01 NEST TypeORM GRIDER

- Proyecto

```
nest new car-pricing
```

- Módulo (--no-spec para que no instale los archivos de testing)

```
nest g res users --no-spec nest g res reports --no-spec
```

- Conexión TypeORM

```
npm i @nestjs/typeorm typeorm sqlite3
```

- En app.module importo TypeOrmModule de @nestjs/typeorm
- Uso .forRoot y le paso el tipo (sqlite), el nombre de la db, las entities, y el synchronize en true

NOTA: algunas importaciones obvias se omitirán para ahorrar espacio

```
import {TypeOrmModule} from "@nestjs/typeorm"

@Module({
  imports: [UsersModule, ReportsModule, TypeOrmModule.forRoot({
    type: 'sqlite',
    database: 'db.sqlite',
    entities: [],
    synchronize: true
  })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

- Levanto el servidor

```
npm run start:dev
```

- Si no tengo errores me crea el archivo db.sqlite

Entity y Repository

- Para la entidad importo 3 decoradores de typeorm
  - **@Entity**, **@Column**, **@PrimaryGeneratedColumn**
- De propiedades tengo el id, mail y password

```
import { Entity, Column, PrimaryGeneratedColumn  } from "typeorm";

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    email: string

    @Column()
    password: string
}
```

- Para conectar la entity voy a users.module e importo el TypeOrmModule y la entity User
- Esta vez uso el .forFeature. Dentro le paso un array con la entity

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';

@Module({
  imports:[TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [UsersService]
})
export class UsersModule {}
```

- Falta el tercer paso que es conectar la entity a la root connection en app.module
- Importo la entity User en app.module, la añado al array de entities dentro del modulo TypeOrmModule

```
import { User } from './users/entities/user.entity';

@Module({
  imports: [UsersModule, ReportsModule, TypeOrmModule.forRoot({
    type: 'sqlite',
    database: 'db.sqlite',
    entities: [User],
    synchronize: true
  })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

- Hago el mismo proceso con la entity reports

- Creo la entidad con esos 3 decoradores

```
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";


@Entity()
export class Report {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    price: number

}
```

- En reports.module uso el **.forFeature** y como argumento le paso un array con la entidad Report
- En app.module importo la entidad Report y la añado al array de entities del **.forRoot TypeOrmModule**
- El mismo proceso anterior pero con la entity Report
- Instalo la extension de SQLite para visualizar la data en el archivo sqlite dentro del editor
- Ctr+Shift+P, buscar sqlite:Open database. Seleccionar car-pricing/db.sqlite y aparece en la izquierda de VSCode SQLITE EXPLORER
- En el CRUD vamos a usar create, save (que vale para insert y update), find, findOne, remove (delete)

Estructura

```
Method and Route              Body or Query String              Description

POST /auth/signup             Body {email-password}             Create a new
user and sign in

POST /auth/signin             Body {email-password}             Sign in as an
existing user

GET /auth/:id                       --------                    Find a user with
given id

GET /auth?email=                    --------                    Find a user with
given email

PATCH /auth/:id               Body {email-password}             Update a user
with given id

DELETE /auth/:id                    ---------                   Delete user with
given id

GET /reports                  QS-make, model, year, mileage,    Get an estimate
for the cars value
```

```
                                    longitude, latitude

POST /reports                  Body{make, model, year, mileage,    Report how much
a vehicle sold for  (update)
                                    longitude, latitude, price}

PATCH /reports/:id             Body {approved}                     Approve or reject
a report submitted by user
```

Body Validation

- Vamos con el método create del controller createUser
- En el decorador le añado la ruta auth **@Controller('auth')**
- Le añado la ruta 'signup' al decorador **@POST('/signup')**
- Para **hacer la validación del dto** debo importar **ValidationPipe** de @nestjs/common en el **main.ts**
    - Le agrego dentro del objeto el whitelist en true

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true
    })
  )
  await app.listen(3000);
}
bootstrap();
```

- Creo el dto create-user.dto
- Para tener los decoradores para hacer las validaciones debo instalar los paquetes

> npm i class-validator class-transformer

```
import { IsEmail, IsString, MinLength } from "class-validator"

export class CreateUserDto {

    @IsEmail()
    email: string

    @IsString()
    @MinLength(6)
```

```
    password: string
}
```

- Uso el decorador **@Body** para extraer el body en el controller
- Notar que en el constructor está el servicio **userService** inyectado
- users.controller

```
import { CreateUserDto } from './dto/create-user.dto';

@Controller('auth')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post('/signup')
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }
}
```

