## **Introduction**

### This case study focuses on developing an **Airline Reservation System** using **microservices architecture**. The system will be divided into 7 independent microservices, each responsible for distinct business functionalities. The architecture will leverage **Reactive Programming**, **MongoDB**, **Circuit Breakers**, **Service Discovery**, and **API Gateways** while ensuring proper **monitoring and logging** through the **ELK stack**. Furthermore, **Docker** and **Kubernetes** will be utilized for containerization and orchestration.

### 

### **System Overview**

### The **Airline Reservation System** will cover the following functionalities through different microservices:

### **Flight Search**: Allows users to search for available flights.

### **Booking**: Handles flight bookings and ticket issuance.

### **Payment**: Manages payment processing.

### **User Management**: Manages user information, registration, and authentication.

### **Notification**: Sends booking and payment notifications to users.

### **Pricing**: Calculates fares based on flight details and other variables.

### **Inventory Management**: Manages the availability of seats across flights.

### Each microservice will be loosely coupled and communicate through APIs, using **REST**, **gRPC**, or **message queues** for inter-service communication. They will be reactive and non-blocking, ensuring scalability and high performance.

### 

## **Microservices Architecture**

### **1. Flight Search Microservice**

### **Purpose**: Handles flight search functionality. It queries available flights from the **Inventory** and filters results based on user inputs (departure, arrival, date).

### **Challenges Addressed**:

### **Reactive Programming**: Efficient handling of asynchronous flight search results.

### **Data Consistency**: Data consistency is maintained with **eventual consistency** models.

### **Communication**: Interacts with the **Inventory Management** microservice using **gRPC**.

### **2. Booking Microservice**

### **Purpose**: Manages flight bookings, confirms availability, and issues tickets.

### **Challenges Addressed**:

### **Saga Pattern**: For maintaining consistency during the multi-step booking process (reserving a seat, charging the customer).

### **CQRS Pattern**: Used to separate the read and write operations in the booking process.

### **Circuit Breakers**: Implemented to handle failure gracefully during booking confirmation.

### **3. Payment Microservice**

### **Purpose**: Handles payment processing for bookings.

### **Challenges Addressed**:

### **Circuit Breakers**: Used to handle failures in payment gateways, ensuring fallback mechanisms.

### **Retries**: Implemented to handle transient payment failures.

### **API Gateway Integration**: Security is enforced using the **API Gateway**, ensuring that only authenticated users can access payment services.

### **4. User Management Microservice**

### **Purpose**: Manages user profiles, registration, login, and authentication.

### **Challenges Addressed**:

### **JWT Authentication**: Uses **JSON Web Tokens (JWT)** for secure API access.

### **Reactive Programming**: Provides a non-blocking user experience, ensuring fast response times.

### **Security via API Gateway**: User data is protected via the API Gateway’s security features, including rate limiting and authentication.

### **5. Notification Microservice**

### **Purpose**: Sends email and SMS notifications to users on booking and payment status.

### **Challenges Addressed**:

### **Event-Driven Architecture**: Receives events (booked flights, payment success) from other services and triggers notifications asynchronously.

### **Message Queues**: Uses message queues like **RabbitMQ** for handling notification messages efficiently.

### **6. Pricing Microservice**

### **Purpose**: Calculates the cost of tickets dynamically based on various factors such as demand, availability, and promotions.

### **Challenges Addressed**:

### **Reactive Streams**: Real-time price calculation based on dynamic variables.

### **Consistency Models**: Ensures strong consistency for pricing calculations.

### **7. Inventory Management Microservice**

### **Purpose**: Manages flight seat availability and ensures data consistency for available seats across flights.

### **Challenges Addressed**:

### **Sharding and Replication**: Uses **MongoDB** for managing seat availability with horizontal scaling via sharding.

### **Reactive Data Handling**: Efficient management of inventory using reactive data streams for consistent performance under load.

### 

### 

### **1. Flight Search Microservice**

* **GET /api/flights/search**
  + **Description:** Search for available flights based on criteria.

