🎤 TIMEOUT AIRLINE - COMPLETE PRESENTATION SPEECH

How to Use This Document

- Read each section out loud
- Practice before your presentation
- The speech is written as if YOU are talking
- Feel free to modify to match your speaking style

PART 1: INTRODUCTION (2 minutes)

Opening

"Good morning/afternoon everyone. Today I'm going to present our project called **Timeout Airline** - a flight booking system that we built using Java Spring Boot.

Before I show you the code, let me explain what this application does:

- Customers can search for flights
- They can book flights online
- The system tracks their bookings and rewards them with discount codes after every 3 flights
- Airline staff can manage planes, airports, flights, and customers

My teammate and I divided the work:

- I handled User, Client, Employee, Booking, and the MilesReward system
- My teammate handled Plane, Airport, Flight, and the flight search feature

Now let me take you through how we built this, step by step."

PART 2: PROJECT SETUP & DEPENDENCIES (3 minutes)

Explaining Spring Initializr

"We started by creating the project using Spring Initializr. Let me explain the choices we made:

Project Type: Maven We chose Maven because it's a build tool that manages our dependencies. Instead of

manually downloading JAR files, we just list what we need in a file called `pom.xml`, and Maven downloads everything for us.

Java Version: 21 This is the latest Long-Term Support version of Java.

Spring Boot Version: 3.4 Spring Boot is a framework that makes it easy to create web applications. It handles a lot of configuration automatically so we can focus on writing business logic.

Now let me explain each dependency we added:"

Dependencies Explanation

"**1. Spring Web** This gives us the ability to create REST APIs. It includes an embedded Tomcat server, so we don't need to install a separate web server. When I write `@RestController` and `@GetMapping`, this dependency makes it work.

2. Spring Data JPA JPA stands for Java Persistence API. It's a specification for how Java objects should be saved to databases. Spring Data JPA implements this and gives us ready-made methods like `save()`, `findById()`, `findAll()`, and `deleteById()` without writing any SQL. We just create an interface that extends `JpaRepository` and Spring generates the implementation automatically.

3. MySQL Driver This is the connector that allows our Java application to talk to the MySQL database. It translates our JPA commands into MySQL-specific SQL queries.

4. Spring Boot DevTools This is a developer convenience tool. When I change my code and save the file, it automatically restarts the application so I don't have to stop and start it manually every time.

5. Lombok (if used) Lombok reduces boilerplate code. Instead of writing getters, setters, and constructors manually, we can use annotations like `@Getter`, `@Setter`, and `@AllArgsConstructor`. However, for this project we wrote them manually to better understand the code.

6. Validation This allows us to validate user input using annotations like `@NotNull`, `@Email`, and `@Size`. For example, we can ensure an email field actually contains a valid email address."

PART 3: PROJECT STRUCTURE (2 minutes)

"Now let me explain how we organized our code. We followed a layered architecture with clear separation of concerns:

```
src/main/java/fr/epita/timeoutairline/
├── model/          → Entities (database tables)
├── repository/     → Database access
├── service/        → Business logic
├── controller/     → REST APIs
├── dto/            → Data transfer objects
```

```
└── exception/      → Custom error handling
```

Why this structure?

Each layer has one job:

- **Model** defines WHAT data we store
- **Repository** handles HOW we access the database
- **Service** contains the business rules
- **Controller** receives HTTP requests and returns responses

This follows the **SOLID principles**, specifically:

- **Single Responsibility Principle**: Each class does one thing
- **Dependency Inversion**: Higher layers depend on abstractions, not concrete implementations

When a request comes in, it flows like this:

```
Client → Controller → Service → Repository → Database
```

And the response flows back:

```
Database → Repository → Service → Controller → Client
```

This makes our code:

- Easy to test (we can test each layer separately)
- Easy to maintain (changes in one layer don't affect others)
- Easy to understand (clear separation of concerns)"

PART 4: THE MODEL LAYER - ENTITIES (5 minutes)

"Let me now explain the Model layer. These are our entities - Java classes that represent database tables.

