Sistema de Gestión de Reservas de Vuelos

Este proyecto es una solución para la **Prueba Técnica – Sistema de Gestión de Reservas de Vuelos**. Se trata de una API REST desarrollada en Java 17 utilizando Spring Boot, que permite gestionar reservas de vuelos de forma concurrente y escalable, garantizando la integridad en la asignación de asientos. Además, se han incorporado mejoras extras para optimizar el rendimiento en consultas, implementar autenticación basada en JWT y facilitar el despliegue mediante Docker.

Tabla de Contenidos

Características

- **Consultar disponibilidad de asientos:** Permite verificar la disponibilidad de asientos en un vuelo específico.
- **Reservar un asiento:** Gestiona la reserva de asientos de forma concurrente, evitando la sobreventa.
- **Cancelar una reserva:** Permite cancelar una reserva previamente confirmada.
- **Listar reservas confirmadas:** Obtiene la lista de reservas confirmadas para un vuelo.
- **Optimización de consultas:** Se utilizan proyecciones y cacheo para mejorar el rendimiento en las consultas de disponibilidad.
- **Autenticación JWT:** Se implementa un sistema de autenticación basado en JSON Web Tokens (JWT) para proteger los endpoints.
- **Mensajería con Kafka:** Se publica un evento asíncrono en Kafka para cada reserva confirmada (en el entorno de producción).
- **Dockerización:** Contiene Dockerfile y docker-compose.yml para facilitar el despliegue y la integración con PostgreSQL, Kafka y Zookeeper.

- **Pruebas Automatizadas:** Incluye tests unitarios e de integración para garantizar la calidad del código.

Requisitos del Ejercicio

- **Operaciones de la API REST:**
  - Consultar disponibilidad de asientos en un vuelo.
  - Reservar un asiento en un vuelo específico.
  - Cancelar una reserva.
  - Obtener la lista de reservas confirmadas para un vuelo.
- **Requisitos técnicos:**
  - Java 17+
  - Spring Boot (con módulos Spring Web, Spring Data JPA, Spring Security)
  - Base de datos: PostgreSQL (en producción) o H2 (para tests y desarrollo)
  - JPA + Hibernate para persistencia
  - Tests automatizados con JUnit 5 y Mockito
  - Documentación de la API mediante SpringDoc (Swagger)
  - Mensajería asíncrona con Kafka (utilizada para la publicación de eventos de reserva)
- **Extras:**
  - Mejor rendimiento en consultas de disponibilidad (usando proyecciones y cacheo).
  - Autenticación robusta mediante JWT.
  - Dockerización y despliegue (Docker Compose, etc.).

Tecnologías Utilizadas

- **Lenguaje:** Java 17
- **Frameworks y Librerías:**
  - Spring Boot 3.x
  - Spring Web
  - Spring Data JPA
  - Spring Security (con JWT)
  - SpringDoc OpenAPI (Swagger)
  - Apache Kafka (para mensajería)
  - H2 Database (para tests y desarrollo)
  - Maven (gestión de dependencias y construcción)
  - JUnit 5, Mockito y AssertJ (para pruebas automatizadas)
- **Docker:** Dockerfile y Docker Compose para la orquestación de servicios (PostgreSQL, Kafka, Zookeeper).

Estructura del Proyecto

La estructura del proyecto es la siguiente:

```
flightreservation
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── example
│   │   │           └── flightreservation
│   │   │               ├── FlightReservationApplication.java
│   │   │               ├── config
│   │   │               │   ├── KafkaProducerConfig.java
│   │   │               │   ├── SecurityConfig.java
│   │   │               │   └── SwaggerConfig.java
│   │   │               ├── controller
│   │   │               │   ├── FlightController.java
│   │   │               │   └── ReservationController.java
│   │   │               ├── dto
│   │   │               │   ├── AuthRequest.java
│   │   │               │   ├── AuthResponse.java
│   │   │               │   ├── FlightCreationRequest.java
│   │   │               │   ├── FlightResponse.java
│   │   │               │   ├── ReservationRequest.java
│   │   │               │   └── ReservationResponse.java
│   │   │               ├── exception
│   │   │               │   ├── GlobalExceptionHandler.java
│   │   │               │   ├── ReservationNotFoundException.java
│   │   │               │   └── SeatNotAvailableException.java
│   │   │               ├── model
│   │   │               │   ├── Flight.java
│   │   │               │   ├── Reservation.java
│   │   │               │   └── ReservationStatus.java
│   │   │               ├── projection
│   │   │               │   └── FlightAvailabilityProjection.java
│   │   │               └── service
│   │   │                   ├── FlightService.java
│   │   │                   ├── KafkaProducerService.java
│   │   │                   └── ReservationService.java
│   │   └── resources
│   │       ├── application.properties
│   │       └── META-INF
│   │           └── spring-configuration-metadata.json  (opcional)
│   └── test
│       ├── java
│       │   └── com
│       │       └── example
│       │           └── flightreservation
│       │               ├── FlightreservationApplicationTests.java
│       │               ├── controller
│       │               │   ├── FlightControllerIntegrationTest.java
│       │               │   └── ReservationControllerIntegrationTest.java
│       │               ├── service
│       │               │   ├── FlightServiceTest.java
```

```
│       │                   │   ├── ReservationServiceTest.java
│       │                   │   └── ConcurrentReservationTest.java
│       └── resources
│           └── application.properties   (configuración para tests: H2,
valores dummy, etc.)
├── Dockerfile
├── docker-compose.yml
└── README.md
```

Instalación y Configuración

Prerrequisitos

- **Java 17** instalado.
- **Maven** instalado.
- **Docker** y **Docker Compose** (para despliegue en contenedores).
- Un IDE de Java (IntelliJ IDEA, Eclipse, VS Code con extensiones de Java).

