Isabella Ocampo Soto - A00382369
Alejandro Córdoba Erazo - A00395678
Valentina Gonzalez Tapiero -A00394152

Nombre y Descripción del Proyecto

TickTopia - Boletos seguros y rápidos para todos.

Este proyecto consiste en el desarrollo de una aplicación web para la gestión de eventos y venta de boletos. Los organizadores podrán crear eventos con detalles específicos, mientras que los usuarios podrán comprar boletos y ver sus adquisiciones. La plataforma garantizará la autenticación de usuarios y la gestión de permisos, permitiendo un control adecuado sobre la creación, modificación y acceso a los eventos con sus presentaciones.

Tecnologías utilizadas:

- NestJS (framework backend)
- TypeORM (ORM)
- PostgreSQL (base de datos)
- JWT (autenticación)
- Swagger (documentación de la API)
- Supertest + Jest (pruebas unitarias y E2E)

Funcionalidades Implementadas con sus endpoints

- **Todas vienen con el prefijo de /api**

Autenticación, Registro, Login de Usuarios

- POST /auth/register: Permite el registro de un nuevo usuario.
- POST /auth/register/event-manager : Permite el registro de un event-manager (admin)
- POST /auth/login: Devuelve un JWT al usuario con las credenciales correctas.
- GET /auth/users: Listado de todos los usuarios (solo admin).
- GET /auth/users/:id : Devuelve los datos de un usuario por ID
- PUT /auth/users/:id : Actualiza un usuario por ID
- PUT /auth/roles/:id Actualiza el rol de un usuario
- DELETE /auth/users/:id : Elimina un usuario po ID

Gestión de Eventos

- POST /event/create: Crear un evento (solo event manager).
- GET /event/findAll: Devuelve todos los eventos creados
- GET /event/find/user/user:id : Devuelve los eventos de un usuario por ID
- GET /event/find/:term : Devuelve los datos de un evento por ID
- PUT /event/update/:id : Actualiza los datos de un evento
- DELETE /event/delete/:id : Elimina un evento por ID (admin y manager)
- DELETE /event/deleteAll: Elimina todos los eventos (solo admin)

Gestión de Presentaciones
- POST /presentation: Crear una presentación ligada a un evento. (admin y manager)
- GET /presentation : Devuelve todas las presentaciones
- GET /presentation/:id : Devuelve una presentación por ID
- PUT /presentation/:id : Actualiza una presentación por ID
- DELETE /presentation/:id : Elimina una presentación por ID

Gestión de Tickets
- POST /tickets/admin: Crear un ticket manualmente (solo admin).
- POST /tickets/buy**: Comprar un ticket como cliente.
- GET /tickets: Ver la lista de todos los tickets (admin).
- GET /tickets/:id : Ver los detalles de un tiquete por ID (admin y cliente)
- PUT /tickets/:id: Actualizar un ticket (admin).
- DELETE /tickets/:id/delete : Eliminar un ticket (admin)

Gestión de Reportes
- GET /report/sales : Devuelve un reporte de ventas
- GET /report/ocupation : Devuelve un reporte de la ocupacion

Pagos (Stripe)
- Simulación de creación de sesiones de pago mediante Stripe con `axios`.

Detalles Técnicos

Autenticación (JWT)
- Se implementó autenticación con JWT. Al hacer login, el sistema genera un token que contiene los datos del usuario (id, email, roles).
- Estrategia ‘JwtStrategy’ implementada para validar y extraer el usuario del token.

Autorización por Roles
- Se creó un decorador `@Auth(...roles)` y un guard personalizado ‘RolesGuard’.
- Dependiendo del rol (`admin`, `client`, `event-manager`), se controla el acceso a rutas específicas.

Persistencia en Base de Datos
- Se utilizó PostgreSQL con TypeORM.
- Las entidades principales son: ‘User’, ‘Event’, ‘Presentation’, ‘Ticket’.
- Las relaciones entre entidades fueron modeladas con decoradores como `@OneToMany`, `@ManyToOne`.

Pruebas

Pruebas Unitarias

- Se implementaron con Jest.
- Se probaron servicios como ‘ticketService’, ‘presentationService’, ‘authService’, ‘eventService’,’reportService’
- Se probaron controladores como ‘ticketController’, ‘presentationService’, ‘authController’, ‘eventController’, ‘failedController’, ‘healthController’,’successController’, ‘reportController’

Pruebas End-to-End (e2e)
- Se usaron `Supertest` y `Jest` para simular peticiones HTTP reales.
- Cada módulo clave tiene pruebas e2e:
  - `ticket-create.e2e-spec.ts`
  - `ticket-update.e2e-spec.ts`
  - `presentation-create.e2e-spec.ts`
  - `auth-login.e2e-spec.ts`
- Se incluyen flujos completos con usuarios registrados y autenticados, verificación de autorizaciones, y validaciones HTTP (200, 401, 403, 404).