User.java - The Parent Entity

```
java
```

```java
@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;
    // ... other fields
}
```

Let me explain each annotation:

`@Entity` This tells JPA that this class should be mapped to a database table. Without this annotation, JPA would ignore this class completely.

`@Table(name = "users")` This specifies the exact table name in MySQL. If I didn't include this, JPA would create a table called `user`, but `user` is a reserved word in some databases, so we use `users` to be safe.

`@Inheritance(strategy = InheritanceType.JOINED)` This is where it gets interesting. We have three types of users: regular User, Client, and Employee. Client and Employee extend User.

With JOINED strategy:

- The `users` table stores common fields (firstname, lastname, email, etc.)
- The `clients` table stores only client-specific fields (numPassport)
- The `employees` table stores only employee-specific fields (numEmp, profession, title)

When we query a Client, JPA automatically JOINS these tables together. This is more normalized and avoids data duplication.

`@Id` This marks `idUser` as the primary key of the table.

`@GeneratedValue(strategy = GenerationType.IDENTITY)` This tells MySQL to auto-increment the ID. When we insert a new user, we don't provide an ID - MySQL generates it automatically (1, 2, 3, ...).

Why do we need a default constructor?

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

JPA needs this because when it loads data from the database, it:

1. Creates an empty object using `new User()`
2. Then fills the fields using setter methods

Without a default constructor, JPA would crash with an error.

Client.java - Child Entity

java

```java
@Entity
@Table(name = "clients")
public class Client extends User {
    @Column(unique = true)
    private String numPassport;
}
```

`extends User` Client inherits ALL fields from User. So a Client has firstname, lastname, email AND numPassport.

`@Column(unique = true)` This creates a UNIQUE constraint in the database. No two clients can have the same passport number. If we try to insert a duplicate, MySQL will reject it.

The constructor uses `super()`:

java

```java
public Client(String firstname, String lastname, ..., String numPassport) {
    super(firstname, lastname, ...);  // Calls User constructor
    this.numPassport = numPassport;
}
```

Flight.java - Entity with Relationships

java

```
@Entity
@Table(name = "flights")
public class Flight {
    @Id
    private String flightNumber;  // Not auto-generated!

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

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

Notice `flightNumber` **has no** `@GeneratedValue` Unlike other entities, the flight number is a String like 'TA101' that we provide, not an auto-generated number.

`@ManyToOne` This defines a relationship. Many flights can use ONE plane. Think about it - the same Boeing 737 can fly Paris to London in the morning, then London to New York in the afternoon.

`@JoinColumn(name = "id_plane")` This creates a foreign key column called `id_plane` in the flights table. This column stores the ID of the related plane.

In the database, it looks like this:

```
flights table:
| flight_number | departure_city | id_plane | departure_airport_id |
|---------------|----------------|----------|----------------------|
| TA101         | Paris          | 1        | 1                    |
| TA102         | London         | 1        | 2                    |
```

See how both flights use plane ID 1? That's the @ManyToOne relationship.

Booking.java - Entity with Enum

java

```
@Enumerated(EnumType.STRING)
private SeatType typeOfSeat;

public enum SeatType {
```

```
    FIRST_CLASS,
    PREMIUM,
    BUSINESS,
    ECONOMICS
}
```

enum SeatType An enum is a fixed set of constants. A seat can ONLY be one of these four types. You can't create a seat type called 'SUPER_LUXURY' - the compiler won't allow it.

@Enumerated(EnumType.STRING) This tells JPA to store the enum as text in the database. So it stores 'BUSINESS', not a number like 2.

Why STRING instead of ORDINAL?

- ORDINAL stores the position (0, 1, 2, 3)
- If we reorder the enum, all our data becomes wrong!
- STRING is safer and more readable"

PART 5: THE REPOSITORY LAYER (3 minutes)

"Now let's talk about the Repository layer. This is where the magic of Spring Data JPA really shines.

UserRepository.java

java

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

This is an INTERFACE, not a class! We don't write any implementation. Spring generates it automatically at runtime.

extends JpaRepository<User, Long> The first parameter `User` is the entity type. The second parameter `Long` is the ID type.

What do we get for FREE?

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 or false
```

I didn't write ANY SQL. Spring generates it all!

