToOne`	Many flights can use ONE plane
`@JoinColumn(name = "id_plane")`	Creates foreign key column `id_plane` in flights table

Relationship Explained:

```
@ManyToOne
@JoinColumn(name = "id_plane")
private Plane plane;
```

This means:

- Many Flight records can point to One Plane record
- Creates a column `id_plane` in the `flights` table
- This column stores the Plane's ID as a foreign key

Visual:

```
flights table:
+--------------+--------+---------------------+--------------------+
| flight_number | id_plane | departure_airport_id | arrival_airport_id |
+--------------+--------+---------------------+--------------------+
| TA101        | 1     | 1                    | 2                  |
| TA102        | 1     | 2                    | 3                  | ← Same plane!
| TA103        | 2     | 1                    | 3                  |
+--------------+-------+---------------------+--------------------+
               ↓       ↓                     ↓
          planes(1)  airports(1)         airports(2)
```

---

8. Booking.java (Entity with Enum)

java

```java
package fr.epita.timeoutairline.model;

import jakarta.persistence.*;
import java.time.LocalDateTime;
```

```java
@Entity
@Table(name = "bookings")
public class Booking {

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

    @ManyToOne
    @JoinColumn(name = "flight_number")
    private Flight flight;

    @ManyToOne
    @JoinColumn(name = "client_id")
    private Client client;

    @Enumerated(EnumType.STRING)
    private SeatType typeOfSeat;

    private LocalDateTime bookingDate;

    public enum SeatType {
        FIRST_CLASS,
        PREMIUM,
        BUSINESS,
        ECONOMICS
    }

    public Booking() {
        this.bookingDate = LocalDateTime.now();
    }

    public Booking(Flight flight, Client client, SeatType typeOfSeat) {
        this.flight = flight;
        this.client = client;
        this.typeOfSeat = typeOfSeat;
        this.bookingDate = LocalDateTime.now();
    }

    // Getters and Setters...
}
```

New Concepts:

Code	Explanation

`@Enumerated(EnumType.STRING)`	Stores enum as text ("BUSINESS") not number (2)
`enum SeatType`	Defines allowed values for seat type
`LocalDateTime.now()`	Auto-sets booking time when object is created

Why EnumType.STRING?

```java
@Enumerated(EnumType.STRING)   // Stores "BUSINESS" in database ✅
@Enumerated(EnumType.ORDINAL)  // Stores 2 in database (position in enum) ❌
```

STRING is better because:

- More readable in database
- If you reorder the enum, data doesn't break

---

9. MilesReward.java

```java



package fr.epita.timeoutairline.model;

import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "miles_rewards")
public class MilesReward {

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

    @ManyToOne
    @JoinColumn(name = "client_id")
    private Client client;

    @ManyToOne
    @JoinColumn(name = "flight_number")
    private Flight flight;

    private LocalDate date;
    private String discountCode;

    public MilesReward() {
        this.date = LocalDate.now();
    }

    public MilesReward(Client client, Flight flight) {
        this.client = client;
        this.flight = flight;
        this.date = LocalDate.now();
    }

    // Getters and Setters...
}
```

This table records every booking and stores discount codes when earned.

📁 REPOSITORY LAYER

Repositories talk to the database. Spring Data JPA generates the implementation automatically!

---

10. UserRepository.java

```java
package fr.epita.timeoutairline.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import fr.epita.timeoutairline.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Breaking It Down:

Code	Explanation
`interface`	Not a class! Just defines methods

@Repository	Marks this as a database access component
extends JpaRepository<User, Long>	User = entity type, Long = ID type

What JpaRepository Gives You FREE:

java

```java
userRepository.save(user);          // INSERT or UPDATE
userRepository.findById(1L);        // SELECT WHERE id = 1
userRepository.findAll();           // SELECT * FROM users
userRepository.deleteById(1L);      // DELETE WHERE id = 1
userRepository.count();             // SELECT COUNT(*)
userRepository.existsById(1L);      // Returns true/false
```

You write ZERO SQL! Spring generates it all.

---

11. ClientRepository.java (with Custom Method)

java

```java
package fr.epita.timeoutairline.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import fr.epita.timeoutairline.model.Client;
import java.util.Optional;

@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {

    Optional<Client> findByNumPassport(String numPassport);
}
```

Custom Method Magic:

java

