



## Flight Booking System (Searching + Booking) Design

-  The document outlines the design and implementation approach for a flight search and booking system. It includes high-level and low-level design aspects, assumptions, APIs, entities, database schema, and relevant components. The system aims to efficiently search for flight information and handle bookings reliably.

Key components include:

1. **Searching:** Finds the cheapest/fastest flights and stores results in cache.
2. **Inventory Management:** Maintains flight database and inventory.
3. **Flight Booking:** Manages the booking process.
4. **Payments:** Handled as a black box.

The project structure consists of four microservices: BookingService, SearchService, PaymentService, and InventoryService. The document details the flight search and booking processes, including APIs for searching flights, managing inventory, and handling bookings, along with considerations for concurrency and failure scenarios. It also discusses the tech stack used, including Java, Spring Boot, PostgreSQL, and Redis.

 Pre requisite: [Flight Booking System \(Searching + Booking\) PRD](#)

### Problem statement:

- Given a source and a destination along with some filters, need find the cheapest/fastest flights. Result should contain List of X flights.
- After showing the result user will opt in for booking a flight. Booking a flight contains reserving the number of seats, making the payment, updating the inventory.
- Also take care of concurrency and failure scenarios. In case of failure the inventory should not be affected. If the seats are blocked it should be unblocked. For concurrency at a time 2 users should not modify the inventory

### Assumptions:

- For output of flight search we are currently showing only top 10 results.
- To limit our search space we are keeping only 50 sources+destinations. Basically 50c2 combination for source and destination

- Cost and duration are as per the ease of calculation does not reflect the exact value
- Max two stops are allowed to go from source to destination
- Source to Destination with one stop is considered to have 2 flights.
  - For example Ahmedabad(S) → Mumbai is Flight1 & Mumbai → Bengaluru(D) is Flight2
- For booking a flight we are considering only single flight (direct flight)
- Each and every data is present in cache for searching. If it is not present in cache we are not doing DB look up.
- We store data in cache for today and the next 180 days (inclusive of today).
- Payment is a black box. We are not integrating any PG or not calling any third party to make the payment.

#### Project Structure:

- We will have four micro services. BookingService, SearchService, PaymentService and InventoryService.
- Since we are maintaining the single repository we will have some rough structure like below.

```

1  flight-booking-system/
2  |  └─ booking-service/
3  |      └─ Dockerfile
4  |      └─ pom.xml
5  |      └─ src/main/java/com/fbs/
6  |          └─ BookingServiceApplication.java
7  |
8  |  └─ search-service/
9  |      └─ Dockerfile
10 |      └─ pom.xml
11 |      └─ src/main/java/com/fbs/
12 |          └─ SearchServiceApplication.java
13 |
14 |  └─ inventory-service/
15 |      └─ Dockerfile
16 |      └─ pom.xml
17 |      └─ src/main/java/com/fbs/
18 |          └─ InventoryServiceApplication.java
19 |
20 |  └─ payment-service/
21 |      └─ Dockerfile
22 |      └─ pom.xml
23 |      └─ src/main/java/com/fbs/
24 |          └─ PaymentServiceApplication.java
25 |
26 |  └─ fbs-commons/
27 |      └─ pom.xml
28 |
29 |  └─ infra/
30 |      └─ docker-compose.yml
31 |
32 |  └─ scripts/
33 |

```

```
34 |— pom.xml (parent)
35 |— README.md
```

- Let's break down problem statement into two parts
  - FLIGHT SEARCHING
  - FLIGHT BOOKING

### Flight Search:

- according to our assumptions there will be limited number of airports.
- Searching should be fast. So let's estimate the size of search space
- Based on our assumptions, the number of airports is limited.
  - Total source–destination pairs: **50**
  - Possible combinations:  **$50C2 \approx 2,500$**
  - Cache duration: **180 days ( $\approx$  6 months, inclusive of today)**
  - Total cache entries:  **$2,500 \times 180 \approx 500K$**

