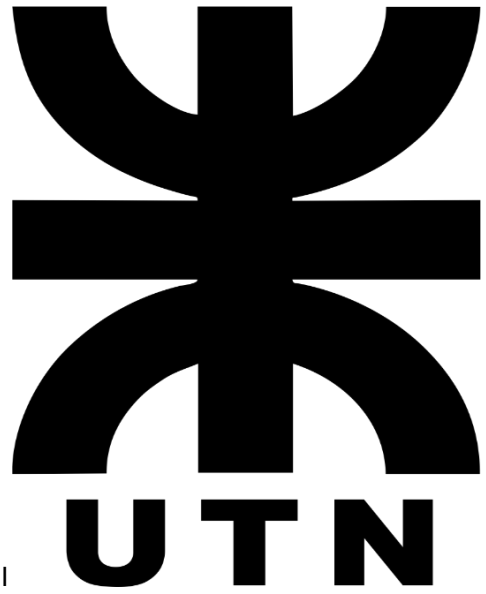


Desarrollo de aplicaciones Cliente - Servidor

Trabajo Práctico Integrador

Grupo N°: 3

Integrantes:

- Acevedo Ariel - acevedoariel.ea1@gmail.com
- Acosta Gaston - gasteac@gmail.com
- Ramirez Joaquin - joaquin.ramirez170998@gmail.com
- Ruiz Franco - ruizfranco2812@gmail.com
- Sosa Diego - dhsosa98@gmail.com
- Vilalta Tomas - tomasfedericovilalta@gmail.com

-2022

Introducción

Este documento es una muestra del sistema desarrollado por el Grupo N°3 haciendo uso de Nest.js y TypeScript. Se describe y explica la arquitectura empleada, tecnologías y los servicios y datos utilizados para desarrollar el módulo correspondiente al equipo. Además, se enseña el funcionamiento de la API y su estructura.

Escenario

En vísperas del mundial de Qatar 2022 la empresa Fantastic Tour (FANTUR S.A.) ha solicitado a los alumnos de la materia Desarrollo de Aplicaciones Cliente-Servidor el desarrollo de un sistema para la venta y administración de paquetes turísticos.

El sistema debe permitir a los clientes registrar en forma electrónica, realizar consultas de paquetes, reservar paquetes y aprobar los mismos por diferentes medios (tarjetas de crédito u otros sistemas de pago on-line) y enviar publicidad (via e-mail) a sus clientes.

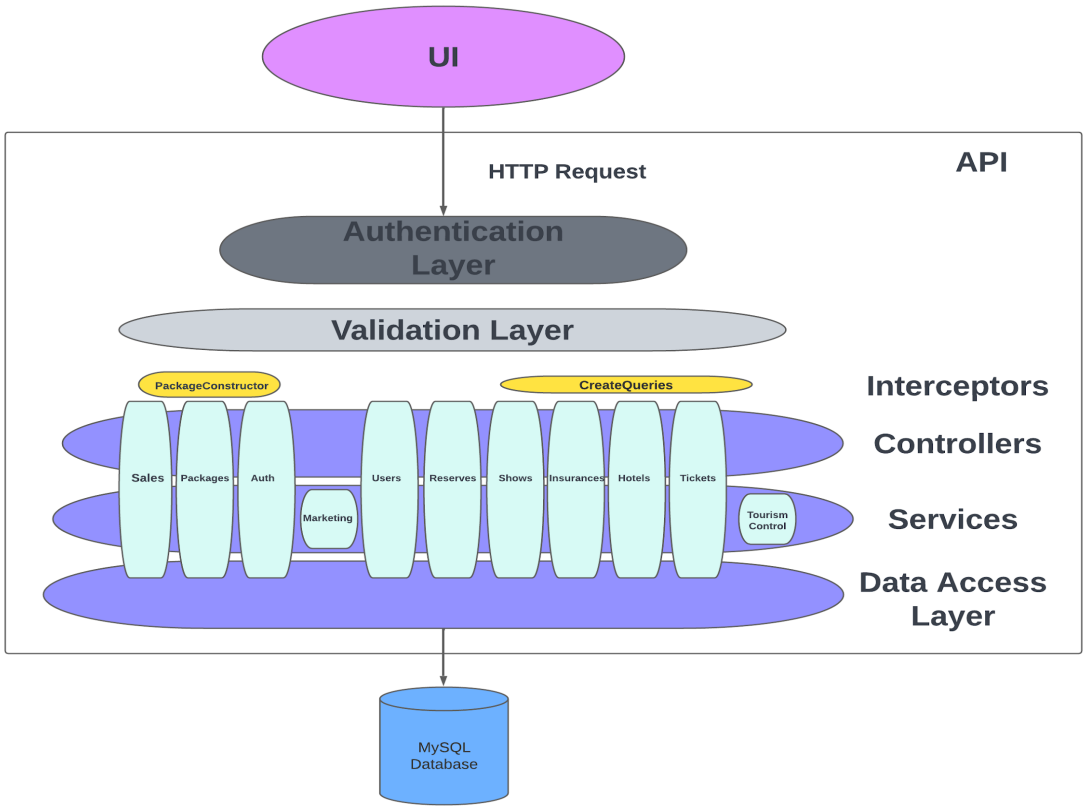
La mayoría de los paquetes turísticos que la empresa comercializa están compuestos por pasajes aéreos o en micro, estadía en hoteles, seguros médicos COVID-19 y entradas a espectáculos. La aplicación debe contemplar el armado y cotización de estos paquetes turísticos por parte de los administradores de la empresa, pudiendo establecerse cuáles paquetes están disponibles o hasta que fecha pueden adquirirse.

Debido a las imposiciones impuestas por los organismos de control que rigen la actividad del sector, las agencias de turismos (incluida FANTUR) deben solicitar permisos al organismo de contralor antes de confirmar las operaciones a sus clientes. Este tipo de solicitudes deber ser realizar en forma on-line y por medio de un web-service que el organismo provee.