Configuración Local

1. **Clonar el repositorio:**

   ```
   git clone <URL_DEL_REPOSITORIO>
   cd flightreservation
   ```

2. **Construir el proyecto con Maven:**

   ```
   mvn clean install
   ```

3. **Configurar propiedades:**
   - La configuración de producción se encuentra en `src/main/resources/application.properties`.
   - Para pruebas, se utiliza `src/test/resources/application.properties`, que está configurado para usar H2 en memoria y valores dummy para Kafka.

Ejecución de la Aplicación

Ejecutar desde Maven

Para arrancar la aplicación en modo desarrollo:

```
mvn spring-boot:run
```

La aplicación se expondrá por defecto en el puerto **8081** (configurado en el archivo de propiedades).

Ejecutar desde el JAR

Después de compilar el proyecto, ejecuta el JAR:

```
java -jar target/app.jar
```

Dockerización y Despliegue

El proyecto incluye un **Dockerfile** y un archivo **docker-compose.yml** para facilitar el despliegue de la aplicación junto con los servicios necesarios (PostgreSQL, Kafka, Zookeeper).

Dockerfile

El Dockerfile realiza lo siguiente:

- Etapa 1: Usa una imagen de Maven y Java 17 para compilar y empaquetar la aplicación.
- Etapa 2: Usa una imagen de Java (JRE) liviana para ejecutar el JAR generado y expone el puerto 8081.

docker-compose.yml

El archivo docker-compose.yml define los siguientes servicios:

- **app:** La aplicación Spring Boot.
- **postgres:** La base de datos PostgreSQL.
- **zookeeper:** Servicio de Zookeeper para Kafka.
- **kafka:** El broker Kafka (con listeners configurados en puertos 9092 y 29092).

Para arrancar todos los servicios:

```
docker-compose up --build
```

Endpoints Principales

La API cuenta con los siguientes endpoints:

Autenticación (JWT)

- **POST** `/auth/login`
  Body:

  ```
  {
      "username": "admin",
      "password": "admin"
  }
  ```

  Descripción: Devuelve un token JWT si las credenciales son válidas.

Vuelos

- **POST** `/api/flights`
  Body:

  ```
  {
      "flightNumber": "AB123",
      "totalSeats": 5
  }
  ```

  Descripción: Crea un vuelo nuevo.

- **GET** `/api/flights/{flightNumber}/availability`
  Descripción: Retorna la disponibilidad de asientos del vuelo indicado.

Reservas

- **POST** `/api/reservations`
  Body:

  ```
  {

      "flightNumber": "AB123",
      "seatNumber": "1A"
  }
  ```

  Descripción: Crea una reserva para un asiento en el vuelo.

- **GET** `/api/flights/{flightNumber}/reservations`
  Descripción: Retorna la lista de reservas confirmadas para el vuelo.
- **DELETE** `/api/reservations/{id}`
  Descripción: Cancela una reserva existente.

Pruebas Automatizadas

El proyecto incluye pruebas unitarias y de integración.

Pruebas Unitarias

- **FlightServiceTest:** Verifica la creación de vuelos, la reserva exitosa de asientos y el manejo de error cuando no hay asientos disponibles.
- **ReservationServiceTest:** Verifica la creación y cancelación de reservas y el manejo de errores al cancelar una reserva inexistente.
- **ConcurrentReservationTest:** Simula múltiples hilos intentando reservar el último asiento y se asegura que solo una solicitud tenga éxito.

Estas pruebas se encuentran en:

`src/test/java/com/example/flightreservation/service`

Pruebas de Integración

Utilizando **MockMvc** se prueban los endpoints de los controladores:

- **FlightControllerIntegrationTest:** Prueba la creación de un vuelo y la consulta de disponibilidad.
- **ReservationControllerIntegrationTest:** Prueba la creación y cancelación de reservas.

Estas pruebas se encuentran en:

`src/test/java/com/example/flightreservation/controller`

Colección Postman

Se incluye una colección Postman automatizada que:

- Ejecuta el flujo completo, desde autenticación (para obtener el JWT) hasta la creación y cancelación de reservas.
- Actualiza automáticamente las variables de entorno (`jwtToken` y `reservationId`) usando scripts en el apartado de *tests*.

La colección se encuentra en el archivo:
`FlightReservationAPI.postman_collection.json`.

Notas y Consideraciones

- **Seguridad:**
  La aplicación utiliza Spring Security con JWT. En producción se debe configurar de forma adecuada el manejo de usuarios y credenciales.
  Durante los tests de integración, se puede desactivar la seguridad usando `@AutoConfigureMockMvc(addFilters = false)` o inyectar un bean dummy para Kafka.
- **Kafka en Test:**
  Para evitar errores al crear un KafkaProducer real en el entorno de test, se han configurado valores dummy en `src/test/resources/application.properties` y se ha excluido la auto-configuración de Kafka mediante:

  ```
  properties
  spring.autoconfigure.exclude=org.springframework.boot.autoconfigure
  .kafka.KafkaAutoConfiguration
  ```

  Además, se puede usar `@MockBean` o un `@TestConfiguration` para sobrescribir el bean de `KafkaProducerService`.

- **Base de Datos:**
  En producción se utiliza PostgreSQL, pero para los tests se ha configurado H2 en memoria para evitar dependencias externas.
- **Validación:**
  Se recomienda agregar una dependencia para Hibernate Validator (por ejemplo, `hibernate-validator`) si se requiere validación de beans, ya que se ha notificado la falta de un proveedor de validación en los logs.
- **Despliegue:**
  Se ha incluido Dockerfile y docker-compose.yml para facilitar el despliegue de la aplicación junto con PostgreSQL, Kafka y Zookeeper.