```
Optional<Client> findByNumPassport(String numPassport);
```

Spring reads the method name and generates:

```sql
SELECT * FROM clients c
JOIN users u ON c.id_user = u.id_user
WHERE c.num_passport = ?
```

Method Naming Rules:

Method Name	Generated SQL
`findByNumPassport(String x)`	WHERE num_passport = x
`findByFirstname(String x)`	WHERE firstname = x
`findByFirstnameAndLastname(String x, String y)`	WHERE firstname = x AND lastname = y
`findByAgeLessThan(int x)`	WHERE age < x
`findByEmailContaining(String x)`	WHERE email LIKE '%x%'

---

12. FlightRepository.java (with Complex Query)

```java




package fr.epita.timeoutairline.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import fr.epita.timeoutairline.model.Flight;
import java.time.LocalDate;
import java.util.List;

@Repository
public interface FlightRepository extends JpaRepository<Flight, String> {

    List<Flight> findByDepartureCityAndArrivalCityAndDepartureDate(
    List<Flight> findByDepartureCityAndArrivalCityAndDepartureDate(
        String departureCity,
        String arrivalCity,
        LocalDate departureDate
    );
}
```

The Long Method Name:

```java
findByDepartureCityAndArrivalCityAndDepartureDate(...)
```

Spring generates:

```sql
SELECT * FROM flights
WHERE departure_city = ?
AND arrival_city = ?
AND departure_date = ?
```

This is used for **Flight Search** (Feature #6)!

---

13. BookingRepository.java

```java
package fr.epita.timeoutairline.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import fr.epita.timeoutairline.model.Booking;
import fr.epita.timeoutairline.model.Client;
import fr.epita.timeoutairline.model.Flight;
import java.util.List;

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

    int countByFlight(Flight flight);
    List<Booking> findByClient(Client client);
    List<Booking> findByFlight(Flight flight);
}
```

countByFlight Explained:

java

```java
int countByFlight(Flight flight);
```

Generates:

sql

```sql
SELECT COUNT(*) FROM bookings WHERE flight_number = ?
```

This is used to **check seat availability**!

---

14. MilesRewardRepository.java (with Custom JPQL Query)

java

```java
package fr.epita.timeoutairline.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import fr.epita.timeoutairline.model.Client;
import fr.epita.timeoutairline.model.MilesReward;
import java.util.List;

@Repository
public interface MilesRewardRepository extends JpaRepository<MilesReward, Long> {

    List<MilesReward> findByClient(Client client);

    @Query("SELECT COUNT(m) FROM MilesReward m WHERE m.client = :client AND YEAR(m.date) = :year")
    int countFlightsByClientAndYear(@Param("client") Client client, @Param("year") int year);
}
```

@Query Explained:

When method naming isn't enough, write your own query:

```java
@Query("SELECT COUNT(m) FROM MilesReward m WHERE m.client = :client AND YEAR(m.date) = :year")
int countFlightsByClientAndYear(@Param("client") Client client, @Param("year") int year);
```

Part	Meaning
`@Query("...")`	Custom JPQL query (Java version of SQL)
`MilesReward m`	Alias for the entity
`:client`	Parameter placeholder
`@Param("client")`	Binds method parameter to query placeholder
`YEAR(m.date)`	Extracts year from date

This counts how many flights a client took in a specific year - used for **discount code generation**!

🔧 SERVICE LAYER

Services contain business logic. They're the "brain" of the application.

---

15. UserService.java

```java
package fr.epita.timeoutairline.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import fr.epita.timeoutairline.model.User;
import fr.epita.timeoutairline.repository.UserRepository;
import java.util.List;
import java.util.Optional;
```

```
@Service
public class UserService {


    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public User updateUser(Long id, User userDetails) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("User not found with id: " + id));

        user.setFirstname(userDetails.getFirstname());
        user.setLastname(userDetails.getLastname());
        user.setAddress(userDetails.getAddress());
        user.setEmail(userDetails.getEmail());
        user.setPhone(userDetails.getPhone());
        user.setBirthdate(userDetails.getBirthdate());

        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("User not found with id: " + id));
        userRepository.delete(user);
    }
}
```

Annotations Explained:

Annotation	What It Does
`@Service`	Marks this class as a service component. Spring creates one instance and manages it.
`@Autowired`	Dependency Injection - Spring automatically provides the UserRepository

@Autowired Deep Dive:

java