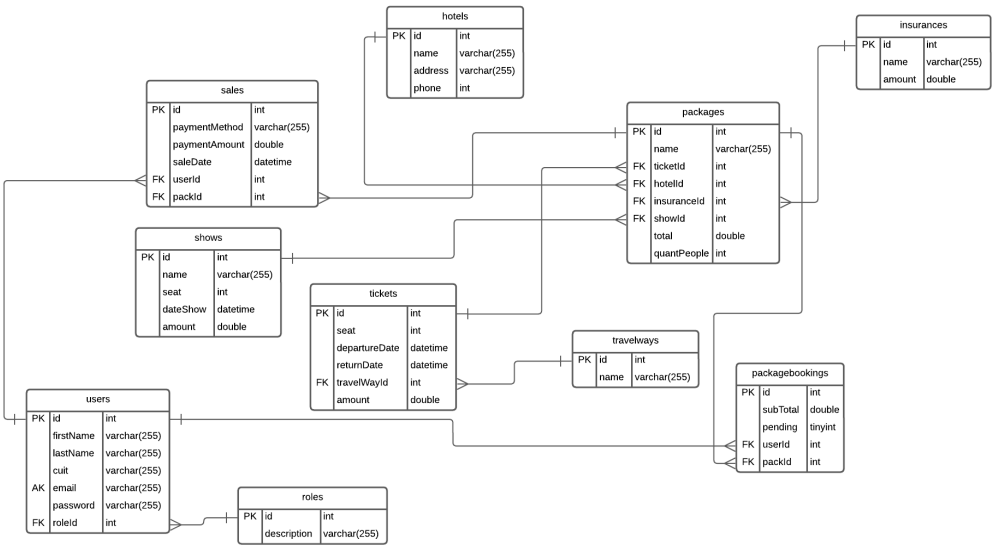
Arquitectura seleccionada

Para la resolución del problema se optó por un servicio web monolítico, desarrollado con Typescript, NodeJS y con el framework de NestJS.

Arquitectura del servicio monolítico:



DER



Motivo de la arquitectura seleccionada

El principal motivo es que al no tener muchos usuarios no es necesario complejizar la arquitectura con microservicios, además de todos los problemas que lleva estructurar una arquitectura basada en microservicios.

Módulos de la arquitectura

- Autenticación
- Usuarios
- Paquetes
- Reservas
- Hoteles
- Seguros
- Pasajes
- Espectáculos
- Ventas
- Turismo Control
- Marketing

Autenticación: Módulo encargado de la autenticación y autorización del acceso a las rutas que lo requieran. Para la autenticación se usó un sistema basado en JWT tokens, y por otro lado se implementó un sistema de roles en donde el usuario tiene que tener el rol correspondiente para poder acceder a dicho endpoint.

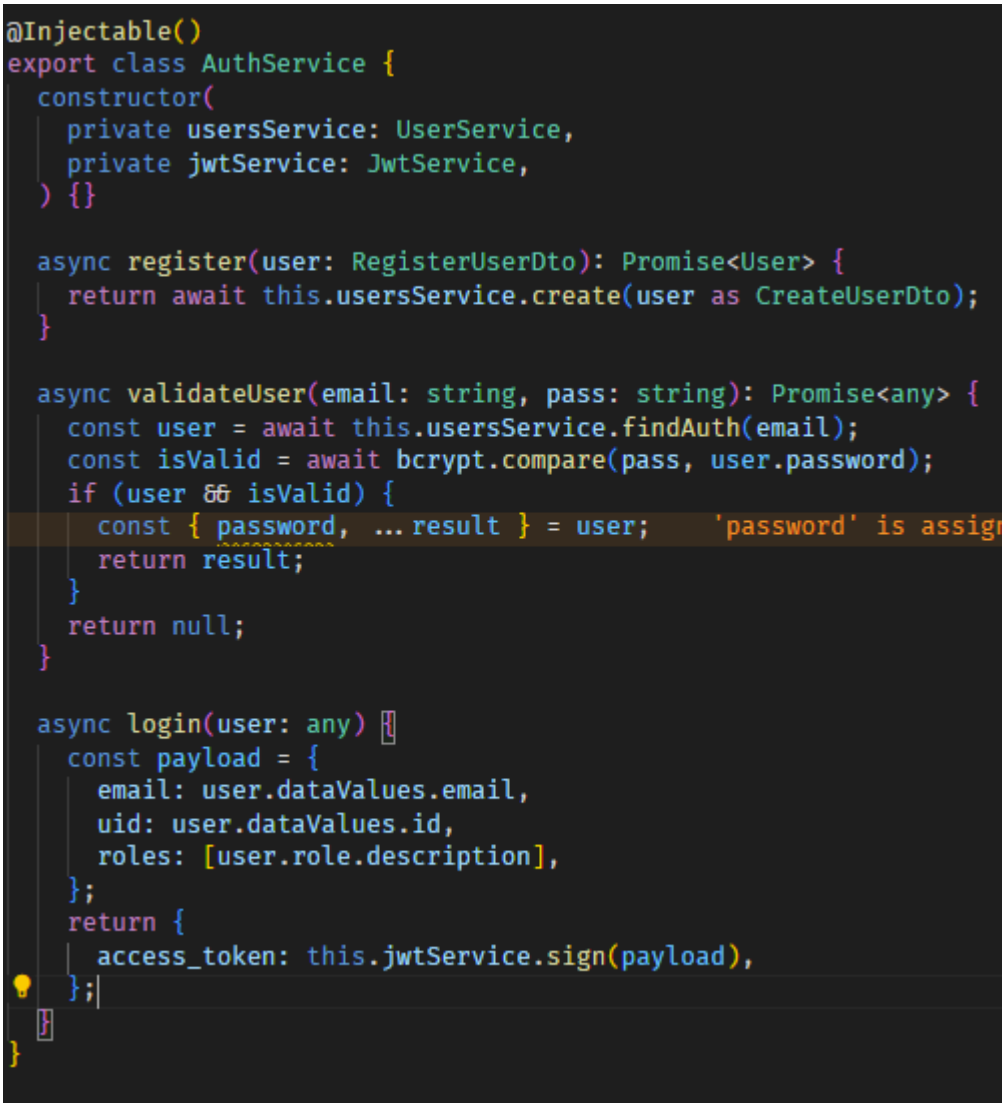
Endpoints expuestos:

POST auth/login

POST auth/signup

Servicios:

Servicio de Autenticación: Servicio utilizado para la autenticación de los usuarios



```
@Injectable()
export class AuthService {
  constructor(
    private usersService: UserService,
    private jwtService: JwtService,
  ) {}

  async register(user: RegisterUserDto): Promise<User> {
    return await this.usersService.create(user as CreateUserDto);
  }

  async validateUser(email: string, pass: string): Promise<any> {
    const user = await this.usersService.findAuth(email);
    const isValid = await bcrypt.compare(pass, user.password);
    if (user && isValid) {
      const { password, ...result } = user;    'password' is assig
      return result;
    }
    return null;
  }

  async login(user: any) {
    const payload = {
      email: user.dataValues.email,
      uid: user.dataValues.id,
      roles: [user.role.description],
    };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

Servicio de JWT: Servicio utilizado para firmar y validar los JWT (Librería externa)

Servicios Externos:

Servicio de Usuarios

Usuarios: Módulo encargado de la gestión y obtención de los usuarios.

Endpoints expuestos:

GET users

GET users/:userId

POST users

DELETE users/:userId

Servicios:

Servicio de Roles**:** Servicio utilizado para la obtención y creación de roles en la base de datos.