**Request Payload:**json  
Copy code  
{

"departureCity": "New York",

"arrivalCity": "Los Angeles",

"departureDate": "2024-12-15",

"returnDate": "2024-12-20",

"passengers": {

"adults": 1,

"children": 0

},

"class": "Economy"

}

**Response Payload:**json  
Copy code  
{

"flights": [

{

"flightId": "FL123",

"airline": "Airways A",

"departureCity": "New York",

"arrivalCity": "Los Angeles",

"departureDate": "2024-12-15T10:00:00Z",

"arrivalDate": "2024-12-15T13:00:00Z",

"price": {

"amount": 300,

"currency": "USD"

},

"seatsAvailable": 50

}

]

}

* **GET /api/flights/{id}**
  + **Description:** Retrieve flight details by ID.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"airline": "Airways A",

"departureCity": "New York",

"arrivalCity": "Los Angeles",

"departureDate": "2024-12-15T10:00:00Z",

"arrivalDate": "2024-12-15T13:00:00Z",

"price": {

"amount": 300,

"currency": "USD"

},

"seatsAvailable": 50,

"aircraft": {

"model": "Boeing 737",

"capacity": 180

},

"amenities": [

"Wi-Fi",

"In-flight entertainment",

"Meal service"

]

}

* **POST /api/flights**
  + **Description:** Create a new flight entry.

**Request Payload:**json  
Copy code  
{

"airline": "Airways A",

"departureCity": "New York",

"arrivalCity": "Los Angeles",

"departureDate": "2024-12-15T10:00:00Z",

"arrivalDate": "2024-12-15T13:00:00Z",

"price": {

"amount": 300,

"currency": "USD"

},

"seatsAvailable": 100

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Created"

}

* **PUT /api/flights/{id}**
  + **Description:** Update flight details by ID.

**Request Payload:**json  
Copy code  
{

"price": {

"amount": 320,

"currency": "USD"

},

"seatsAvailable": 90

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Updated"

}

* **DELETE /api/flights/{id}**
  + **Description:** Delete a flight entry by ID.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Deleted"

}

* **GET /api/flights/availability/{id}**
  + **Description:** Retrieve flight seat availability.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"seatsAvailable": 50,

"seatMap": [

{"row": 1, "seats": 6},

{"row": 2, "seats": 6}

]

}

* **GET /api/flights/price/{id}**
  + **Description:** Retrieve the price of a specific flight.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"basePrice": 250,

"taxes": 50,

"totalPrice": 300,

"currency": "USD"

}

### **2. Booking Microservice**

* **POST /api/bookings**
  + **Description:** Create a new booking.

**Request Payload:**json  
Copy code  
{

"userId": "user123",

"flightId": "FL123",

"passengers": [

{

"name": "John Doe",

"age": 30,

"passportNumber": "X1234567"

}

],

"contactDetails": {

"email": "john.doe@example.com",

"phone": "+1234567890"

},

"payment": {

"amount": 300,

"currency": "USD",

"paymentMethod": "credit\_card",

"cardDetails": {

"cardNumber": "4111111111111111",

"expiryDate": "2025-12",

"cvv": "123"

}

}

}

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"flightId": "FL123",

"userId": "user123",

"status": "Confirmed",

"totalPrice": {

"amount": 300,

"currency": "USD"

},

"departureDate": "2024-12-15T10:00:00Z",

"arrivalDate": "2024-12-15T13:00:00Z",

"passengers": [

{

"name": "John Doe",

"age": 30

}

]

}

* **GET /api/bookings/{id}**
  + **Description:** Retrieve booking details by ID.

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"flight": {

"flightId": "FL123",

"departureCity": "New York",

"arrivalCity": "Los Angeles",

"departureDate": "2024-12-15T10:00:00Z",

"arrivalDate": "2024-12-15T13:00:00Z"

},

"user": {

"userId": "user123",

"name": "John Doe",

"contactDetails": {

"email": "john.doe@example.com",

"phone": "+1234567890"

}

},

"passengers": [

{

"name": "John Doe",

"age": 30

}

],

"totalPrice": {

"amount": 300,

"currency": "USD"

},

"status": "Confirmed"

}

* **PUT /api/bookings/{id}**
  + **Description:** Update booking details by ID.

**Request Payload:**json  
Copy code  
{

"passengers": [

{

"name": "John Doe",

"age": 31

}

],

"contactDetails": {

"email": "john.doe@newemail.com",

"phone": "+0987654321"

}

}

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"status": "Updated"

}

* **DELETE /api/bookings/{id}**
  + **Description:** Cancel a booking by ID.

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"status": "Cancelled"

}

* **GET /api/bookings/user/{userId}**
  + **Description:** Retrieve all bookings for a user.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"bookings": [

{

"bookingId": "BK123456",

"flightId": "FL123",

"status": "Confirmed"

}

]

}

* **POST /api/bookings/{id}/add-passenger**
  + **Description:** Add a new passenger to an existing booking.

**Request Payload:**json  
Copy code  
{

"passenger": {

"name": "Jane Doe",

"age": 28,

"passportNumber": "X7654321"

}

}

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"status": "Passenger Added"

}

* **POST /api/bookings/{id}/update-payment**
  + **Description:** Update payment details for a booking.