### Redis Key Format:

```
<source>+<destination>+<date>
```

### Value Structure:

A list of flight paths with up to **2 stops**:

- `{f1}` , `{f1, f2}` , `{f1, f2, f3}`
- Max 50 elements per list
- Each element = UUID (32 bytes)

### Memory Calculation:

- $50 \times 32B = \sim 1.5KB$  per list
- With 2 stops:  $\sim 3KB$  per entry
- Total:  $500K \times 3KB \approx$  **1.5 GB RAM**
- 1.5 GB of RAM is not a huge memory. We can pre compute all the entries once for all and store them in Redis while booting up the search-service.
- Now let's see how our flight entity will look like.

- ```
1 public class Flight {
2     @Id
3     @GeneratedValue(strategy = GenerationType.UUID)
4     private UUID flightId;           // Primary key
5
6     private String flightNumber;     // e.g., "AI101"
7     private String source;           // 3-letter airport code (e.g., "AMD")
8     private String destination;     // 3-letter airport code (e.g., "BLR")
9     private Double cost;             // Flight cost in currency units
```

```

10     private Integer duration;           // Duration in minutes
11     private Integer availableSeats;     // Available seats
12     private Integer occupiedSeats;     // Occupied seats
13     private FlightStatus flightStatus; // Current flight status
14     private LocalTime departureTime;    // Departure time
15     private LocalTime arrivalTime;     // Arrival time
16     private LocalDateTime createdAt;    // Record creation timestamp
17     private LocalDateTime updatedAt;    // Last update timestamp
18 }

```

- To start with we can run a python script to populate the data in DB.
- To find the top 10 cheapest/fastest path between source and destination we will run an algorithm which will give us the result.

APIs for Flight Search:

### 1. Search Flights

- **Endpoint:** `GET /v1/search`
- **Description:** Find optimal flight paths between cities.
- **Parameters:**
  - `SOURCE` (string, required): Source airport code (3-letter)
  - `destination` (string, required): Destination airport code (3-letter)
  - `date` (string, required): Search date (YYYY-MM-DD format)
  - `criteria` (string, required): "CHEAPEST" or "FASTEST"
- **Responses:**
  - `200 OK` : Flight paths found
  - `204 No Content` : No flights available
  - `400 Bad Request` : Invalid parameters

### 2. Health Check

- **Endpoint:** `GET /v1/health`
- **Description:** Service health status.
- **Response:** `200 OK` : "Search Service is running"

Inventory Service APIs:

### 1. Get All Flights

- **Endpoint:** `GET /v1/flights/all`
- **Description:** Retrieve all flight records (used for pre-computation).
- **Response:** `200 OK` : Array of all flights

## 2. Get Flight by ID

- **Endpoint:** `GET /v1/flights/{flightId}`
- **Description:** Get specific flight details.
- **Parameters:**
  - `flightId` (UUID, path): Flight identifier
- **Responses:**
  - `200 OK` : Flight details
  - `404 Not Found` : Flight not found

## 3. Get Flights by Route

- **Endpoint:** `GET /v1/flights/route`
- **Description:** Get flights for specific route.
- **Parameters:**
  - `source` (string, query): Source airport code
  - `destination` (string, query): Destination airport code
- **Response:** `200 OK` : Array of flights for route

## 4. Get Flight Status

- **Endpoint:** `GET /v1/flights/{flightId}/status`
- **Description:** Get current flight status.
- **Response:** `200 OK` : Flight status information

## 5. Health Check

- **Endpoint:** `GET /v1/health`
- **Description:** Service health status.
- **Response:** `200 OK` : "Inventory Service is running"