```
@Injectable()
export class RoleService {
  constructor(@Inject('ROLE_REPOSITORY') private roleRepository: typeof Role) {}

  async findAll(): Promise<Role[]> {
    const roles = await this.roleRepository.findAll();
    if (!roles.length) {
      await this.create({ description: 'user' });
    }
    return await this.roleRepository.findAll();
  }

  async create(role: any): Promise<Role> {
    const newRole = new Role({ ...role });
    await newRole.save();
    return newRole;
  }
}
```

Servicio de usuarios: Servicio utilizado para la gestión y obtención de usuarios de la base de datos.

```typescript
@Injectable()
export class UserService {
  constructor(
    @Inject('USER_REPOSITORY') private userRepository: typeof User,
    private roleService: RoleService,
  ) {}

  async findOne(id: any): Promise<User> {
    const user = await this.userRepository.findOne({
      where: { id },
      include: [Role],
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }
    return user;
  }

  async findAuth(email: string): Promise<User | undefined> {
    return this.userRepository.findOne({ where: { email }, include: [Role] });
  }

  // BUSAR TODOS LOS USUARIOS NO ES NECESARIO PERO QUERIA PROBAR :)
  async findAll(options?: any): Promise<User[]> {
    return await this.userRepository.findAll(options);
  }

  async create(user: CreateUserDto): Promise<any> {
    let { roleId, password } = user;     'password' is never reassigned. Use 'const
    const roles = await this.roleService.findAll();
    const exist = await this.userRepository.findOne({
      where: { email: user.email },
    });
    if (exist) {
      throw new UnauthorizedException('User already exists');
    }
    if (!roleId) {
      roleId = roles[0].id;
    }
    if (roleId) {
      if (!roles.find((role) => role.id === roleId)) {
        throw new UnauthorizedException('Invalid role');
      }
    }
    const newUser = await this.userRepository.create({
      ...user,
      password: await bcrypt.hash(
        password,
        Number(process.env.HASH_SALT) || 10,
      ),
      roleId,
    });
    await newUser.save();
    return newUser;
  }

  async delete(id: number): Promise<User> {
    const user = await this.userRepository.findOne({
      where: { id },
    });
    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }
    await user.destroy();
    return user;
  }
}
```

Paquetes: Módulo encargado de la gestión y obtención de paquetes turísticos, en este módulo se permite a cualquier usuario obtener los paquetes del sistema, pero solo usuarios con rol administrativo van a poder eliminarlos, actualizarlos o crearlos. Los paquetes además pueden tener o no espectáculos, shows, hoteles y pasajes.

Endpoints expuestos:

GET packages

GET packages/:id

POST packages

DELETE packages/:packageId

PATCH packages/:id

Servicios:

Servicio de paquetes reservados: servicio para el tratamiento de paquetes reservados.

```typescript
@Injectable()
export class ReservedPackagesService {
  constructor(
    @Inject('RESERVED_PACKAGES_REPOSITORY')
    private reservedPackagesRepository: typeof ReservedPackages,
    private packagesService: PackagesService,
    private userService: UserService,
  ) {}

  async findAllReserves(userId: number, options?: any): Promise<any> {
    return await this.reservedPackagesRepository.findAll({
      where: { userId },
      include: [{ model: Package, as: 'pack' }],
    });
  }

  async deleteReserve(userId: number, packageId: number) {
    const reserve = await this.reservedPackagesRepository.findOne({
      where: { userId, packId: packageId },
    });
    if (!reserve) {
      throw new UnauthorizedException('Reserve not found');
    }
    return await reserve.destroy();
  }

  async create(reserve: any): Promise<ReservedPackages> {
    return await this.reservedPackagesRepository.create(reserve);
  }

  async delete(reserve: any) {
    return await this.reservedPackagesRepository.destroy(reserve);
  }

  async createReserve(userId: number, reserve: any) {
    const { packageId } = reserve;
    const user = await this.userService.findOne(userId);
    const packageToBuy = await this.packagesService.findOne(packageId);
    if (!user) {
      throw new UnauthorizedException('User not found');
    }
    if (!packageToBuy) {
      throw new UnauthorizedException('The package doesnt exist');
    }
    const exist = await this.reservedPackagesRepository.findOne({
      where: { userId, packId: packageId },
    });
    if (exist) {
      throw new UnauthorizedException('The package is already reserved');
    }
    return this.create({
      packId: packageId,
      userId,
      subTotal: packageToBuy.total,
      pending: true,
    });
  }
}
```

Servicio de paquetes: servicio para el tratamiento de paquetes.

```typescript
@Injectable()
export class PackagesService {
  constructor(
    @Inject('PACKAGE_REPOSITORY') private packagesRepository: typeof Package,
  ) {}

  async findAll(): Promise<Package[]> {
    return await this.packagesRepository.findAll({
      include: [
        Insurance,
        { model: Ticket, include: [TravelWay] },
        Hotel,
        Show,
      ],
    });
  }

  async create(pack: any): Promise<Package> {
    const newPack = await this.packagesRepository.create({ ...pack });
    await newPack.save();
    return newPack;
  }

  async delete(id: number): Promise<Package> {
    const pack = await this.packagesRepository.findOne({ where: { id } });
    if (!pack) {
      throw new NotFoundException('Package does not exist');
    }
    await pack.destroy();
    return pack;
  }

  async findOne(id: number): Promise<Package> {
    const pack = await this.packagesRepository.findOne({
      where: { id },
      include: [
        Insurance,
        { model: Ticket, include: [TravelWay] },
        Hotel,
        Show,
      ],
    });
    if (!pack) {
      throw new NotFoundException('Package does not exist');
    }
    return pack;
  }

  async update(id: number, pack: any): Promise<Package> {
    let { total } = pack;

    const packToUpdate = await this.findOne(id);
    total += packToUpdate.insurance ? packToUpdate.insurance.amount : 0;
    total += packToUpdate.ticket ? packToUpdate.ticket.amount : 0;
    total += packToUpdate.show ? packToUpdate.show.amount : 0;
    const quant = pack.quantPeople || 1;
    total = total * quant;