ClientRepository.java - Custom Query Methods

java

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {
    Optional<Client> findByNumPassport(String numPassport);
}
```

Method Name Magic Spring reads the method name and generates the SQL:

`findByNumPassport` becomes:

sql

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

The naming convention is:

- `findBy` + `FieldName`
- `findByFirstnameAndLastname` for multiple conditions
- `findByAgeLessThan` for comparisons
- `countBy` to count records

MilesRewardRepository.java - Custom JPQL 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);
```

When the method name isn't enough, we write our own query using `@Query`.

JPQL vs SQL

- JPQL uses class names (MilesReward) not table names (miles_rewards)
- JPQL uses field names (m.client) not column names (client_id)

`:client` **and** `@Param("client")` The colon creates a parameter placeholder. `@Param` binds the method parameter to the placeholder.

This query counts how many flights a client has taken in a specific year - we use this to determine if they've earned a discount code!"

PART 6: THE SERVICE LAYER (4 minutes)

"The Service layer is where our business logic lives. This is the brain of the application.

UserService.java - Basic CRUD

java

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

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

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

        user.setFirstname(userDetails.getFirstname());
        user.setLastname(userDetails.getLastname());
        // ... update other fields

        return userRepository.save(user);
    }
}
```

This means: @service

— Spring manages this class as a service bean

— It contains business logic

— Controllers should call services, not repositories directly

✅ Correct architecture (Controller → Service → Repository)

This tells Spring: @autowired
“Inject the UserRepository implementation here”

`@Service` This marks the class as a service component. Spring creates one instance and manages its lifecycle.

`@Autowired` This is Dependency Injection. Instead of creating the repository ourselves:

java

```
// WITHOUT @Autowired (bad)
```

```
// WITHOUT @Autowired (bad):
private UserRepository userRepository = new UserRepositoryImpl(); // Doesn't work!

// WITH @Autowired (good):
@Autowired
private UserRepository userRepository; // Spring provides it automatically!
```

Spring:

1. Creates the UserRepository
2. Finds this field marked with @Autowired
3. Injects the repository automatically

This is called **Inversion of Control** - we don't control object creation, Spring does.

Update Logic Explained

```
java

User user = userRepository.findById(id)
    .orElseThrow(() -> new RuntimeException("User not found"));
```

UPDATE:: What happens step by step:
Fetch user from DB
Throw error if not found
Update allowed fields
Save updated entity

`findById` returns an `Optional<User>`. An Optional either contains a User or is empty.

- If found: we get the user
- If not found: we throw an exception

Why Optional?

-- User may not exist
— Avoids NullPointerException
— Controller decides how to respond (404, etc.)

Then we update each field and call `save()`. Since the user already has an ID, JPA performs an UPDATE, not an INSERT.

DELETE: WHAT HAPPENS
Checks existence
Deletes safely
Prevents silent failures

BookingService.java - Complex Business Logic

This is the most important service. Let me walk through the booking flow:

```
java

public Booking createBookingFromRequest(BookingRequest request) {
    // Step 1: 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);
```

```
        return clientRepository.save(newClient);
    });
```

orElseGet(() -> { ... }) If the client doesn't exist, we CREATE a new one. This allows new customers to book without registering first.

```java
    // Step 2: Find flight
    Flight flight = flightRepository.findById(request.getFlightNumber())
        .orElseThrow(() -> new ResourceNotFoundException("Flight", "flightNumber", request.getFlightNumber()));
```

If the flight doesn't exist, we throw our custom exception.