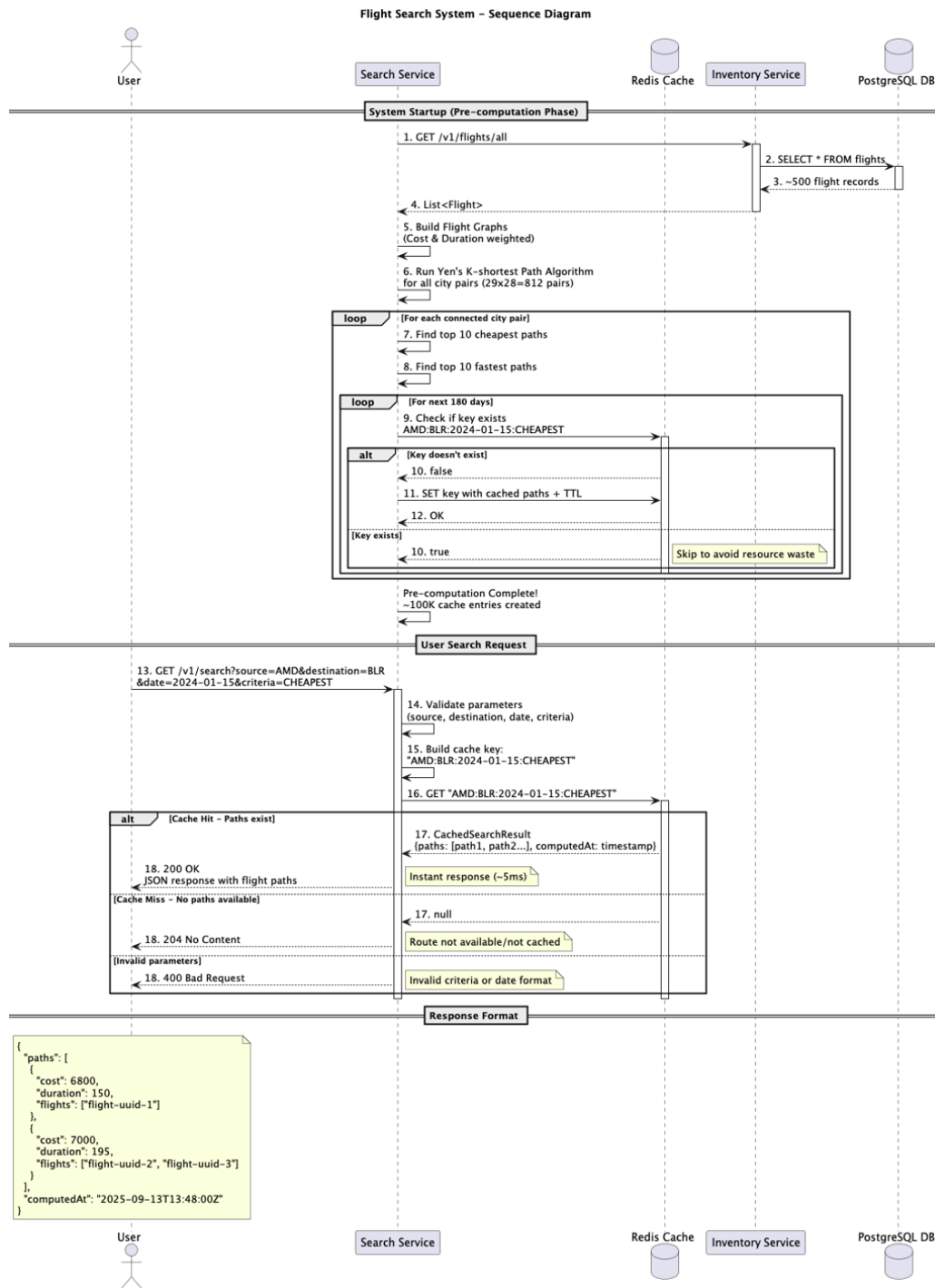
Trade-offs:

1. PostgreSQL for Inventory Service as well as Booking service.

| Criteria           | PostgreSQL<br>(Chosen) | MongoDB                 | Redis   |
|--------------------|------------------------|-------------------------|---------|
| ACID<br>Compliance | ACID support           | Limited<br>transactions | No ACID |

|                     |                          |                        |                        |
|---------------------|--------------------------|------------------------|------------------------|
| Data Consistency    | ✓ Strong consistency     | ⚠ Eventual consistency | ✓ Strong (single node) |
| Query Complexity    | ✓ Complex SQL queries    | ⚠ Limited aggregation  | ✗ Simple key-value     |
| Performance (Read)  | ✓ Excellent with indexes | ✓ Good                 | ✓ Excellent            |
| Performance (Write) | ⚠ ACID overhead          | ✓ Fast writes          | ✓ Ultra fast           |
| Memory Usage        | ⚠ Moderate               | ⚠ Moderate             | ✗ High (in-memory)     |
| Cost                | ⚠ Moderate               | ✓ Lower                | ✗ High (memory)        |

## Low level sequence diagram for flight searching:



## Flight Booking:

- According to our assumptions we will book only direct flights.
- After searching flights user shows intent to book the flight.
- There are few edge cases that needs to handled gracefully.
  - User drops off in journey.
  - Payment got failed (need to unblock booked seats)

- In parallel two user try to book the same flight (concurrency night mare)
- Blockage of resource (Booked seats never got released in case of failure)

#### APIs for Flight Booking

### Booking Service APIs

#### Create Booking

- **Endpoint:** `POST /v1/bookings`
- **Description:** Create a new flight booking with atomic seat reservation.
- **Parameters:**
  - `flightId` (UUID, required): Flight identifier
  - `customerEmail` (string, required): Customer email address
  - `numberOfSeats` (integer, required): Number of seats to book (min: 1)
- **Responses:**
  - `200 OK` : Booking created successfully
  - `400 Bad Request` : Invalid request parameters
  - `500 Internal Server Error` : Booking failed (insufficient seats/payment issues)

#### Get Booking by ID

- **Endpoint:** `GET /v1/bookings/{bookingId}`
- **Description:** Retrieve booking details by booking ID.
- **Parameters:**
  - `bookingId` (UUID, required): Booking identifier
- **Responses:**
  - `200 OK` : Booking details found
  - `404 Not Found` : Booking not found

#### Get Bookings by Customer

- **Endpoint:** `GET /v1/bookings?customerEmail={email}`
- **Description:** Retrieve all bookings for a customer.
- **Parameters:**
  - `customerEmail` (string, required): Customer email address



- **Responses:**

- `200 OK` : List of customer bookings

## Health Check

- **Endpoint:** `GET /health`
- **Description:** Service health status.
- **Response:** `200 OK` : "Booking Service is running"

## Payment Service APIs

### Process Payment

- **Endpoint:** `POST /v1/payments/process`
- **Description:** Process payment for booking (configurable simulator).
- **Parameters:**
  - `bookingId` (UUID, required): Booking identifier
  - `amount` (decimal, required): Payment amount
  - `currency` (string, required): Currency code (e.g., "USD")
  - `customerEmail` (string, required): Customer email
- **Responses:**
  - `200 OK` : Payment processed (success/failure based on configuration)

## Health Check

- **Endpoint:** `GET /health`
- **Description:** Service health status.
- **Response:** `200 OK` : "Payment Service is running"

## Inventory Service APIs (New Booking-Related)

### Reserve Seats

- **Endpoint:** `POST /v1/flights/{flightId}/reserve-seats?numberOfSeats={seats}`
- **Description:** Atomically reserve seats for a flight.
- **Parameters:**
  - `flightId` (UUID, required): Flight identifier
  - `numberOfSeats` (integer, required): Number of seats to reserve

- **Responses:**

- `200 OK` : true if seats reserved, false if insufficient seats

## Release Seats

- **Endpoint:** `POST /v1/flights/{flightId}/release-seats?`

`numberOfSeats={seats}`

- **Description:** Release previously reserved seats (rollback operation).