    packToUpdate.name = pack.name;
    packToUpdate.quantPeople = pack.quantPeople;
    packToUpdate.insuranceId = pack.insuranceId;
    packToUpdate.ticketId = pack.ticketId;
    packToUpdate.hotelId = pack.hotelId;
    packToUpdate.showId = pack.showId;
    packToUpdate.total = total;
    await packToUpdate.save();
    return packToUpdate;
```

Servicios externos:

Servicio de hoteles

Servicio de pasajes

Servicio de espectáculos

Servicio de seguros

Servicio de usuarios

Reservas: Módulo encargado de la gestión y obtención de reservas. Los usuarios registrados van a poder reservar un único paquete de ese tipo pero la cantidad que quieran de otros paquetes.

Endpoints expuestos:

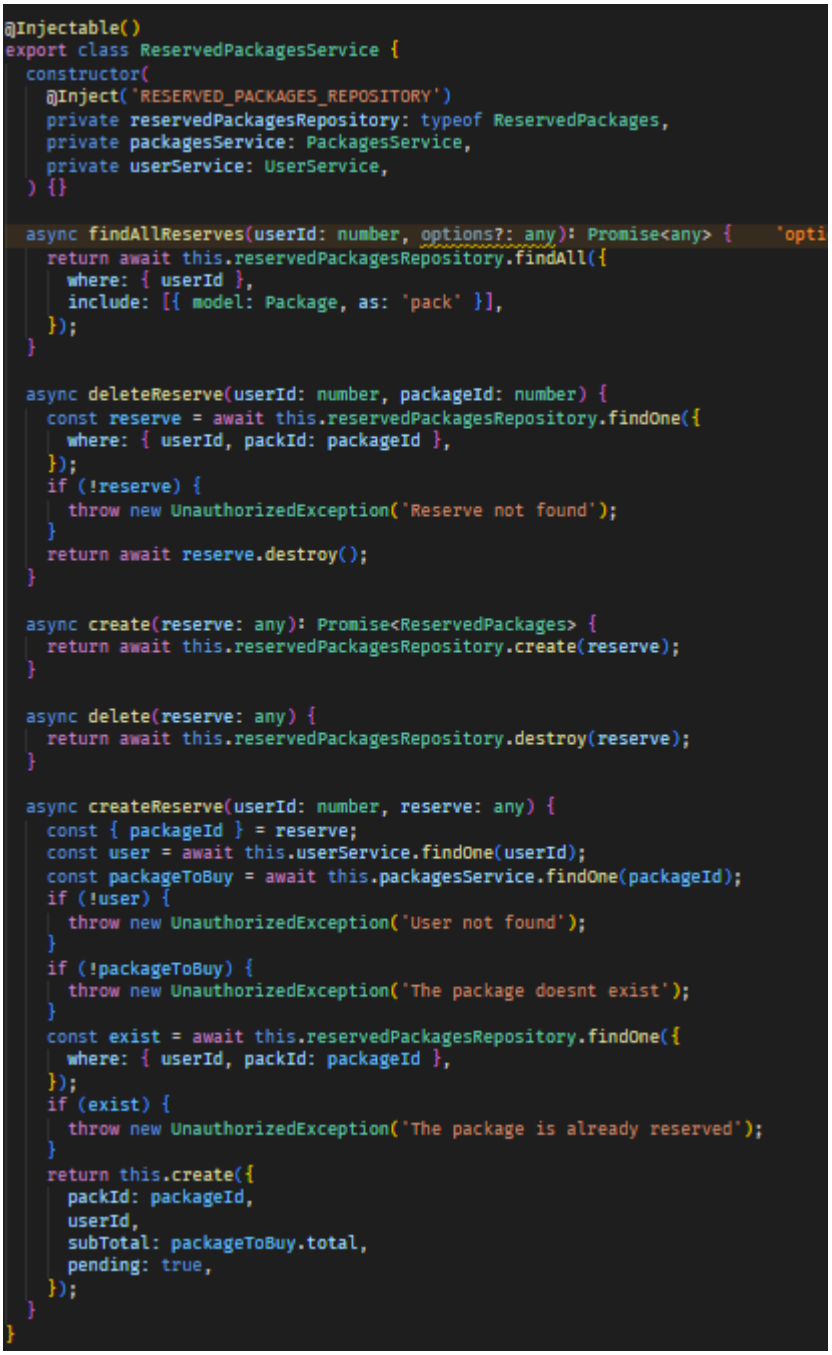
POST reserves

GET reserves

DELETE reserves/:reserveId

Servicios:

Servicio de Reservas: Servicio encargado de la gestión y obtención de reservas



```
@Injectable()
export class ReservedPackagesService {
  constructor(
    @Inject('RESERVED_PACKAGES_REPOSITORY')
    private reservedPackagesRepository: typeof ReservedPackages,
    private packagesService: PackagesService,
    private userService: UserService,
  ) {}

  async findAllReserves(userId: number, options?: any): Promise<any> {    'opti
    return await this.reservedPackagesRepository.findAll({
      where: { userId },
      include: [{ model: Package, as: 'pack' }],
    });
  }

  async deleteReserve(userId: number, packageId: number) {
    const reserve = await this.reservedPackagesRepository.findOne({
      where: { userId, packId: packageId },
    });
    if (!reserve) {
      throw new UnauthorizedException('Reserve not found');
    }
    return await reserve.destroy();
  }

  async create(reserve: any): Promise<ReservedPackages> {
    return await this.reservedPackagesRepository.create(reserve);
  }

  async delete(reserve: any) {
    return await this.reservedPackagesRepository.destroy(reserve);
  }

  async createReserve(userId: number, reserve: any) {
    const { packageId } = reserve;
    const user = await this.userService.findOne(userId);
    const packageToBuy = await this.packagesService.findOne(packageId);
    if (!user) {
      throw new UnauthorizedException('User not found');
    }
    if (!packageToBuy) {
      throw new UnauthorizedException('The package doesnt exist');
    }
    const exist = await this.reservedPackagesRepository.findOne({
      where: { userId, packId: packageId },
    });
    if (exist) {
      throw new UnauthorizedException('The package is already reserved');
    }
    return this.create({
      packId: packageId,
      userId,
      subTotal: packageToBuy.total,
      pending: true,
    });
  }
}
```

Servicios Externos:

Servicio de Usuarios

Servicio de paquetes

Ventas: Módulo encargado de la gestión de ventas de los paquetes a los clientes. Los usuarios registrados van a poder hacer compra de cualquier paquete del sistema a través de una tarjeta de crédito o débito.

Endpoints expuestos:

POST /sales

Servicios:

Servicio de pagos: Servicio utilizado para el procesamiento de pagos

```
@Injectable()
export class PaymentService {
  processPayment(payment: any, amount: any) {
    if (payment.paymentMethod === 'card') {
      return this.cardPayment(payment);
    }
    return false;
  }

  cardPayment(cardPayment) {
    return true;
  }
}
```

Servicio de ventas: Servicio utilizado para buscar ventas, procesar ventas, crear y eliminar ventas.