Creando y salvando un usuario

- En el users.service importo **@Repository** de 'typeorm' y **@InjectRepository** de @nestjs/typeorm
- Importo la entidad **User**
- Inyecto el repositorio en el constructor
- Notar que el servicio tiene el decorador **@Injectable**

```
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';

@Injectable()
export class UsersService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  create(createUserDto: CreateUserDto) {

    const user = this.userRepository.create(createUserDto)

    return this.userRepository.save(user)
  }
}
```

- El método create crea la instancia pero es el método save el que guarda la data
- Uso el método POST y añado en el body el email y el password

---

Hooks de TypeORM

- Puedo usar **AfterInsert** para ejecutar una función después de realizar una inserción con la entity **User**
- También tengo los hooks **AfterRemove** y **AfterUpdate**

```
import { AfterInsert, AfterRemove, AfterUpdate, Entity, Column,
PrimaryGeneratedColumn  } from "typeorm";

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    email: string

    @Column()
    password: string

    @AfterInsert()
    logInsert(){
        console.log('Inserted user with id', this.id)
    }

    @AfterRemove()
    logRemove(){
        console.log('Deleted user with id', this.id)
    }

    @AfterUpdate()
    logUpdate(){
        console.log('Updated user with id', this.id)
    }
}
```

- Los métodos find y findOne del usersService

```
async findOne(id: number) {
  const user = await this.userRepository.findOneBy({id});
  if(!user){
    throw new NotFoundException(`User with id ${id} not found`)
  }
  return user
}
```

```
async find(email: string){
  return await this.userRepository.find({where: {email}})

  //aqui no me sirve la validación con el if
  //si no encuentra el user devuelve un status 200 y un array vacío

}
```

- Si quiero encontrar por mail tengo que apuntar a http://localhost:3000/auth?email=correo@gmail.com con GET
- Si envío un email que no existe me da un status 200 y me devuelve un array vacío
- Para el update, el update-user dto ya lleva el Partial, por lo que las propiedades que le pase son opcionales

```
async update(id: number, updateUserDto: UpdateUserDto) {
  const user = await this.findOne(id)


  Object.assign(user, updateUserDto)

  return await this.userRepository.save(user)
}
```

- El delete (que es un remove, si fuera delete sería para una propiedad en concreto)

```
async remove(id: number) {
  const user = await this.findOne(id)

  return await this.userRepository.remove(user)
}
```

- Vamos con el controller a hacer los endpoints!!

```
import { Controller, Get, Post, Body, Patch, Param, Delete, Query } from
'@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';

@Controller('auth')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post('/signup')
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }
```

```
  @Get()
  findAll(@Query('email') email: string) {
    return this.usersService.find(email);
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(+id);
  }

  @Patch(':id')
  update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.usersService.update(+id, updateUserDto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.usersService.remove(+id);
  }
}
```

02 NEST Custom Data Serialization GRIDER

- No queremos incluir el password en la respuesta y además debería estar encriptado
- Entonces, debemos asegurarnos que cuando devolvamos un usuario, no devolvamos el password
- Para ello usaremos un **decorador interceptor**, **Class Serializer Interceptor**
    - devuelve una instancia en un objeto plano basado en algunas reglas
- En la entidad User importo **Exclude de class-transformer**
- Se lo añado a la propiedad password

```
import { AfterInsert, AfterRemove, AfterUpdate, Entity, Column,
PrimaryGeneratedColumn  } from "typeorm";
import { Exclude } from "class-transformer";

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    email: string

    @Column()
    @Exclude()   //le añado el decorador Exclude para que no lo devuelva en la
respuesta
    password: string
```

```
    @AfterInsert()
    logInsert(){
        console.log('Inserted user with id', this.id)
    }

    @AfterRemove()
    logRemove(){
        console.log('Deleted user with id', this.id)
    }

    @AfterUpdate()
    logUpdate(){
        console.log('Updated user with id', this.id)
    }
}
```

- En el user.controller importo **UseInterceptor** y **ClassSerializerInterceptor** de @nestjs/common
- Lo uso en el findOne

```
@UseInterceptors(ClassSerializerInterceptor)
@Get(':id')
findOne(@Param('id') id: string) {
return this.usersService.findOne(+id);
}
```

- Esta solución (recomendada por NEST) quizá no es la mejor
- Si quiero dos endpoints diferentes, y en uno de ellos más info del usuario que en el otro, Exclude no me va a servir
- En lugar de usar el Exclude vamos a crear un **Custom Interceptor** con un **UserDTO** que describa como serializar un User para la ruta en particular
- Habrá otro DTO para serializar otra ruta