**Request Payload:**json  
Copy code  
{

"paymentMethod": "credit\_card",

"cardDetails": {

"cardNumber": "4111111111111112",

"expiryDate": "2026-01",

"cvv": "124"

}

}

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"status": "Payment Updated"

}

### **3. Payment Microservice**

* **POST /api/payments**
  + **Description:** Process a new payment.

**Request Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"amount": 300,

"currency": "USD",

"paymentMethod": "credit\_card",

"cardDetails": {

"cardNumber": "4111111111111111",

"expiryDate": "2025-12",

"cvv": "123"

}

}

**Response Payload:**json  
Copy code  
{

"paymentId": "PY123456",

"status": "Processed",

"amount": 300,

"currency": "USD"

}

* **GET /api/payments/{id}**
  + **Description:** Retrieve payment details by ID.

**Response Payload:**json  
Copy code  
{

"paymentId": "PY123456",

"bookingId": "BK123456",

"amount": 300,

"currency": "USD",

"status": "Processed",

"paymentMethod": "credit\_card"

}

* **PUT /api/payments/{id}**
  + **Description:** Update payment details by ID.

**Request Payload:**json  
Copy code  
{

"amount": 320,

"currency": "USD"

}

**Response Payload:**json  
Copy code  
{

"paymentId": "PY123456",

"status": "Updated"

}

* **DELETE /api/payments/{id}**
  + **Description:** Delete a payment record by ID.

**Response Payload:**json  
Copy code  
{

"paymentId": "PY123456",

"status": "Deleted"

}

* **GET /api/payments/booking/{bookingId}**
  + **Description:** Retrieve all payments for a booking.

**Response Payload:**json  
Copy code  
{

"bookingId": "BK123456",

"payments": [

{

"paymentId": "PY123456",

"amount": 300,

"currency": "USD",

"status": "Processed"

}

]

}

* **GET /api/payments/user/{userId}**
  + **Description:** Retrieve all payments for a user.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"payments": [

{

"paymentId": "PY123456",

"amount": 300,

"currency": "USD",

"status": "Processed"

}

]

}

* **POST /api/payments/{id}/refund**
  + **Description:** Process a refund for a payment.

**Request Payload:**json  
Copy code  
{

"amount": 150,

"currency": "USD",

"refundReason": "Flight canceled"

}

**Response Payload:**json  
Copy code  
{

"paymentId": "PY123456",

"status": "Refunded",

"refundAmount": 150,

"currency": "USD"

}

### **4. User Management Microservice**

* **POST /api/users**
  + **Description:** Register a new user.

**Request Payload:**json  
Copy code  
{

"name": "John Doe",

"email": "john.doe@example.com",

"password": "password123",

"phone": "+1234567890"

}

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"name": "John Doe",

"email": "john.doe@example.com",

"phone": "+1234567890",

"status": "Registered"

}

* **POST /api/users/login**
  + **Description:** Authenticate a user.

**Request Payload:**json  
Copy code  
{

"email": "john.doe@example.com",

"password": "password123"

}

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"token": "jwt-token",

"status": "Authenticated"

}

* **GET /api/users/{id}**
  + **Description:** Retrieve user details by ID.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"name": "John Doe",

"email": "john.doe@example.com",

"phone": "+1234567890"

}

* **PUT /api/users/{id}**
  + **Description:** Update user details by ID.

**Request Payload:**json  
Copy code  
{

"name": "John D.",

"phone": "+0987654321"

}

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"status": "Updated"

}

* **DELETE /api/users/{id}**
  + **Description:** Delete a user by ID.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"status": "Deleted"

}

* **POST /api/users/{id}/change-password**
  + **Description:** Change user password.

**Request Payload:**json  
Copy code  
{

"oldPassword": "password123",

"newPassword": "newpassword456"

}

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"status": "Password Changed"

}

* **GET /api/users/{id}/bookings**
  + **Description:** Retrieve all bookings for a user.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"bookings": [

{

"bookingId": "BK123456",

"flightId": "FL123",

"status": "Confirmed"

}

]

}

### **5. Notification Microservice**

* **POST /api/notifications**
  + **Description:** Send a notification to a user.

**Request Payload:**json  
Copy code  
{

"userId": "user123",

"message": "Your booking has been confirmed.",

"type": "BookingConfirmation"

}

**Response Payload:**json  
Copy code  
{

"notificationId": "NT123456",

"userId": "user123",

"status": "Sent"

}

* **GET /api/notifications/{id}**
  + **Description:** Retrieve notification details by ID.