```typescript
@Injectable()
export class SalesService {
  constructor(
    private paymentService: PaymentService,
    @Inject('SALE_REPOSITORY') private saleRepository: typeof Sale,
    private tourismControlService: ControlTourismService,
    private packagesService: PackagesService,
    private userService: UserService,
    private reservedPackagesService: ReservedPackagesService,
  ) {}

  async findAll(options: any): Promise<Sale[]> {
    return this.saleRepository.findAll({ ...options });
  }

  async findAllBoughts(userId: number, options?: any): Promise<Sale[]> {
    return await this.saleRepository.findAll({
      where: { userId },
      include: [{ model: Package, as: 'sales' }],
    });
  }

  async findAllByUserId(userId: number) {
    return this.saleRepository.findAll({
      where: { userId },
    });
  }

  handlePayment(payment: any, amount: number) {
    const isValid = this.paymentService.processPayment(payment, amount);
    if (isValid) return true;
    return false;
  }

  async create(sale: any, userId: number, amount: number) {
    const { packageId, payment } = sale;
    const { paymentMethod } = payment;
    const newSale = await this.saleRepository.create({
      paymentAmount: amount,
      saleDate: new Date(),
      userId,
      packId: packageId,
      paymentMethod,
    });
    return await newSale.save();
  }

  async createSales(userId: number, packagesByClient: CreateSaleDto) {
    const { packageId, payment } = packagesByClient;
    const user = await this.userService.findOne(userId);
    const packageToBuy = await this.packagesService.findOne(packageId);
    if (!user) {
      throw new UnauthorizedException('User not found');
    }
    if (!packageToBuy) {
      throw new UnauthorizedException('The package doesnt exist');
    }
    const isValid = await this.tourismControlService.validate({
      cuit: user.cuit,
      fecha_incio: new Date().toISOString(),
      fecha_fin: new Date().toISOString(),
      precio: packageToBuy.total,
    });
    if (!isValid) {
      throw new UnauthorizedException(
        'The user has not authorized to buy this package',
      );
    }
    const successPayment = this.handlePayment(payment, packageToBuy.total);
    if (!successPayment) {
      throw new UnauthorizedException('Payment not valid');
    }
    await this.reservedPackagesService.delete({
      where: { userId, packId: packageId },
    });
    return await this.create(packagesByClient, userId, packageToBuy.total);
  }
}
```

Servicios externos:

Servicio de paquetes.

Servicio de control de turismo.

Servicio de usuarios.

Turismo Control: Módulo encargado de la comunicación con el organismo contralor, este módulo le enviará los datos necesarios del cliente que quiera comprar un paquete para saber si el cliente es apto para hacerlo.

Endpoints expuestos:

No tiene

Servicios:

Servicio de Control de Turismo: Servicio encargado de la comunicación con el organismo contralor, este servicio se encargará además de reintentar las requests a ese organismo hasta un máximo de 10 veces en el caso que este no se encuentre disponible.

```typescript
@Injectable()
export class ControlTourismService {
  constructor(private httpService: HttpService) {}

  httpComptroller(req: HttpBodyRequestComptroller): Observable<any> {
    return this.httpService
      .post(
        'http://localhost:8080/operacion',
        { ...req },
        {
          headers: {
            'Content-Type': 'application/json',
            Accept: 'application/json',
          },
        },
      )
      .pipe(
        mergeMap((val: any) => {
          if (val.status >= 400) {
            return throwError(`${val.status}`);
          }
          return of(val.data);
        }),
        retry(10),
        catchError((err) => {
          if (!err.response) {
            return of({ status: 500 });
          }
          return of({ status: err.response.status });
        }),
      );
  }

  async validate(req: HttpBodyRequestComptroller): Promise<boolean> {
    const response = await this.httpComptroller(req).toPromise();
    if (response.aprobada) {
      return true;
    }
    if (response.status >= 400) {
      throw new InternalServerErrorException('Max re-try Exceded');
    }
    return false;
  }
}
```

Servicio Http: Servicio encargado de la lógica de envío de Http requests (Librería externa).

Marketing: Módulo encargado de enviar información a los correos de los clientes registrados del sistema.

Endpoints expuestos:

GET marketing

Servicios:

Servicio de marketing: servicio encargado de enviar e-mails a los usuarios cada mes.

```ts
@Injectable()
export class MarketingService {
  private readonly logger = new Logger(MarketingService.name);
  constructor(
    private userService: UserService,
    private mailerService: MailerService,
  ) {}

  async sendMail(email: string, name: string, packs: any[]) {
    const total = packs.reduce((ant, curr) => {
      return ant + curr.total;
    }, 0);
    if (packs.length) {
      await this.mailerService.sendMail({
        to: email,
        subject: 'Dont forget your Packages',
        template: '/email',
        context: {
          email: email,
          name: name,
          packs: packs,
          sale: {
            id: packs[0].Sale.id,
            saleDate: packs[0].Sale.saleDate,
            total,
          },
        },
      });
    }
  }

  @Cron('* * 19 27 * *')
  // @Timeout(3000)
  async handleNotifications() {
    const date = new Date();
    date.setDate(date.getDate() + 60);
    const salesByUser = await this.userService.findAll({
      include: [
        {
          model: Package,
          as: 'sales',
          include: [
            {
              model: Ticket,
              where: {
                departureDate: {
                  [Op.lt]: date,
                },
              },
              include: TravelWay,
            },
            Show,
            Hotel,
            Insurance,
          ],
        },
      ],
    });
    // return salesByUser;
    salesByUser.forEach(async (user) => {
      const { email, firstName, lastName, sales } = user as any;
      const packs = sales.map(
        ({
          name,
          total,
          quantPeople,
          ticket,
          show,
          hotel,
          insurance,
          id,
          Sale,
        }) => {
          return {
            name,
            total,
            quantPeople,
            ticket,
            show,
            hotel,
            insurance,
            id,
            Sale,
          };
        },
      );
      await this.sendMail(email, `${firstName} ${lastName}`, packs);
    });
    this.logger.debug('Called in intervals of 10 seconds');
  }
}
```

Servicios Externos:

Servicio de correos electrónicos

Servicio de usuarios

Pasajes: Módulo encargado de la gestión y obtención de los pasajes. En este módulo se permite a cualquier usuario obtener los pasajes del sistema, pero solo usuarios con rol administrativo van a poder eliminarlos, actualizarlos o crearlos.

GET tickets

GET tickets/:ticketId

POST tickets

DELETE tickets/:ticketId

PATCH tickets/:ticketId

Servicios:

Servicio de Pasajes**:** Servicio utilizado para la gestión y obtención de pasajes.