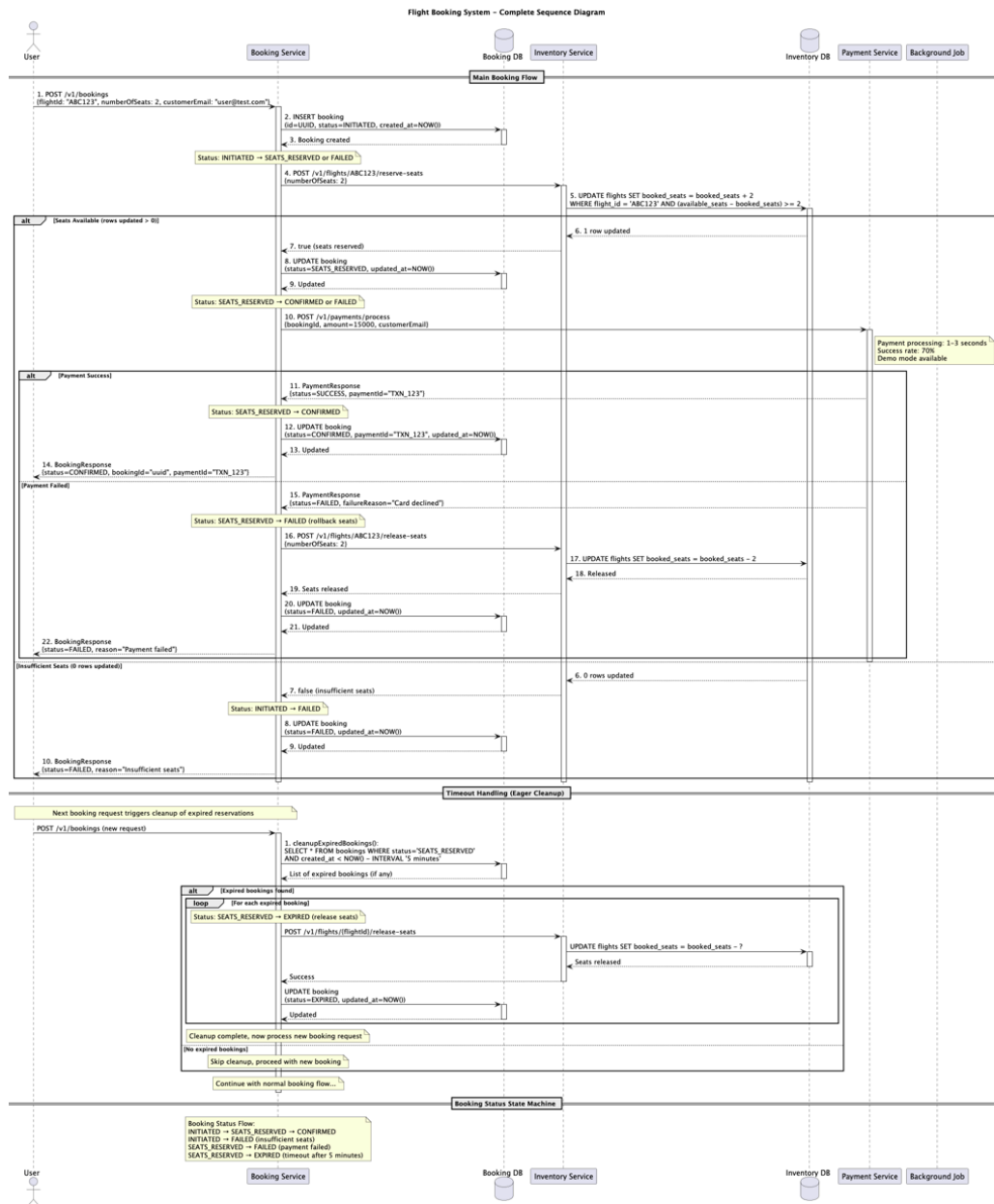
- **Parameters:**

- `flightId` (UUID, required): Flight identifier
- `numberOfSeats` (integer, required): Number of seats to release