**Response Payload:**json  
Copy code  
{

"notificationId": "NT123456",

"userId": "user123",

"message": "Your booking has been confirmed.",

"type": "BookingConfirmation",

"status": "Sent"

}

* **PUT /api/notifications/{id}**
  + **Description:** Update notification details by ID.

**Request Payload:**json  
Copy code  
{

"message": "Your booking has been updated.",

"status": "Updated"

}

**Response Payload:**json  
Copy code  
{

"notificationId": "NT123456",

"status": "Updated"

}

* **DELETE /api/notifications/{id}**
  + **Description:** Delete a notification by ID.

**Response Payload:**json  
Copy code  
{

"notificationId": "NT123456",

"status": "Deleted"

}

* **GET /api/notifications/user/{userId}**
  + **Description:** Retrieve all notifications for a user.

**Response Payload:**json  
Copy code  
{

"userId": "user123",

"notifications": [

{

"notificationId": "NT123456",

"message": "Your booking has been confirmed.",

"type": "BookingConfirmation",

"status": "Sent"

}

]

}

* **POST /api/notifications/{id}/resend**
  + **Description:** Resend a notification.

**Response Payload:**json  
Copy code  
{

"notificationId": "NT123456",

"status": "Resent"

}

* **GET /api/notifications/status/{status}**
  + **Description:** Retrieve notifications by status.

**Response Payload:**json  
Copy code  
{

"status": "Sent",

"notifications": [

{

"notificationId": "NT123456",

"userId": "user123",

"message": "Your booking has been confirmed."

}

]

}

### **6. Pricing Microservice**

* **GET /api/pricing/{flightId}**
  + **Description:** Retrieve pricing details for a flight.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"basePrice": 500,

"currency": "USD",

"tax": 50,

"totalPrice": 550

}

* **PUT /api/pricing/{flightId}**
  + **Description:** Update pricing details for a flight.

**Request Payload:**json  
Copy code  
{

"basePrice": 520,

"tax": 55

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Updated"

}

* **POST /api/pricing**
  + **Description:** Set pricing details for a flight.

**Request Payload:**json  
Copy code  
{

"flightId": "FL123",

"basePrice": 500,

"currency": "USD",

"tax": 50

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Set"

}

* **DELETE /api/pricing/{flightId}**
  + **Description:** Delete pricing details for a flight.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Deleted"

}

* **GET /api/pricing/flight/{flightId}/discounts**
  + **Description:** Retrieve all discounts for a flight.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"discounts": [

{

"discountId": "DS123",

"amount": 50,

"currency": "USD",

"type": "Seasonal"

}

]

}

* **POST /api/pricing/{flightId}/discounts**
  + **Description:** Apply a discount to a flight.

**Request Payload:**json  
Copy code  
{

"amount": 50,

"currency": "USD",

"type": "Seasonal"

}

**Response Payload:**json  
Copy code  
{

"discountId": "DS123",

"status": "Applied"

}

* **DELETE /api/pricing/discounts/{discountId}**
  + **Description:** Remove a discount from a flight.

**Response Payload:**json  
Copy code  
{

"discountId": "DS123",

"status": "Removed"

}

### **7. Flight Microservice**

* **POST /api/flights**
  + **Description:** Add a new flight.

**Request Payload:**json  
Copy code  
{

"flightId": "FL123",

"departure": "2024-09-15T08:00:00Z",

"arrival": "2024-09-15T10:00:00Z",

"origin": "NYC",

"destination": "LAX"

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Added"

}

* **GET /api/flights/{id}**
  + **Description:** Retrieve flight details by ID.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"departure": "2024-09-15T08:00:00Z",

"arrival": "2024-09-15T10:00:00Z",

"origin": "NYC",

"destination": "LAX"

}

* **PUT /api/flights/{id}**
  + **Description:** Update flight details by ID.

**Request Payload:**json  
Copy code  
{

"departure": "2024-09-15T09:00:00Z"

}

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Updated"

}

* **DELETE /api/flights/{id}**
  + **Description:** Delete a flight by ID.

**Response Payload:**json  
Copy code  
{

"flightId": "FL123",

"status": "Deleted"

}

* **GET /api/flights/origin/{origin}**
  + **Description:** Retrieve all flights from a specific origin.

**Response Payload:**json  
Copy code  
{

"origin": "NYC",

"flights": [

{

"flightId": "FL123",

"departure": "2024-09-15T08:00:00Z",

"arrival": "2024-09-15T10:00:00Z",

"destination": "LAX"

}

]

}

* **GET /api/flights/destination/{destination}**
  + **Description:** Retrieve all flights to a specific destination.