```java
    // Step 3: Check seat availability
    int bookedSeats = bookingRepository.countByFlight(flight);
    if (bookedSeats >= flight.getNumberOfSeat()) {
        throw new NoSeatAvailableException(flight.getFlightNumber());
    }
```

We count existing bookings and compare with total seats. If the flight is full, we reject the booking. This prevents overbooking!

```java
    // Step 4: Create and save booking
    Booking booking = new Booking(flight, client, request.getTypeOfSeat());
    Booking savedBooking = bookingRepository.save(booking);

    // Step 5: Record in MilesReward
    recordMilesReward(client, flight);

    return savedBooking;
}
```

The Discount Code Logic

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

BUSINESS RULES THAT SERVICE CONTAINS

Airline Project — Concrete Service Examples

✈️ UserService
Contains:
Email uniqueness check
Update rules
Deletion constraints (e.g., cannot delete user with active bookings)

✈️ FlightService
Contains:
Seat availability logic
Price rules

Flight scheduling rules
if (departureDate.isBefore(LocalDate.now())) {
throw new IllegalStateException("Cannot schedule flight in the past");
}

✈️ BookingService (VERY IMPORTANT)
Contains:
Seat availability rules
Creating bookings
Creating miles rewards
Generating discount codes
Coordinating multiple repositories

✈️ MilesRewardService
Contains:
Reward eligibility rules
Discount generation logic
Expiry logic

5️⃣ Golden Rule (memorize this)
If the rule describes HOW the business works → Service
If it describes HOW data is stored → Repository
If it describes HOW the API is called → Controller

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

    if ((flightsThisYear + 1) % 3 == 0) {
        String discountCode = generateDiscountCode();
        milesReward.setDiscountCode(discountCode);
    }

    milesRewardRepository.save(milesReward);
}
```

The modulo logic explained:

Flights This Year	+1	% 3	== 0?	Discount?
0 (first booking)	1	1	No	❌
1	2	2	No	❌
2	3	0	Yes	✅ Third flight!
3	4	1	No	❌
4	5	2	No	❌
5	6	0	Yes	✅ Sixth flight!

Every third flight, we generate a discount code using `UUID.randomUUID()` which creates a unique random string."

PART 7: THE CONTROLLER LAYER (4 minutes)

"The Controller layer is the entry point for HTTP requests. It receives requests, calls services, and returns responses.

UserController.java

java

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

    @Autowired
    private UserService userService;
```

```java
    @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());
    }
}
```

`@RestController` This combines two annotations:

- `@Controller` - marks this as a web controller
- `@ResponseBody` - automatically converts return values to JSON

`@RequestMapping("/api/v1/users")` This sets the base URL for all endpoints in this controller. All methods will start with `/api/v1/users`.

HTTP Method Annotations:

Annotation	HTTP Method	Purpose	Example URL
`@PostMapping`	POST	Create	POST /api/v1/users
`@GetMapping`	GET	Read	GET /api/v1/users
`@PutMapping`	PUT	Update	PUT /api/v1/users/1
`@DeleteMapping`	DELETE	Delete	DELETE /api/v1/users/1

`@RequestBody`

```java
@PostMapping
public User createUser(@RequestBody User user) {
```

When someone sends:

```json
POST /api/v1/users
Content-Type: application/json

{
  "firstname": "John",
  "lastname": "Doe",
  "email": "john@example.com"
}
```

`@RequestBody` converts this JSON into a User object automatically. Spring uses the Jackson library to do this conversion.

`@PathVariable`

```java

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

When someone requests `GET /api/v1/users/42`, the `{id}` placeholder captures `42`, and `@PathVariable` puts it into the `id` parameter.

`ResponseEntity` This gives us control over the HTTP response:

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

FlightController.java - Query Parameters

```java
@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 This captures query string parameters:

```
GET /api/v1/flights/search?from=Paris&to=London&date=2025-01-15
                          ↓     ↓       ↓
                        from    to     date
```

@DateTimeFormat This tells Spring how to parse the date string '2025-01-15' into a LocalDate object."

PART 8: DTOs - DATA TRANSFER OBJECTS (2 minutes)

"DTOs are simple classes used to transfer data between layers.

BookingRequest.java

java

One-sentence answer (memorize this)
A DTO is used to safely transfer data between layers, especially from client requests, without exposing internal entities or database structure.