- **Responses:**

- `200 OK` : Seats released successfully

## Low level sequence diagram for Flight Booking:

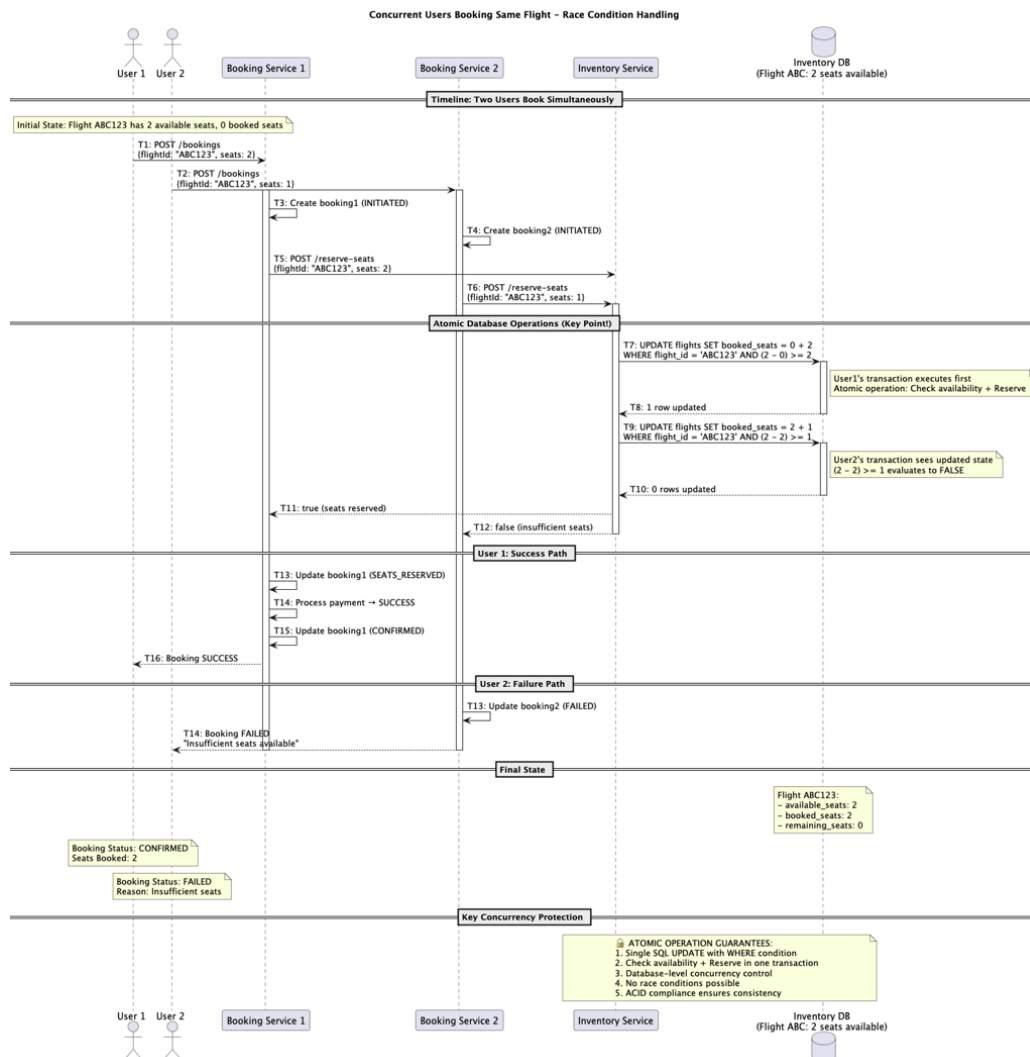


### How to handle timeout?

- When user reserve the seat, after some period of time user has to make the payment because he is holding the seats.
- There are various way to solve this
  - Schedule background Job which will check every minute or every 5 minutes (configurable). Check the DB for status: SEATS\_RESERVED and last updated is now - 5 (time out timer) minutes. Then mark status failed for such bookings and release the seats
  - Timer per Booking (Async Thread sleep) with CompletableFuture. Major red flag for this approach is One thread per booking sleeping for 5 minutes. Can consume all available threads

- Lazy cleanup, whenever new request comes for any booking check status and created before 5 minutes. And do the releasing seat task. **(Going with this)**
- For now we are going with lazy cleanup and next phase can go for Cron Job (Tech Debt)

How to handle concurrency?