**Response Payload:**json  
Copy code  
{

"destination": "LAX",

"flights": [

{

"flightId": "FL123",

"departure": "2024-09-15T08:00:00Z",

"arrival": "2024-09-15T10:00:00Z",

"origin": "NYC"

}

]

}

* **GET /api/flights/schedule/{date}**
  + **Description:** Retrieve all flights scheduled for a specific date.

**Response Payload:**json  
Copy code  
{

"date": "2024-09-15",

"flights": [

{

"flightId": "FL123",

"departure": "2024-09-15T08:00:00Z",

"arrival": "2024-09-15T10:00:00Z",

"origin": "NYC",

"destination": "LAX"

}

]

}

### **Microservices Architecture**

#### **1. Flight Search Microservice**

* **Purpose:** Handles flight search functionality. It queries available flights from the Inventory and filters results based on user inputs (departure, arrival, date).
* **Challenges Addressed:**
  + **Reactive Programming:** Efficient handling of asynchronous flight search results.
  + **Data Consistency:** Data consistency is maintained with eventual consistency models.
  + **Communication:** Interacts with the Inventory Management microservice using gRPC.
* **Endpoints with RBAC:**
  + GET /api/flights/search
    - **Role:** Guest, User, Admin
    - **Description:** Search for available flights based on criteria.
  + GET /api/flights/{id}
    - **Role:** Guest, User, Admin
    - **Description:** Retrieve flight details by ID.
  + POST /api/flights
    - **Role:** Admin
    - **Description:** Create a new flight entry.
  + PUT /api/flights/{id}
    - **Role:** Admin
    - **Description:** Update flight details by ID.
  + DELETE /api/flights/{id}
    - **Role:** Admin
    - **Description:** Delete a flight entry by ID.
  + GET /api/flights/availability/{id}
    - **Role:** User, Admin
    - **Description:** Retrieve flight seat availability.
  + GET /api/flights/price/{id}
    - **Role:** User, Admin
    - **Description:** Retrieve the price of a specific flight.

#### **2. Booking Microservice**

* **Purpose:** Manages flight bookings, confirms availability, and issues tickets.
* **Challenges Addressed:**
  + **Saga Pattern:** For maintaining consistency during the multi-step booking process (reserving a seat, charging the customer).
  + **CQRS Pattern:** Used to separate the read and write operations in the booking process.
  + **Circuit Breakers:** Implemented to handle failure gracefully during booking confirmation.
* **Endpoints with RBAC:**
  + POST /api/bookings
    - **Role:** User
    - **Description:** Create a new booking.
  + GET /api/bookings/{id}
    - **Role:** User (own bookings), Admin
    - **Description:** Retrieve booking details by ID.
  + PUT /api/bookings/{id}
    - **Role:** User (own bookings), Admin
    - **Description:** Update booking details by ID.
  + DELETE /api/bookings/{id}
    - **Role:** User (own bookings), Admin
    - **Description:** Cancel a booking by ID.
  + GET /api/bookings/user/{userId}
    - **Role:** User (own bookings), Admin
    - **Description:** Retrieve all bookings for a user.
  + POST /api/bookings/{id}/add-passenger
    - **Role:** User (own bookings), Admin
    - **Description:** Add a new passenger to an existing booking.
  + POST /api/bookings/{id}/update-payment
    - **Role:** User (own bookings), Admin
    - **Description:** Update payment details for a booking.

#### **3. Payment Microservice**

* **Purpose:** Handles payment processing for bookings.
* **Challenges Addressed:**
  + **Circuit Breakers:** Used to handle failures in payment gateways, ensuring fallback mechanisms.
  + **Retries:** Implemented to handle transient payment failures.
  + **API Gateway Integration:** Security is enforced using the API Gateway, ensuring that only authenticated users can access payment services.
* **Endpoints with RBAC:**
  + POST /api/payments
    - **Role:** User
    - **Description:** Process a new payment.
  + GET /api/payments/{id}
    - **Role:** User (own payments), Admin
    - **Description:** Retrieve payment details by ID.
  + PUT /api/payments/{id}
    - **Role:** User (own payments), Admin
    - **Description:** Update payment details by ID.
  + DELETE /api/payments/{id}
    - **Role:** Admin
    - **Description:** Delete a payment record by ID.
  + GET /api/payments/booking/{bookingId}
    - **Role:** User (own payments), Admin
    - **Description:** Retrieve all payments for a booking.
  + GET /api/payments/user/{userId}
    - **Role:** User (own payments), Admin
    - **Description:** Retrieve all payments for a user.
  + POST /api/payments/{id}/refund
    - **Role:** User (own payments), Admin
    - **Description:** Process a refund for a payment.