Cobertura
- Se alcanzó un 96% de cobertura en la capa de servicios y controladores.

Autenticación paso a paso:

1. **Registro de Usuarios**
   - Ruta: POST /api/auth/register
   - Cuerpo de la solicitud:
     {
       "email": "val31@mail.com",
       "password": "Abc12345",
       "name": "Valentina",
       "lastname": "Gonzalez"
     }
   - El usuario se guarda en la base de datos con el rol por defecto (client si no se asigna otro)
   - La contraseña se encripta con **bcrypt** antes de guardarse.

2. **Inicio de sesión y login**
   - Ruta : POST /api/auth/login
   - El sistema verifica que:
     - El email exista
     - La contraseña sea correcta
   - Si es valido, genera un JWT token:
     {
       "id": "uuid-del-usuario",
       "email": "usuario@mail.com",
       "roles": ["client"]

}

- Este token se retorna en la respuesta y se usa para autenticar futuras peticiones.

3. **Validación del JWT**
   -Todas las rutas están protegidas usan un guard global (AuthGuard) que:
   - Lee el token del header Authorization : Bearer TOKEN
   - Verifica que sea válido con la secret key JWT_SECRET
   - Extrae la información del usuario y la adjunta al request (req.user)

4. **Autorización por roles**
   - Se utiliza un decorador personalizado: @Auth(ValidRoles.admin)
   - Junto con un RoleGuard, que verifica si el req.user.roles incluye alguno de los roles permitidos
   - Así se controla el acceso a rutas como:
     - Solo admin : /tickets/admin
     - Solo client : /tickets/buy
     - Solo event-manager : /event/create

Persistencia de Datos

1. **Tecnología Utilizada**
   - Se utilizó TypeORM como ORM para interactuar con la base de datos relacional
   - La base de datos utilizada es PostgreSQL
   - Todas las entidades están decoradas con @Entity() y representan tablas en la base de datos
2. **Entidades Principales**

- User: Representa a los usuarios del sistema
- Event : Representa a los eventos creados por event managers
- Presentation: Representa presentaciones dentro de un evento
- Ticket : Representa boletos comprados por los usuarios

Cada entidad está relacionada por medio de asociaciones como @ManyToOne, @OneToMany

3. **Ciclo de persistencia**
   a. **Creación** : Se usan metodos de servicio como repository.create() y repository.save() para persistir datos nuevos
   const user = this.userRepository.create(dto);
   await this.userRepository.save(user);

   b. **Lectura**: Se usa repository.find() o repository.findOne() para consultar la base de datos
   const tickets = await this.ticketRepository.find();
   c. **Actualización**: se usa repository.update() o repository.preload() para modificar registros existentes
   await this.presentationRepository.update(id, dto);

d. **Eliminación** : Se usa repository.delete o repository.remove() para borrar registros
   await this.ticketRepository.delete({ id });

4. **Manejando errores y validaciones**
   - Si un registro no existe, se lanza un NotFoundException
   - Se implementan validaciones con class-validator en los DTOs para garantizar la integridad de los datos
   - El uso de try-catch y logs ayuda a capturar errores de base de datos
5. **Seed y limpieza**
   - Existe un endpoint /api/seed para insertar datos de prueba
   - En las pruebas e2e, se crean usuarios y registros con POST, y se limpian con repository.delete() en afterAll()

Ejecución de Pruebas

1. **Tipos de pruebas implementadas**
   - **Unitarias:** Validan el comportamiento individual de los servicios (Service)
   - **E2E(end-to-end):** Validan el flujo completo desde los endpoints de la API, incluyendo autenticación, autorización y persistencia de datos
2. **Herramientas Utilizadas**
   - **Jest**: Framework de pruebas principal
   - **Supertest**: Utilizado para simular peticiones HTTP en pruebas E2E
   - **TypeORM + SQLite/PostgreSQL**: Dependiendo del entorno de pruebas(base de datos real o en memoria)
3. **Comando de ejecución**
   a. Ejecutar pruebas unitarias y e2e: npm run test
   b. Ejecutar solo pruebas e2e: npm run test:e2e
   c. Ver el reporte de cobertura: npm run test:cov
4. **Cobertura de prueba**
   - Se logró una cobertura del código del 93,81% según el reporte generado por Jest
   - Se cubrieron:
     - Casos exitosos(200/201)
     - Casos de error como 401,400,403,404
     - Reglas de negocio, validaciones, relaciones entre entidades y manejo de roles