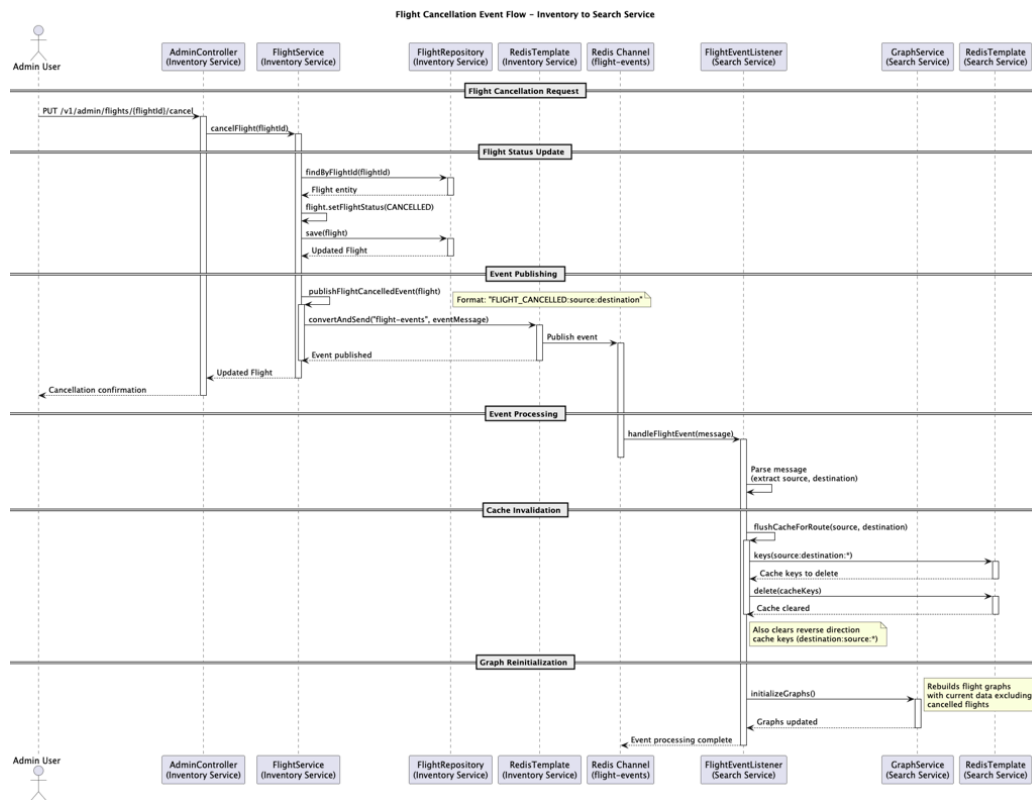


## Event publisher on flight cancelation:

Scope:

- Admin can cancel the flight
- When an admin cancels a flight, the **inventory-service** emits an event. The **search-service** listens to this event, flushes cache entries related to the affected source or destination, and recomputes the keys.

*(Optimization planned as tech debt for the next phase.)*



## Tech Stack:

### Programming Languages:

- Java - Main application development (Spring Boot microservices)
- Python - Data seeding scripts

### Framework:

- Spring Boot - Backend framework for all microservices

### API Protocol:

- REST - API communication protocol

### Database & Storage:

- PostgreSQL - Primary database for Flight and Booking data storage
- Redis - Caching layer for search results

### Containerisation:

- Docker - Application containerization
- Docker Compose - Multi-service orchestration

### Architecture:

- Microservices - Distributed system architecture
- Service-to-Service Communication - REST APIs between services