Cómo construir Interceptors

- Los interceptors pueden trabajar con las **requests y/o las responses**
- Son parecidos a los middlewares
- El interceptor se puede aplicar **en un handler** del controller, o también a nivel de la clase @Controller para que **afecte a todos los handlers**, **o globalmente**
- Se crean con **una clase**
- Necesitamos el **context: ExecutionContext** con información de la request y el **next: CallHandler**, que es como una referencia al handler del controller
- Ya podemos sacar el **@Exclude** de la entity User
- Creo en */src/interceptors/serialize.interceptor.ts*, serialize porque va a serializar un objeto a JSON
- Hago los imports de @nestjs/common
    - UseInterceptors
    - NestInterceptor

    - ExecutionContext
    - CallHandler
- También importo Observable de rxjs
- Importo map de rxjs/operators
- Importo plainToInstance de class-transformer
- Uso **implements NestInterceptor** para que cumpla con la interfaz de interceptor de Nest
- Creo el metodo **intercept** al que le paso el ExecutionContext y el CallHandler
    - Devuelve algo de tipo Observabe o una promesa de tipo Observable, de genérico de momento le pongo any

NOTA: obtengo la información poniendo el cursor encima

- serialize.interceptor.ts

```
import { UseInterceptors, NestInterceptor, ExecutionContext, CallHandler } from
"@nestjs/common";
import { Observable } from "rxjs";
import { map } from "rxjs";
import { plainToInstance } from "class-transformer";

export class SerializeInterceptor implements NestInterceptor {

    intercept(context:ExecutionContext, next:CallHandler): Observable<any>{


        console.log('Esto va antes del handler del controller', context)

        return next.handle().pipe(
            map((data: any)=>{

                console.log('Esto va antes de que la response sea enviada')
                return data
            })
        )
    }
}
```

- Dentro de intercept puedo escribir código antes de que sea procesado por el handler del controller
- Dentro del map (dentro del next.handle().pipe()) puedo hacer correr algo antes de que la response sa enviada
- Entonces en el cuerpo de intercept trabajo con la incoming data, y en el callback del map en next.handle().pipe() la outcoming data, para después retornarla
- Para usar el interceptor lo importo en el controlador y lo uso en el handler findUser
- Para observar los console.logs en orden, coloco un console.log en el handler

```
@UseInterceptors(SerializeInterceptor)
@Get(':id')
async findUser(@Param('id') id: string) {
```

```
    return await this.usersService.findOne(+id);
 }
```

- Lo primero imprime esto va antes del handler... luego el ExecutionContext, luego imprime desde el handler y finalmente esto va antes que la response....

Serialization en el Interceptor

- Vamos a usar un dto que describa como serializar (pasar a JSON) a un user en este handler en particular
- Es transformar la instancia de User en un dto, y este dto en un JSON
- Expose sirve para exponer explicitamente esas propiedades
- user.dto.ts

```
import { Expose } from "class-transformer"

export class UserDto{

    @Expose()
    id: number

    @Expose()
    email: string
}
```

- Importo el dto en el interceptor
- Luego se hará una refactorización para no hardcodear el dto en el interceptor
- Para transformar la data en el userDto usaré **plainToInstance**
- Le paso el dto, la data y un objeto con excludeExtraneousValues en true, de esta manera solo va a extraer en el JSON las propiedades con el **@Expose**
- Otras propiedades serán excluidas

```
import { UseInterceptors, NestInterceptor, ExecutionContext, CallHandler } from
"@nestjs/common";
import { Observable } from "rxjs";
import { map } from "rxjs";
import { plainToInstance } from "class-transformer";
import { UserDto } from "src/users/dto/user.dto";

export class SerializeInterceptor implements NestInterceptor {

    intercept(context:ExecutionContext, next:CallHandler): Observable<any>{

        return next.handle().pipe(
            map((data: any)=>{

                return plainToInstance(UserDto, data, {
                    excludeExtraneousValues: true
```

```
                })

            })
        )
    }
}
```

- Ahora si hago una petición a localhost:3000/auth/1 no me devuelve el password en el objeto de retorno
- Vamos a hacer el interceptor más reutilizable, por si queremos extraer fotos, más datos o lo que sea

---

Customize Interceptor

- Importo el dto en el controller
- Lo que necesito es pasarle en el constructor de SerializeInterceptor el UserDto
- users.controller

```
@UseInterceptors(new SerializeInterceptor(UserDto))
@Get(':id')
async findUser(@Param('id') id: string) {
   return await this.usersService.findOne(+id);
}
```

- Agrego el constructor en el Interceptor y le paso el dto de tipo any (de momento)

```
export class SerializeInterceptor implements NestInterceptor {

    constructor(private dto:any){}

    intercept(context:ExecutionContext, next:CallHandler): Observable<any>{

        return next.handle().pipe(
            map((data: any)=>{

                return plainToInstance(this.dto, data, {
                    excludeExtraneousValues: true
                })
            })
        )
    }
}
```