#### **4. User Management Microservice**

* **Purpose:** Manages user profiles, registration, login, and authentication.
* **Challenges Addressed:**
  + **JWT Authentication:** Uses JSON Web Tokens (JWT) for secure API access.
  + **Reactive Programming:** Provides a non-blocking user experience, ensuring fast response times.
  + **Security via API Gateway:** User data is protected via the API Gateway’s security features, including rate limiting and authentication.
* **Endpoints with RBAC:**
  + POST /api/users
    - **Role:** Guest (for registration)
    - **Description:** Register a new user.
  + POST /api/users/login
    - **Role:** Guest
    - **Description:** Authenticate a user.
  + GET /api/users/{id}
    - **Role:** User (own profile), Admin
    - **Description:** Retrieve user details by ID.
  + PUT /api/users/{id}
    - **Role:** User (own profile), Admin
    - **Description:** Update user details by ID.
  + DELETE /api/users/{id}
    - **Role:** Admin
    - **Description:** Delete a user by ID.
  + POST /api/users/{id}/change-password
    - **Role:** User (own profile)
    - **Description:** Change user password.
  + GET /api/users/{id}/bookings
    - **Role:** User (own profile), Admin
    - **Description:** Retrieve all bookings for a user.

#### **5. Notification Microservice**

* **Purpose:** Sends email and SMS notifications to users on booking and payment status.
* **Challenges Addressed:**
  + **Event-Driven Architecture:** Receives events (booked flights, payment success) from other services and triggers notifications asynchronously.
  + **Message Queues:** Uses message queues like RabbitMQ for handling notification messages efficiently.
* **Endpoints with RBAC:**
  + POST /api/notifications
    - **Role:** Admin
    - **Description:** Send a notification to a user.
  + GET /api/notifications/{id}
    - **Role:** User (own notifications), Admin
    - **Description:** Retrieve notification details by ID.
  + PUT /api/notifications/{id}
    - **Role:** Admin
    - **Description:** Update notification details by ID.
  + DELETE /api/notifications/{id}
    - **Role:** Admin
    - **Description:** Delete a notification by ID.
  + GET /api/notifications/user/{userId}
    - **Role:** User (own notifications), Admin
    - **Description:** Retrieve all notifications for a user.
  + POST /api/notifications/{id}/resend
    - **Role:** Admin
    - **Description:** Resend a notification.
  + GET /api/notifications/status/{status}
    - **Role:** Admin
    - **Description:** Retrieve notifications by status.

#### **6. Pricing Microservice**

* **Purpose:** Calculates the cost of tickets dynamically based on various factors such as demand, availability, and promotions.
* **Challenges Addressed:**
  + **Reactive Streams:** Real-time price calculation based on dynamic variables.
  + **Consistency Models:** Ensures strong consistency for pricing calculations.
* **Endpoints with RBAC:**
  + GET /api/pricing/{flightId}
    - **Role:** User, Admin
    - **Description:** Retrieve pricing details for a flight.
  + PUT /api/pricing/{flightId}
    - **Role:** Admin
    - **Description:** Update pricing details for a flight.
  + POST /api/pricing
    - **Role:** Admin
    - **Description:** Set pricing details for a flight.
  + DELETE /api/pricing/{flightId}
    - **Role:** Admin
    - **Description:** Delete pricing details for a flight.
  + GET /api/pricing/flight/{flightId}/discounts
    - **Role:** Admin
    - **Description:** Retrieve all discounts for a flight.
  + POST /api/pricing/{flightId}/discounts
    - **Role:** Admin
    - **Description:** Apply a discount to a flight.
  + DELETE /api/pricing/discounts/{discountId}
    - **Role:** Admin
    - **Description:** Remove a discount from a flight.

### **7. Flight Microservice (with RBAC)**

**Purpose:** This microservice manages all operations related to flights, including flight creation, retrieval, updating, deletion, and flight schedule queries. It provides the interface to manage the available flights and their schedules.

**Challenges Addressed:**

* **Data Consistency:** Ensure that flights are accurately represented across all requests.
* **Reactive Programming:** Handle a large number of requests efficiently without blocking.
* **API Gateway:** Secures all endpoints and ensures proper user roles through JWT authentication.