```typescript
@Injectable()
export class TicketService {
  constructor(
    @Inject('TICKET_REPOSITORY')
    private ticketRepository: typeof Ticket,
    private travelWaysService: TravelWaysService,
  ) {}

  async findOne(id: number): Promise<Ticket> {
    const ticket = await this.ticketRepository.findOne({ where: { id } });
    if (!ticket) {
      throw new NotFoundException('Ticket does not exist');
    }
    return ticket;
  }

  async findAll(options?: any): Promise<Ticket[]> {
    return await this.ticketRepository.findAll({
      ...options,
      include: TravelWay,
    });
  }

  async delete(id: number): Promise<Ticket> {
    const ticket = await this.findOne(id);
    await this.ticketRepository.destroy({ where: { id } });
    return ticket;
  }

  async create(ticket: TicketDto): Promise<Ticket> {
    const { travelWayId } = ticket;
    const travelWays = await this.travelWaysService.findAll();
    if (!travelWayId) {
      throw new UnauthorizedException('Must be a travel way');
    }
    if (!travelWays.find((travelWay) => travelWay.id === travelWayId)) {
      throw new UnauthorizedException('Invalid travel way');
    }
    const newTicket = await this.ticketRepository.create({ ...ticket });
    await newTicket.save();
    return newTicket;
  }

  async update(ticketId: number, ticket: TicketOnUpdateDto) {
    const { travelWayId } = ticket;
    const ticketToUpdate = await this.findOne(ticketId);
    const travelWays = await this.travelWaysService.findAll();
    if (travelWayId) {
      if (!travelWays.find((travelWay) => travelWay.id === travelWayId)) {
        throw new UnauthorizedException('Invalid travel way');
      }
    }
    ticketToUpdate.seat = ticket.seat;
    ticketToUpdate.departureDate = ticket.departureDate;
    ticketToUpdate.returnDate = ticket.returnDate;
    ticketToUpdate.amount = ticket.amount;
    ticketToUpdate.travelWayId = travelWayId;
    await ticketToUpdate.save();
    return ticketToUpdate;
  }
}
```

Servicio de Formas de Viajes: Servicio utilizado para la obtención y creación de formas de viaje.

```typescript
@Injectable()
export class TicketService {
  constructor(
    @Inject('TICKET_REPOSITORY')
    private ticketRepository: typeof Ticket,
    private travelWaysService: TravelWaysService,
  ) {}

  async findOne(id: number): Promise<Ticket> {
    const ticket = await this.ticketRepository.findOne({ where: { id } });
    if (!ticket) {
      throw new NotFoundException('Ticket does not exist');
    }
    return ticket;
  }

  async findAll(options?: any): Promise<Ticket[]> {
    return await this.ticketRepository.findAll({
      ...options,
      include: TravelWay,
    });
  }

  async delete(id: number): Promise<Ticket> {
    const ticket = await this.findOne(id);
    await this.ticketRepository.destroy({ where: { id } });
    return ticket;
  }

  async create(ticket: TicketDto): Promise<Ticket> {
    const { travelWayId } = ticket;
    const travelWays = await this.travelWaysService.findAll();
    if (!travelWayId) {
      throw new UnauthorizedException('Must be a travel way');
    }
    if (!travelWays.find((travelWay) => travelWay.id == travelWayId)) {
      throw new UnauthorizedException('Invalid travel way');
    }
    const newTicket = await this.ticketRepository.create({ ...ticket });
    await newTicket.save();
    return newTicket;
  }

  async update(ticketId: number, ticket: TicketOnUpdateDto) {
    const { travelWayId } = ticket;
    const ticketToUpdate = await this.findOne(ticketId);
    const travelWays = await this.travelWaysService.findAll();
    if (travelWayId) {
      if (!travelWays.find((travelWay) => travelWay.id == travelWayId)) {
        throw new UnauthorizedException('Invalid travel way');
      }
    }
    ticketToUpdate.seat = ticket.seat;
    ticketToUpdate.departureDate = ticket.departureDate;
    ticketToUpdate.returnDate = ticket.returnDate;
    ticketToUpdate.amount = ticket.amount;
    ticketToUpdate.travelWayId = travelWayId;
    await ticketToUpdate.save();
    return ticketToUpdate;
  }
}
```

Hoteles: Módulo encargado de la gestión y obtención de los hoteles. En este módulo se permite a cualquier usuario obtener los hoteles del sistema, pero solo usuarios con rol administrativo van a poder eliminarlos, actualizarlos o crearlos.

Endpoints expuestos:

GET hotels

GET hotels/:hotelId

POST hotels

PATCH hotels/:hotelId

DELETE hotels/:hotelId

Servicios:

Servicio de Hoteles**:** Servicio utilizado para la gestión y obtención de hoteles.

```
@Injectable()
export class HotelService {
  constructor(
    @Inject('HOTEL_REPOSITORY')
    private hotelRepository: typeof Hotel,
  ) {}

  async findAll(options?: any): Promise<Hotel[]> {
    return this.hotelRepository.findAll(options);
  }

  async findOne(id: number): Promise<Hotel> {
    const hotel = await this.hotelRepository.findOne({
      where: { id },
    });
    if (!hotel) {
      throw new NotFoundException('Hotel not found');
    }
    return hotel;
  }

  async create(hotel: HotelDto): Promise<Hotel> {
    const newHotel = await this.hotelRepository.create({ ...hotel });
    await newHotel.save();
    return newHotel;
  }

  async delete(id: number): Promise<Hotel> {
    const hotel = await this.findOne(id);
    await hotel.destroy();
    return hotel;
  }
  async update(id: number, hotel: any): Promise<Hotel> {
    const HotelOnUpdate = await this.findOne(id);
    HotelOnUpdate.name = hotel.name;
    HotelOnUpdate.address = hotel.address;
    HotelOnUpdate.phone = hotel.phone;
    await HotelOnUpdate.save();
    return HotelOnUpdate;
  }
}
```

Espectáculos: Módulo encargado de la gestión y obtención de los espectáculos. En este módulo se permite a cualquier usuario obtener los espectáculos del sistema, pero solo usuarios con rol administrativo van a poder eliminarlos, actualizarlos o crearlos.

Endpoints expuestos:

GET shows

GET shows/:showId

POST shows

PATCH shows/:showId

DELETE shows/:showId

Servicios:

Servicio de Espectáculos: Servicio utilizado para la gestión y obtención de espectáculos.

