1. Fetch Plans and Strategies

In my implementation, **EAGER fetching** was used for relationships that are fundamental to the identity of the parent entity or always required by endpoints. For example, a booking has no meaning without its associated flight, so **Booking → Flight** was set to @ManyToOne(fetch = FetchType.EAGER) to ensure complete flight details (airports, times, aircraft) are always loaded together, avoiding N+1 query problems when displaying multiple bookings *(see model/Booking.java, lines 21–22)*. Similarly, **Booking → User** is eagerly loaded since user information is always required for authorization and context when displaying or cancelling bookings *(lines 18-19)*. Within the Flight entity, the **origin, destination, and aircraftType** relationships are all eagerly fetched *(see model/Flight.java, lines 29–36)*, as every endpoint returning flight data (e.g., /api/flights, /api/bookings) requires these fields. This ensures that the route ("Auckland → Sydney") and aircraft information are available without triggering additional queries, improving performance and simplifying serialization.

Conversely, LAZY fetching was applied where data is rarely needed. **Flight → Bookings** is lazy because bookings aren't required when searching or displaying flights; loading them would create unnecessary overhead for flights with many passengers (model/Flight.java, lines 56–58). **User → Bookings (**model/User.java, lines 34-36**)** is also lazy since bookings are never accessed via the User entity; they're always queried directly through BookingRepository with custom @Query methods (repository/BookingRepository.java). This strategy minimizes unnecessary queries while ensuring essential data is available in end-points that are hit frequently.

2. Domain and DTO classes

Core domain entities (Flight, Airport, AircraftType, SeatingZone, Booking, User) were designed to align directly with API requirements, minimizing DTO use. Flight serializes cleanly since it includes departure/arrival times, route, aircraft type, and status. Airport and AircraftType act as value objects containing display data such as location, timezone, and seating configuration. SeatingZone is embedded within AircraftType to define cabin layout and serializes naturally.

A key challenge with bidirectional JPA relationships was preventing circular references during JSON serialization. Without intervention, serializing a booking would trigger infinite loops: Flight **→ Booking → Flight → Booking → … and User → Booking → User → Booking → …,** ultimately causing stack overflow errors. This was resolved by applying **@JsonIgnore** to the reverse sides of these relationships (**Flight.bookings and User.bookings at model/Flight.java, line 57 and model/User.java, line 35**). This ensures bookings can reference their flight and user, but flights and users don't serialize their entire booking collections, breaking the cycle cleanly.

DTOs were introduced where the API requirements did not match the domain persistence model. **UserDTO** handles authentication with only username and password. **BookingRequestDTO** captures booking input (flightId, requestedSeats). **BookingResponseDTO** adds computed totalCost, dynamically calculated from seat positions and pricing. **BookingInfoDTO** consists of aircraft details, pricing, and booked seats for the booking-info endpoint that cannot be mapped to any of the domain models. This separation maintains a clean domain model while ensuring REST endpoints deliver precisely structured responses.

3. Authentication

Authentication was implemented using a custom **@AuthenticatedUser** annotation and **HandlerMethodArgumentResolver** to automatically inject authenticated users into controller methods. This centralizes authentication logic and eliminates manual JWT validation in each endpoint.https://medium.com/@AlexanderObregon/how-spring-boot-configures-custom-argument-resolvers-ed4833420549, i used this article for reference.

The **@AuthenticatedUser** annotation marks controller parameters requiring authentication (auth/AuthenticatedUser.java) **CustomArgumentResolver** class implements **HandlerMethodArgumentResolver** (auth/CustomArgumentResolver.java) with two key methods: supportsParameter() checks if a parameter has @AuthenticatedUser and is of type User (lines 38-42); resolveArgument() handles the authentication flow:

1. Extract JWT from authToken cookie (line 63)

2. Decode token using SecurityUtils.decodeJWTFromCookies() (line 64)

3. Retrieve username claim from JWT (line 71)

4. Load User entity from UserRepository (line 74)

5. Return User if valid, throw 401 Unauthorized if not (lines 76,77)

The resolver is registered via CustomArgumentResolverConfig through WebMvcConfigurer (auth/CustomArgumentResolverConfig.java, lines 14-29), ensuring Spring automatically applies it to matching parameters.

To protect a new endpoint, developers simply add an annotated parameter:

@GetMapping("/api/booking")

public ResponseEntity<?> example(@AuthenticatedUser User user)

{ // User automatically injected or 401 returned }

This declarative approach provides seamless, consistent authentication across all controllers without boilerplate code.

4. Flight Search

Flight search was implemented using a hybrid approach combining JPQL for origin/destination matching and Java-based post-filtering for timezone-aware date ranges.

The repository uses a custom **@Query** with case-insensitive wildcard searches on both airport names and codes (repository/FlightRepository.java, lines 15-20). The query uses LOWER() and LIKE with CONCAT('%', :param, '%') to match partial strings, allowing users to search "Auckland" or "AKL" interchangeably. ORDER BY f.departureTime ASC ensures

results are chronologically sorted, which is critical for displaying flights consistently and supporting the booking list ordering requirements.

When departureDate is supplied, timezone handling occurs in the controller (controller/FlightController.java, lines 70-80). Each flight's origin timezone is used with **TimeZoneUtils.toZonedDateRange()** to generate an accurate ZonedDateTime range. For example, searching "August 11" for Auckland flights means August 11 in Pacific/Auckland timezone, not UTC. Flights are filtered where departureTime falls within this range, with dayRange extending the window by ±N days.

This post-filtering approach was chosen over pure JPQL because each flight has a different origin timezone, making SQL-based timezone conversion complex and database-specific. The hybrid approach balances efficiency (JPQL narrows the dataset) with correctness (Java ensures precise timezone-aware matching).

5.Generative AI

The Claude Sonnet 4.5 model (Anthropic, via claude.ai) was used as a learning tool throughout the assignment. Instead of supplying code directly, it clarified Spring concepts, explained architectural decisions, and assisted with debugging. It was especially helpful for understanding `HandlerMethodArgumentResolver`–based authentication and resolving circular JSON reference issues.

Claude's greatest strength was ideation and deep explanation , it excelled at generating ideas and thoroughly explaining them. For instance, when addressing infinite recursion during entity serialization, it compared alternative approaches (`@JsonIgnore`, `@JsonIdentityInfo`, and DTOs) and helped identify the simplest, most appropriate fix.

At times, Claude suggested overly advanced or context-misaligned solutions (such as full Spring Security integration), which I was not yet familiar with. Overall, it transformed the project from a straightforward coding exercise into a valuable learning opportunity in software design, helping me understand not only what works, but also why, and how to evaluate multiple solutions before selecting the best one.