#### **Endpoints with RBAC**

1. **POST /api/flights**
   * **Description:** Add a new flight.
   * **Roles:** Admin
   * **RBAC:** Only administrators are allowed to add new flights.
2. **GET /api/flights/{id}**
   * **Description:** Retrieve flight details by ID.
   * **Roles:** Guest, User, Admin
   * **RBAC:** All roles can access flight details, including guests.
3. **PUT /api/flights/{id}**
   * **Description:** Update flight details by ID.
   * **Roles:** Admin
   * **RBAC:** Only administrators can update flight information.
4. **DELETE /api/flights/{id}**
   * **Description:** Delete a flight by ID.
   * **Roles:** Admin
   * **RBAC:** Only administrators are authorized to delete a flight.
5. **GET /api/flights/origin/{origin}**
   * **Description:** Retrieve all flights from a specific origin.
   * **Roles:** Guest, User, Admin
   * **RBAC:** All roles can search for flights based on origin, including guests.
6. **GET /api/flights/destination/{destination}**
   * **Description:** Retrieve all flights to a specific destination.
   * **Roles:** Guest, User, Admin
   * **RBAC:** Accessible to all roles, including guests, users, and admins.
7. **GET /api/flights/schedule/{date}**
   * **Description:** Retrieve all flights scheduled for a specific date.
   * **Roles:** Guest, User, Admin
   * **RBAC:** Available to all roles to check flight schedules based on dates.

a focus on **Eureka Server**, **Config Server**, **Reactiveness**, **ELK Stack Integration**, **API Gateway**, and **Monitoring Tools** like **Grafana** and **Prometheus**:

### **1. Eureka Server**

**Purpose**: **Eureka Server** is used for **service discovery** in a microservices architecture. It allows microservices to register themselves and discover other services dynamically, eliminating the need for hardcoded service locations.

* **Deployment**: Eureka Server is deployed as a standalone service.
* **Configuration**: Each microservice registers with Eureka Server at startup, and services can discover each other using the service name.
* **Endpoints**:
  + **/eureka/apps**: Endpoint for querying registered services.
  + **/eureka/apps/{serviceId}**: Endpoint for querying a specific service instance.

**Example Configuration**:

yaml

Copy code

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

instance:

hostname: ${hostname:localhost}

### **2. Config Server**

**Purpose**: **Config Server** provides centralized configuration management. It allows microservices to retrieve their configuration from a central location.

* **Deployment**: Config Server is deployed as a standalone service, often using a Git repository or a file system for storing configuration files.
* **Configuration**: Microservices pull their configuration from the Config Server at startup and whenever there are changes.
* **Endpoints**:
  + **/actuator/env**: Endpoint to view the environment properties.
  + **/actuator/refresh**: Endpoint to refresh configurations without restarting the application.

**Example Configuration**:

yaml

Copy code

spring:

cloud:

config:

server:

git:

uri: https://github.com/your-repo/config-repo

search-paths: config

### **3. Reactiveness**

**Purpose**: **Reactiveness** refers to the use of reactive programming principles to handle asynchronous data flows and improve system scalability and responsiveness.

* **Frameworks**: Use **Spring WebFlux** or **Project Reactor** to implement reactive programming in microservices.
* **Benefits**: Handles a high volume of concurrent requests with non-blocking I/O operations.
* **Endpoints**:
  + **Reactive Controllers**: Implemented using @RestController with reactive types like Mono and Flux.

**Example Code**:

java

Copy code

@RestController

public class FlightController {

@GetMapping("/api/flights/{id}")

public Mono<Flight> getFlight(@PathVariable String id) {

return flightService.findById(id);

}

}

### **4. ELK Stack Integration**

**Purpose**: The **ELK Stack** (Elasticsearch, Logstash, Kibana) is used for centralized logging and analysis.

* **Elasticsearch**: Stores and indexes log data.
* **Logstash**: Collects, processes, and forwards logs from various sources.
* **Kibana**: Provides a user interface to visualize and search log data.

**Deployment**:

* **Logstash** agents collect logs from microservices and push them to **Elasticsearch**.
* **Kibana** provides dashboards for querying and visualizing log data.

**Example Logstash Configuration**:

yaml

Copy code

input {

file {

path => "/var/log/microservice/\*.log"

start\_position => "beginning"

}

}

output {

elasticsearch {

hosts => ["http://localhost:9200"]

index => "microservice-logs-%{+YYYY.MM.dd}"

}

}

**Kibana Dashboards**:

* **Visualizations**: Create visualizations like histograms, pie charts, and line graphs based on log data.
* **Search**: Query logs for specific events or patterns.