```
@Injectable()
export class ShowsService {
  constructor(
    @Inject('SHOW_REPOSITORY')
    private showRepository: typeof Show,
  ) {}

  async findOne(id: number): Promise<Show> {
    const show = await this.showRepository.findByPk(id);
    if (!show) {
      throw new NotFoundException('Show not found');
    }
    return show;
  }

  async findAll(options?: any): Promise<Show[]> {
    return await this.showRepository.findAll(options);
  }

  async create(show: ShowDto): Promise<Show> {
    const newShow = await this.showRepository.create({ ...show });
    await newShow.save();
    return newShow;
  }

  async delete(id: number): Promise<Show> {
    const show = await this.findOne(id);
    await show.destroy();
    return show;
  }

  async update(id: number, show: any): Promise<Show> {
    const showToUpdate = await this.findOne(id);

    showToUpdate.name = show.name;
    showToUpdate.seat = show.seat;
    showToUpdate.dateShow = show.dateShow;
    showToUpdate.amount = show.amount;

    await showToUpdate.update(show);
    return showToUpdate;
  }
}
```

Seguros: Módulo encargado de la gestión y obtención de los seguros. En este módulo se permite a cualquier usuario obtener los seguros del sistema, pero solo usuarios con rol administrativo van a poder eliminarlos, actualizarlos o crearlos.

Endpoints expuestos:

GET insurances

GET insurances/:insuranceId

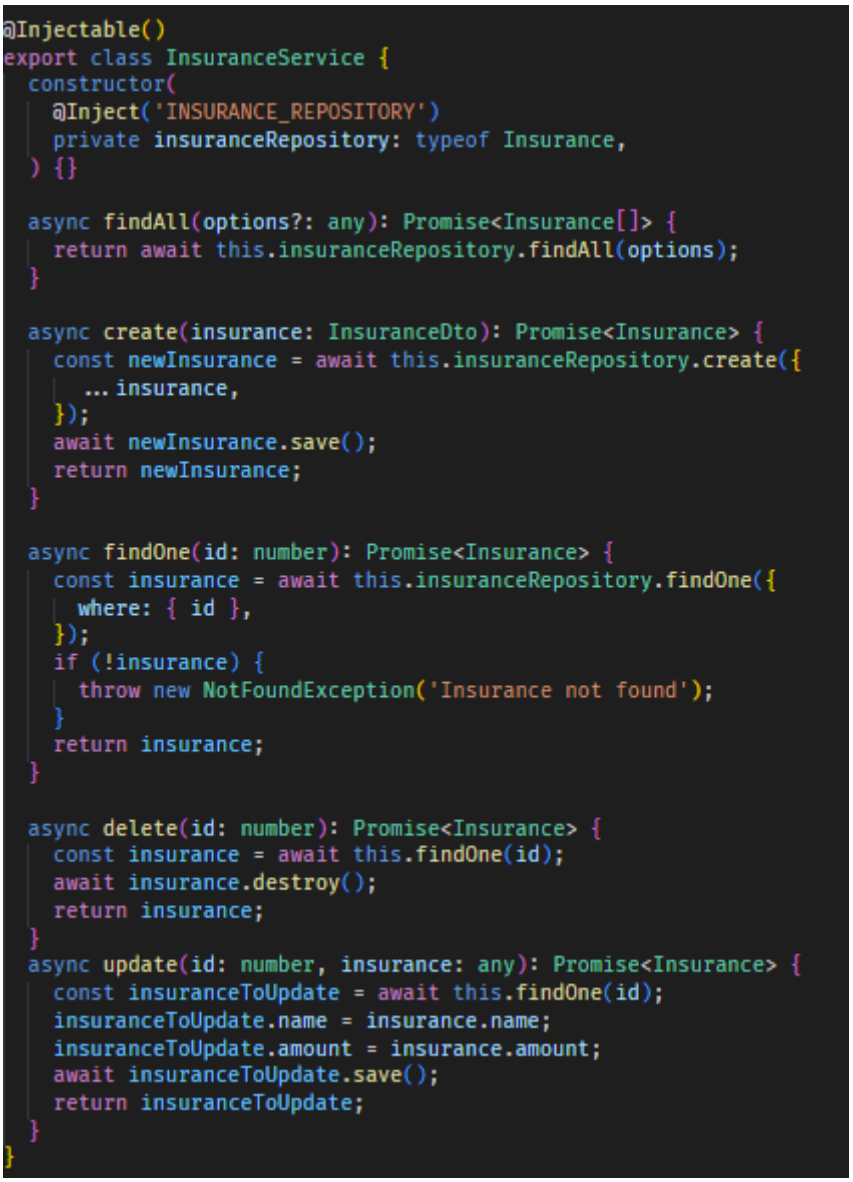
POST insurances

PATCH insurances/:insuranceId

DELETE insurances/:insuranceId

Servicios:

Servicio de Seguros: Servicio utilizado para la gestión y obtención de seguros.