```
public class BookingRequest {
    private String lastname;
    private String firstname;
    private String passportNumber;
    private LocalDate birthdate;
    private String flightNumber;
    private SeatType typeOfSeat;
    // ... getters and setters
}
```

Why use DTOs instead of entities directly?

1. **Decoupling**: The client doesn't need to know our entity structure
2. **Security**: We control exactly what data comes in and goes out
3. **Flexibility**: We can accept different data than what we store

For example, our BookingRequest matches the project requirement:

> 'Allow a customer to book a flight by providing lastname, firstname, passport number, birthdate, flight number...'

The service then converts this DTO into the actual entities (Client, Booking, MilesReward)."

Why do we NEED DTOs?

❌ Problem without DTOs (very common beginner mistake)

If you used Booking directly in your controller:
@PostMapping("/bookings")
public Booking createBooking(@RequestBody Booking booking

This is dangerous because:
Client can send anything
Client can set id, bookingDate, client, flight manually
Security risk
Tight coupling between API and database

🚨 ~~Never expose entities directly in APIs~~

3️⃣ What your BookingRequest DTO does

Your DTO represents:

The data a user must provide to book a flight

Not:
How data is stored
How entities are related
Internal IDs

Fields breakdown (WHY each exists)
\
private String firstname;
private String lastname;
private String email;

private String passportNumber;
private LocalDate birthdate;

➡ User identification
➡ Used to find or create a Client
private String flightNumber;

➡ Enough to identify a flight
➡ No need to expose the whole Flight entity
private SeatType typeOfSeat;

➡ User choice
➡ Controlled via enum (safe)

Fields you do NOT expose (on purpose)
❌ idReservation
❌ bookingDate
❌ MilesReward
❌ Internal DB IDs
✔ These are system-managed

PART 9: EXCEPTION HANDLING (2 minutes)

"We created custom exceptions for better error handling.

ResourceNotFoundException.java

```java
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {
        super(String.format("%s not found with %s: '%s'", resourceName, fieldName, fieldValue));
    }
}
```

@ResponseStatus(HttpStatus.NOT_FOUND) When this exception is thrown, Spring automatically returns HTTP 404.

GlobalExceptionHandler.java

```java
@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", 404);
        body.put("error", "Not Found");
        body.put("message", ex.getMessage());
        body.put("path", request.getDescription(false));

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

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

@ExceptionHandler(ResourceNotFoundException.class) When a `ResourceNotFoundException` is thrown anywhere in the application, this method catches it and returns a nice JSON error response:

json

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

This is much more professional than showing a stack trace!"

PART 10: DEMONSTRATION (5 minutes)

"Now let me demonstrate the application. I'll use Postman to test the APIs.

Step 1: Create a Plane

```
POST http://localhost:8084/api/v1/planes
{
  "brand": "Boeing",
  "model": "737",
  "manufacturingYear": 2020
}
```

Show the response with the generated ID

Step 2: Create Airports

```
POST http://localhost:8084/api/v1/airports
{
  "nameAirport": "Charles de Gaulle",
  "countryAirport": "France",
  "cityAirport": "Paris"
}
```

Create Paris and London airports

Step 3: Create a Flight

```
POST http://localhost:8084/api/v1/flights
{
```

```
  "flightNumber": "TA101",
  "departureCity": "Paris",
  "arrivalCity": "London",
  "numberOfSeat": 150,
  "plane": {"idPlane": 1},
  "departureAirport": {"idAirport": 1},
  "arrivalAirport": {"idAirport": 2}
}
```