### **5. API Gateway**

**Purpose**: The **API Gateway** acts as a single entry point for all client requests, routing them to the appropriate microservice.

* **Responsibilities**: Request routing, authentication, rate limiting, and response aggregation.
* **Tools**: Commonly used tools include **Spring Cloud Gateway**, **Kong**, and **Zuul**.

**Example Configuration (Spring Cloud Gateway)**:

yaml

Copy code

spring:

cloud:

gateway:

routes:

- id: flight-service

uri: lb://FLIGHT-SERVICE

predicates:

- Path=/api/flights/\*\*

filters:

- AddRequestHeader=Authorization, Bearer ${jwt.token}

**Features**:

* **Security**: Handles JWT authentication and authorization.
* **Rate Limiting**: Controls the rate of requests to prevent abuse.

### **6. Monitoring Tools**

### **Prometheus**

**Overview**:

* **Prometheus** is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is particularly well-suited for microservices and containerized environments.

**Key Features**:

1. **Metrics Collection**:
   * Prometheus collects metrics by scraping HTTP endpoints exposed by applications (typically /actuator/prometheus in Spring Boot applications).
   * Supports various types of metrics such as counters, gauges, histograms, and summaries.
2. **Data Model**:
   * Uses a time-series data model where each metric is stored with a set of labels (key-value pairs) to provide context.
3. **Query Language**:
   * Prometheus uses **PromQL (Prometheus Query Language)** to query metrics and generate insights. PromQL allows for powerful querying and aggregation of time-series data.
4. **Alerting**:
   * Prometheus integrates with **Alertmanager** to handle alerts based on PromQL queries. Alerts can be configured to send notifications via email, Slack, or other channels.
5. **Storage**:
   * Data is stored on disk in a time-series database optimized for fast writes and queries.

**Example Configuration**:

**Prometheus Configuration File (prometheus.yml)**:

yaml

Copy code

global:

scrape\_interval: 15s # How frequently to scrape metrics from targets

evaluation\_interval: 15s # How frequently to evaluate rules

scrape\_configs:

- job\_name: 'microservices'

scrape\_interval: 15s

static\_configs:

- targets: ['localhost:8080', 'localhost:8081'] # Add your service endpoints here

alerting:

alertmanagers:

- static\_configs:

- targets:

- 'localhost:9093' # Alertmanager endpoint

**PromQL Examples**:

**Total Requests**:  
promql  
Copy code  
sum(rate(http\_requests\_total[5m]))

**Error Rate**:  
promql  
Copy code  
rate(http\_errors\_total[5m]) / rate(http\_requests\_total[5m])

### **Grafana**

**Overview**:

* **Grafana** is an open-source platform for monitoring and observability that allows you to visualize time-series data from multiple sources, including Prometheus.

**Key Features**:

1. **Dashboards**:
   * Grafana provides a rich interface to create interactive and customizable dashboards. Dashboards can include various types of visualizations such as graphs, tables, and gauges.
2. **Data Sources**:
   * Grafana supports multiple data sources, including Prometheus, InfluxDB, Elasticsearch, and others. You can add and configure these data sources through the Grafana UI.
3. **Alerts**:
   * You can set up alert rules within Grafana panels. Alerts can be configured to trigger based on specific conditions and send notifications to various channels.
4. **Plugins**:
   * Grafana supports a wide range of plugins for enhanced visualizations, data sources, and other functionalities.

**Example Configuration**:

**Adding Prometheus as a Data Source**:

1. Navigate to **Configuration** > **Data Sources** in Grafana.
2. Click **Add data source** and select **Prometheus**.
3. Configure the Prometheus URL (e.g., http://localhost:9090).
4. Click **Save & Test** to verify the connection.

**Creating a Dashboard**:

1. Click **Create** > **Dashboard**.
2. Add a new panel and select **Prometheus** as the data source.
3. Use PromQL queries to define the metrics you want to visualize.
4. Configure the visualization type (e.g., graph, gauge) and panel settings.
5. Save the dashboard.

**Example Panel Configuration**:

* **Panel Type**: Graph

**Query**:  
promql  
Copy code  
sum(rate(http\_requests\_total[5m])) by (job)

* **Visualization Settings**: Configure colors, legends, and axis labels.

**Alerting**:

1. In a panel, click **Alert** to create an alert rule.
2. Define conditions (e.g., if a metric exceeds a threshold).
3. Configure notification channels (e.g., Slack, email).

**Example Alert Rule**:

**Condition**:  
promql  
Copy code  
sum(rate(http\_errors\_total[5m])) > 5

* **Notification**: Send alert to a configured Slack channel if the condition is met.