```
@Injectable()
export class InsuranceService {
  constructor(
    @Inject('INSURANCE_REPOSITORY')
    private insuranceRepository: typeof Insurance,
  ) {}

  async findAll(options?: any): Promise<Insurance[]> {
    return await this.insuranceRepository.findAll(options);
  }

  async create(insurance: InsuranceDto): Promise<Insurance> {
    const newInsurance = await this.insuranceRepository.create({
      ...insurance,
    });
    await newInsurance.save();
    return newInsurance;
  }

  async findOne(id: number): Promise<Insurance> {
    const insurance = await this.insuranceRepository.findOne({
      where: { id },
    });
    if (!insurance) {
      throw new NotFoundException('Insurance not found');
    }
    return insurance;
  }

  async delete(id: number): Promise<Insurance> {
    const insurance = await this.findOne(id);
    await insurance.destroy();
    return insurance;
  }
  async update(id: number, insurance: any): Promise<Insurance> {
    const insuranceToUpdate = await this.findOne(id);
    insuranceToUpdate.name = insurance.name;
    insuranceToUpdate.amount = insurance.amount;
    await insuranceToUpdate.save();
    return insuranceToUpdate;
  }
}
```

Tecnologías utilizadas

Nest.js

Nest.js es un framework progresivo de Node.js para la creación de aplicaciones eficientes, confiables y escalables del lado del servidor, el cual está construido y es completamente compatible con TypeScript. Combina elementos de la programación orientada a objetos, programación funcional y programación reactiva funcional.

TypeScript

Typescript es un lenguaje de programación que amplía Javascript con una nueva sintaxis que añade, entre otras cosas, el tipado estático opcional, genéricos, decoradores y elementos de POO como interfaces o property accessors.

MySQL

MySQL es un sistema de gestión de bases de datos relacionales (RDBMS) de código abierto respaldado por Oracle y basado en el lenguaje de consulta estructurado (SQL).

Sequelize

Sequelize es un ORM moderno que nos permite manipular bases de datos SQL.

GIT - GitHub

Los estándares de la industria para el control de versiones y el alojamiento de proyecto

Patrones

El framework de Nest.js utiliza varios patrones de diseño, entre los que tenemos:

- **Decorator:** Este patrón lo utilizamos para darle funcionalidad a un método de una clase, una clase o incluso los parámetros recibidos de un método.

```
@Controller('users')
export class UserController {
```

Por ejemplo, lo que hacemos es decirle a la clase UserController que adopte la funcionalidad de controlador de Nest en las rutas /users.

```
@Get()
@Roles(Role.Admin)
getUsers() {
  return this.userService.findAll();
}
```

Ahora acá le estamos diciendo que con las request con el método GET en la ruta de ese endpoint /users, entre a ese método. Pero antes tenemos un middleware por el cuál tiene que pasar esa request donde el rol de ese usuario tiene que ser Admin.

```
async createUser(@Body() user: CreateUserDto) {
  return this.userService.create(user);
}
```

En el caso de un decorador de un parámetro en este caso lo que hacemos es decirle que esa información la obtiene del body de la request.

- **Strategy:** Lo utilizamos para implementar estrategias en los middlewares de autenticación y autorización. Uno como estrategia local para obtener el usuario y la password del body y luego devolver el Jwt, y otro para extraer el Jwt del header.

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: process.env.JTW_SECRET || 'secretKey',
    });
  }

  async validate(payload: any) {
    return { email: payload.email, userId: payload.uid, roles: payload.roles };
  }
}
```

```
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super();
  }

  async validate(email: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
```

- **Singleton:** Directamente lo utiliza Nest para la inyección de dependencias, instanciando un solo componente de cada clase.

Buenas Prácticas del Framework

Interceptores

Utilizamos Interceptores para cumplir el principio de única responsabilidad de Solid, y dejar a los servicios solo encargarse de obtener, o gestionar los datos obtenidos de la capa de acceso a datos.

Estos interceptores son:

CreateQueries: Para obtener las queries de la url y crear el correspondiente objeto de la query que va a necesitar el ORM.

```
  const { skip, limit, sort, date, search } = paginationParams;
  //Create Database Query
  const options = {
    offset: skip,
    limit,
    order: sort.map((order) => [order.field, order.by]),
    where: {
      [Op.and]: [
        ...search.map(({ field, value }) => {
          return { [field]: { [Op.like]: `%${value}%` } };
        }),
        ...date.map(({ field, from, to }) => {
          return {
            [field]: { [Op.between]: [from, to || Sequelize.fn('NOW')] },
          };
        }),
      ],
    },
  };

  return options;
```

PackageConstructor: Para validar los datos obtenidos de la request de paquetes si los hoteles, pasajes, seguros y shows son correctos y además obtener el total.

```
@Injectable()
export class ValidatePackages implements NestInterceptor {
  constructor(
    private insuranceService: InsuranceService,
    private ticketService: TicketService,
    private hotelService: HotelService,
    private showService: ShowsService,
  ) {}
  async intercept(
    context: ExecutionContext,
    next: CallHandler<any>,
  ): Promise<Observable<any>> {
    const { body } = context.switchToHttp().getRequest();
    const { insuranceId, ticketId, hotelId, showId } = body;
    let total = 0;
    if (insuranceId) {
      const insurance = await this.insuranceService.findOne(insuranceId);
      total += insurance.amount;
    }
    if (ticketId) {
      const ticket = await this.ticketService.findOne(ticketId);
      total += ticket.amount;
    }
    if (hotelId) {
      const hotel = await this.hotelService.findOne(hotelId);     'hotel'
    }
    if (showId) {
      const show = await this.showService.findOne(showId);
      total += show.amount;
    }
    body.total = total;
    return next.handle();
  }
}
```

Pipes

Utilizamos pipes para validar los datos JSON de las request con Dtos y si no cumplían devolver un response con un campo que contenían todos los errores.

El Dto para la creación de paquetes:

```
export class PackageDto {
  @IsString()
  name: string;

  @IsNumber()
  quantPeople: number;

  @IsOptional()
  @IsNumber()
  hotelId: number;

  @IsOptional()
  @IsNumber()
  ticketId: number;

  @IsOptional()
  @IsNumber()
  insuranceId: number;

  @IsOptional()
  @IsNumber()
  showId: number;
}
```

Guards

Utilizamos guards (más información acá https://docs.nestjs.com/guards) para poder gestionar el acceso a las rutas con métodos de autenticación y roles.

Uno de ellos:

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    console.log(user);
    if (!user) {
      return false;
    }
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}
```

Conclusiones

Este proyecto fue desarrollado entre todos los integrantes del equipo compartiendo nuestros conocimientos y aprendiendo a medida que el desarrollo fue evolucionando desde la primera Historia de Usuario hasta la última funcionalidad incluida. Pudimos comprender el significado de trabajar en proyectos medianamente grandes y tuvimos la oportunidad de implementar diversos patrones de diseño, buenas prácticas y otros conocimientos que hemos visto a lo largo de toda la carrera.

Fuentes

https://docs.nestjs.com/
https://medium.com/geekculture/nest-js-architectural-pattern-controllers-providers-and-modules-406d9b192a3a
https://betterprogramming.pub/building-an-e-commerce-api-using-nestjs-sqlite-and-typeorm-25a7978de666
https://sequelize.org/