```java
@Autowired
private UserRepository userRepository;
```

Without @Autowired, you'd have to do:

java

```java
private UserRepository userRepository = new UserRepositoryImpl();  // Doesn't work!
```

With @Autowired, Spring:

1. Creates UserRepository automatically
2. Injects it into this field
3. You just use it!

Update Logic Explained:

java

```java
public User updateUser(Long id, User userDetails) {
    // Step 1: Find existing user or throw error
    User user = userRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("User not found"));

    // Step 2: Update each field
    user.setFirstname(userDetails.getFirstname());
    user.setLastname(userDetails.getLastname());
    // ... etc

    // Step 3: Save (UPDATE in SQL because user already has an ID)
    return userRepository.save(user);
}
```

---

16. BookingService.java (Complex Business Logic)

java

```java
package fr.epita.timeoutairline.service;
package fr.epita.timeoutairline.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import fr.epita.timeoutairline.model.*;
import fr.epita.timeoutairline.repository.*;
import fr.epita.timeoutairline.dto.BookingRequest;
import fr.epita.timeoutairline.exception.ResourceNotFoundException;
import fr.epita.timeoutairline.exception.NoSeatAvailableException;
import java.time.LocalDate;
import java.util.List;
import java.util.Optional;
import java.util.UUID;

@Service
public class BookingService {

    @Autowired
    private BookingRepository bookingRepository;

    @Autowired
    private MilesRewardRepository milesRewardRepository;

    @Autowired
    private FlightRepository flightRepository;

    @Autowired
    private ClientRepository clientRepository;

    public Booking createBooking(Booking booking) {
        Flight flight = booking.getFlight();

        // Check if flight exists
        Flight existingFlight = flightRepository.findById(flight.getFlightNumber())
            .orElseThrow(() -> new ResourceNotFoundException("Flight", "flightNumber", flight.getFlightNumber()));

        // Check seat availability
        int bookedSeats = bookingRepository.countByFlight(existingFlight);
        if (bookedSeats >= existingFlight.getNumberOfSeat()) {
            throw new NoSeatAvailableException(existingFlight.getFlightNumber(),
                existingFlight.getNumberOfSeat(), bookedSeats);
        }

        booking.setFlight(existingFlight);
        Booking savedBooking = bookingRepository.save(booking);

        // Record in MilesReward
        recordMilesReward(booking.getClient(), existingFlight);
        return savedBooking;
    }


    public Booking createBookingFromRequest(BookingRequest request) {
        // Find or create client
        Client client = clientRepository.findByNumPassport(request.getPassportNumber())
            .orElseGet(() -> {
                Client newClient = new Client();
                newClient.setFirstname(request.getFirstname());
                newClient.setLastname(request.getLastname());
                newClient.setNumPassport(request.getPassportNumber());
                newClient.setBirthdate(request.getBirthdate());
                return clientRepository.save(newClient);
            });

        // Find flight
        Flight flight = flightRepository.findById(request.getFlightNumber())
            .orElseThrow(() -> new ResourceNotFoundException("Flight", "flightNumber", request.getFlightNumber()));

        // Check seats
        int bookedSeats = bookingRepository.countByFlight(flight);
        if (bookedSeats >= flight.getNumberOfSeat()) {
            throw new NoSeatAvailableException(flight.getFlightNumber());
        }

        // Create booking
        Booking booking = new Booking(flight, client, request.getTypeOfSeat());
        Booking savedBooking = bookingRepository.save(booking);

        // Record miles
        recordMilesReward(client, flight);

        return savedBooking;
    }

    private void recordMilesReward(Client client, Flight flight) {
        MilesReward milesReward = new MilesReward(client, flight);

        int currentYear = LocalDate.now().getYear();
        int flightsThisYear = milesRewardRepository.countFlightsByClientAndYear(client, currentYear);

        // Generate discount on every 3rd flight
        if ((flightsThisYear + 1) % 3 == 0) {
            String discountCode = generateDiscountCode();
            milesReward.setDiscountCode(discountCode);
        }
        milesRewardRepository.save(milesReward);
    }


    private String generateDiscountCode() {
        return "DISC-" + UUID.randomUUID().toString().substring(0, 8).toUpperCase();
    }

    // ... other methods
}
```

Business Logic Flow:

```
createBookingFromRequest(request):
  │
  ├─▶ 1. Find client by passport OR create new client
  │       └─▶ clientRepository.findByNumPassport()
  │       └─▶ If not found: clientRepository.save(newClient)
  │
  ├─▶ 2. Find flight
  │       └─▶ flightRepository.findById()
  │       └─▶ If not found: throw ResourceNotFoundException
  │
  ├─▶ 3. Check seat availability
  │       └─▶ bookingRepository.countByFlight()
  │       └─▶ If full: throw NoSeatAvailableException
  │
  ├─▶ 4. Create and save booking
  │       └─▶ bookingRepository.save()
  │
  ├─▶ 5. Record in MilesReward
  │       └─▶ Count flights this year
  │       └─▶ If 3rd flight: generate discount code
  │       └─▶ milesRewardRepository.save()
  │
  └─▶ 6. Return saved booking
```

Discount Code Logic:

```java
if ((flightsThisYear + 1) % 3 == 0) {
    // Generate discount
}
```

flightsThisYear	+1	% 3	== 0?	Discount?
0	1	1	No	❌
1	2	2	No	❌
2	3	0	Yes	✅
3	4	1	No	❌
4	5	2	No	❌
5	6	0	Yes	✅

🌐 CONTROLLER LAYER

Controllers receive HTTP requests and return responses.

---

17. UserController.java

```java
package fr.epita.timeoutairline.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import fr.epita.timeoutairline.model.User;
import fr.epita.timeoutairline.service.UserService;
import java.util.List;

@RestController
@RequestMapping("/api/v1/users")
public class UserController {

    @Autowired
    private UserService userService;

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }
    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User userDetails) {
        try {
            User updatedUser = userService.updateUser(id, userDetails);
            return ResponseEntity.ok(updatedUser);
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        try {
            userService.deleteUser(id);
            return ResponseEntity.noContent().build();
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Annotations Explained:

Annotation	What It Does
`@RestController`	This class handles REST API requests. Returns JSON automatically.
`@RequestMapping("/api/v1/users")`	Base URL for all endpoints in this controller
`@PostMapping`	Handles POST requests (CREATE)
`@GetMapping`	Handles GET requests (READ)
`@PutMapping`	Handles PUT requests (UPDATE)
`@DeleteMapping`	Handles DELETE requests (DELETE)

@DeleteMapping	Handles DELETE requests (DELETE)
@RequestBody	Converts JSON body to Java object
@PathVariable	Gets value from URL path

Endpoint Mapping:

Annotation	Full URL	HTTP Method
@PostMapping	POST /api/v1/users	CREATE
@GetMapping	GET /api/v1/users	READ ALL
@GetMapping("/{id}")	GET /api/v1/users/1	READ ONE
@PutMapping("/{id}")	PUT /api/v1/users/1	UPDATE
@DeleteMapping("/{id}")	DELETE /api/v1/users/1	DELETE

@RequestBody Explained:

```java
@PostMapping
public User createUser(@RequestBody User user) {
    return userService.createUser(user);
}
```

```
Request:
POST /api/v1/users
Content-Type: application/json
{
    "firstname": "John",
    "lastname": "Doe",
    "email": "john@example.com"
}

@RequestBody converts this JSON ──▶ User object
```

@PathVariable Explained:

```java
@GetMapping("/{id}")
```

```
public ResponseEntity<User> getUserById(@PathVariable Long id) {
    // ...
}
```

```
Request: GET /api/v1/users/42
                          ↓
@PathVariable extracts ─▶ id = 42
```

ResponseEntity Explained:

```java
return ResponseEntity.ok(user);           // 200 OK + body
return ResponseEntity.notFound().build(); // 404 Not Found
return ResponseEntity.noContent().build(); // 204 No Content
return ResponseEntity.badRequest().body("Error message"); // 400 Bad Request
```

---

18. FlightController.java (with Query Parameters)

```java
@RestController
@RequestMapping("/api/v1/flights")
public class FlightController {

    @Autowired
    private FlightService flightService;

    // ... other endpoints

    @GetMapping("/search")
    public List<Flight> searchFlights(
            @RequestParam String from,
            @RequestParam String to,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate date) {
        return flightService.searchFlights(from, to, date);
    }
}
```

@RequestParam Explained:

```java
```

```
@GetMapping("/search")
public List<Flight> searchFlights(
    @RequestParam String from,
    @RequestParam String to,
    @RequestParam LocalDate date
) { ... }
```

```
Request: GET /api/v1/flights/search?from=Paris&to=London&date=2025-01-15
                                        ↓       ↓          ↓
@RequestParam extracts ──────────────────▶ from    to         date
```

@DateTimeFormat Explained:

```java
@RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate date
```

Tells Spring how to parse the date string "2025-01-15" into a LocalDate object.