Show how the relationships work

Step 4: Book a Flight

```
POST http://localhost:8084/api/v1/bookings/book
{
  "firstname": "John",
  "lastname": "Doe",
  "passportNumber": "AB123456",
  "birthdate": "1990-05-15",
  "flightNumber": "TA101",
  "typeOfSeat": "BUSINESS"
}
```

Show the booking response

Step 5: Check Miles Rewards

```
GET http://localhost:8084/api/v1/miles-rewards
```

Show one record without discount code

Step 6: Book Two More Times

Book twice more with the same client

Step 7: Show Discount Code

```
GET http://localhost:8084/api/v1/miles-rewards
```

Show the third record has a discount code like 'DISC-EDDB7CDE'

Step 8: Search Flights

```
GET http://localhost:8084/api/v1/flights/search?from=Paris&to=London&date=2025-01-15
```

```
GET http://localhost:8084/api/v1/flights/search?from=Paris&to=London&date=2025-01-15
```

Show the search results

Step 9: Check Available Seats

```
GET http://localhost:8084/api/v1/bookings/available-seats/TA101
```

Show: 'Available seats: 147' (150 - 3 bookings)"

PART 11: CONCLUSION (1 minute)

"To summarize what we built:

Technical Achievements:

- A complete REST API with Spring Boot
- MySQL database with JPA/Hibernate
- Proper layered architecture following SOLID principles
- Custom exception handling
- Business logic for seat availability and reward system

Project Features:

- Full CRUD for all entities
- Flight search functionality
- Smart booking that creates new customers automatically
- Miles reward system with automatic discount code generation
- Overbooking prevention

What I Learned:

- How to design a REST API
- How JPA maps objects to database tables
- How Spring's dependency injection works
- How to structure a real-world application

Thank you for your attention. Do you have any questions?"

BONUS: COMMON QUESTIONS & ANSWERS

Q: Why did you choose Spring Boot?

"Spring Boot simplifies configuration. Without it, we'd spend hours configuring XML files. With Spring Boot, we add dependencies and it auto-configures everything. Plus, it includes an embedded Tomcat server so we don't need to deploy to a separate server."

Q: Why MySQL and not another database?

"MySQL is widely used in industry, it's open source, and it works well with JPA. We could easily switch to PostgreSQL by just changing the driver and connection URL - that's the benefit of using JPA as an abstraction layer."

Q: How does the discount code generation work?

"Every time a client books a flight, we record it in the MilesReward table. We count how many flights they've taken this calendar year. If that number is divisible by 3, we generate a random discount code using UUID. So they get a discount on their 3rd, 6th, 9th flight, and so on."

Q: What happens if two people try to book the last seat at the same time?

"Good question! In a production system, we'd need to handle this with database transactions and locking. Currently, our countByFlight check and save are not atomic, so there's a small race condition window. For a real airline, we'd use pessimistic locking or a database constraint."

Q: Why separate Service and Repository layers?

"Separation of concerns. The Repository only handles database operations. The Service contains business logic like checking seat availability and generating discount codes. This makes testing easier - we can mock the Repository and test the Service logic independently."

CHEAT SHEET: QUICK REFERENCE

Annotations You Must Know

Annotation	Layer	Purpose
`@SpringBootApplication`	Main	Enables everything
`@Entity`	Model	Class = Table

`@Id`	Model	Primary key
`@GeneratedValue`	Model	Auto-increment
`@ManyToOne`	Model	Foreign key relationship
`@Repository`	Repository	Database access
`@Service`	Service	Business logic
`@Autowired`	Service/Controller	Dependency injection
`@RestController`	Controller	REST API handler
`@GetMapping`	Controller	Handle GET requests
`@PostMapping`	Controller	Handle POST requests
`@RequestBody`	Controller	JSON → Object
`@PathVariable`	Controller	URL path → variable
`@RequestParam`	Controller	Query string → variable

Request Flow

```
HTTP Request → Controller → Service → Repository → Database
HTTP Response ← Controller ← Service ← Repository ← Database
```

Good luck with your presentation! 🎉