📦 DTO LAYER

DTOs (Data Transfer Objects) are simple classes for transferring data.

---

19. BookingRequest.java

```java
```

```java
package fr.epita.timeoutairline.dto;

import fr.epita.timeoutairline.model.Booking.SeatType;
import java.time.LocalDate;

public class BookingRequest {

    private String lastname;
    private String firstname;
    private String passportNumber;
    private LocalDate birthdate;
    private String departureCity;
    private String arrivalCity;
    private String departureHour;
    private String arrivalHour;
    private String flightNumber;
    private SeatType typeOfSeat;

    // Default constructor
    public BookingRequest() {
    }

    // Getters and Setters...
}
```

Why DTOs?

Without DTO	With DTO
API receives entity directly	API receives DTO, service converts to entity
Client must know entity structure	Client sends simple, flat JSON
Tight coupling	Loose coupling

BookingRequest matches Project Requirement #7:

> "Implement a rest api that will allow a new customer or an existing customer to book a flight by providing a **lastname, firstname, passport number, birthdate, departure city, arrival city, departure hour, arrival hour, flight number**."

⚠️ EXCEPTION LAYER

Custom exceptions for better error handling.

---

20. ResourceNotFoundException.java

java

```java
package fr.epita.timeoutairline.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {

    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {
        super(String.format("%s not found with %s: '%s'", resourceName, fieldName, fieldValue));
    }
}
```

@ResponseStatus Explained:

```java
@ResponseStatus(value = HttpStatus.NOT_FOUND)
```

When this exception is thrown, Spring automatically returns HTTP 404.

Usage:

```java
throw new ResourceNotFoundException("Flight", "flightNumber", "TA999");
// Returns: "Flight not found with flightNumber: 'TA999'"
```

---

21. NoSeatAvailableException.java

```java
package fr.epita.timeoutairline.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public class NoSeatAvailableException extends RuntimeException {

    public NoSeatAvailableException(String flightNumber) {
        super(String.format("No available seats on flight: %s", flightNumber));
    }

    public NoSeatAvailableException(String flightNumber, int totalSeats, int bookedSeats) {
        super(String.format("No available seats on flight %s. Total: %d, Booked: %d",
            flightNumber, totalSeats, bookedSeats));
    }
}
```

---

22. GlobalExceptionHandler.java

java

```java
package fr.epita.timeoutairline.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Map<String, Object>> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {

        Map<String, Object> body = new HashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", HttpStatus.NOT_FOUND.value());
        body.put("error", "Not Found");
        body.put("message", ex.getMessage());
        body.put("path", request.getDescription(false).replace("uri=", ""));

        return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
    }

    // ... other handlers
}
```

@ControllerAdvice Explained:

java

```java
@ControllerAdvice
```

This class catches exceptions from ALL controllers. It's a global exception handler.

@ExceptionHandler Explained:

java

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<...> handleResourceNotFoundException(...) {
    // Handle this specific exception
}
```

When `ResourceNotFoundException` is thrown anywhere, this method catches it and returns a nice JSON error response.

Result:

```json
{
    "timestamp": "2025-12-15T18:30:00",
    "status": 404,
    "error": "Not Found",
    "message": "Flight not found with flightNumber: 'TA999'",
    "path": "/api/v1/flights/TA999"
}
```

🔗 HOW EVERYTHING CONNECTS

Complete Request Flow Example

Request: `POST /api/v1/bookings/book` with booking details

```
┌─────────────────────────────────────────────────────────────┐
│ 1. HTTP Request arrives                                     │
│   POST /api/v1/bookings/book                                │
│   Body: {"firstname": "John", "passportNumber": "AB123", ...} │
└─────────────────────────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
│ 2. BookingController.bookFlight()                           │
│   @PostMapping("/book")                                     │
│   @RequestBody converts JSON → BookingRequest object        │
│   Calls: bookingService.createBookingFromRequest(request)   │
└─────────────────────────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
```