- En este momento este controller necesita importar @UseInterceptors, SerialInterceptor y UserDto
- Vamos a refactorizarlo para no escribir tanto código

Wrapping the interceptor in a Decorator

- Los **decoradores son simples funciones**
- En el interceptor exporto una función que voy a llamar Serialize
- Dentro voy a retornar exactamente lo que coloqué en el controlador
- Escribo la función **fuera** del interceptor (en la cabecera)
- serialize.interceptor.ts

```
export function Serialize(dto: any){
    return UseInterceptors(new SerializeInterceptor(dto))
}
```

- En el users.controller hago uso del decorador. Lo añado sin más y le paso el dto!

```
@Serialize(UserDto)
@Get(':id')
async findUser(@Param('id') id: string) {
    return await this.usersService.findOne(+id);
}
```

Controller-Wide Serialization

- Puedo aplicar el decorador **@Serialize** que acabo de crear **a nivel de controlador**
- Al fin y al cabo todos los otros endpoints devuelven un user de uno u otro modo
- users.controller.ts

```
@Controller('auth')
@Serialize(UserDto)
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  (...)
  }
```

- Ahora los users que devuelven todos los handlers son sin el password
- Pero puedo necesitar otro tipo de respuesta (otro dto) por lo que colocaría **@Serialize(dto)** en el handler especifico
- Lo dejo en el controller
- Vamos con el tipado del dto

A Bit of Type Safety Around Serialize

- Realizar tipado en decoradores es bastante **desafiante**

- Typescript **no da soporte** a tipado en decoradores, por lo general
- Recuerda que en el callback de la función map del interceptor data es de tipo any
- Podemos hacer que al menos, lo que sea que le pase a **@Serialize** sea **una clase**
- Creo una interface fuera del interceptor

```
interface ClassConstructor{
    new(...args: any[]): {}
}
```

- Esta interfaz viene a decir **cualquier clase que me pases está bien**
- Le paso el tipo al dto, tanto en el constructor del interceptor como en el decorador **@Serialize**

03NEST AUTH GRIDER

Authentication Overview

- /auth/signup y /auth/signin van a usar autenticación
- Pongamos que para el resto de rutas no es necesario autenticación
- Con el signup hay que verificar si el correo está en uso (en ese caso devolver un error)
    - Encriptar el password
    - Guardar el usuario en la db
    - Devolver una cookie que contiene el user id
        - El browser automáticamente almacena la cookie y la adjudica a siguientes requests
- Cuando voy a POST /reports está la cookie userId=34 con info en un objeto
    - Compruebo la data en la cookie
    - Miro el userId para saber quien está haciendo la request
        - Que el usuario que hace la request sea el mismo que el que ha ingresado
- Hay que añadir dos nuevos métodos en el servicio: signup y signin
- Para ello podemos crear un nuevo servicio llamado Auth Service que interactue con Users Service
- Para una aplicación pequeña quizá no fuera necesario, pero a medida que crezca y necesite otros métodos como resetear el password, establecer preferencias, etc. si será necesario tener su propio servicio de auth

---

Reminder on Service Setup

- Haciendo un pequeño diagrama de dependencias
    - Users Controller va a usar Users Service y Auth Service
    - Auth Service va a usar Users Service
    - Users Service va a usar Users Repository
- Para hacer la inyección de dependencias se utiliza el constructor, y se añade el servicio a la lista de prviders en el módulo
- Creo el servicio dentro de /users/auth.service.ts
- Importo **@Injectable** de @nestjs/common como decorador de la clase
- Importo **UsersService** y lo inyecto

```
import { Injectable } from "@nestjs/common";
import { UsersService } from "./users.service";

@Injectable()
export class AuthService{
    constructor(
        private usersService: UsersService
    ){}
}
```

- Lo añado a la lista de providers en users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { AuthService } from './auth.service';

@Module({
  imports:[TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [UsersService, AuthService]
})
export class UsersModule {}
```

- Habrá que inyectar el AuthService en el controller

Implementing Signup Functionality

- En signup compruebo si el email está en uso
- Uso .length para hacer la verificación por que devuelve un array de la promesa
- Encripto el password, importo randomBytes (para generar el salt) y scrypt (para hashear, lo renombro a _scrypt para codificarlo como promesa) del paquete 'crypto'
    - scrypt es de naturaleza asincrona, para evitar trabajar con callbacks usaremos promisify del paquete 'util'
    - Typescript no tiene idea de cual es el valor de retorno de scrypt cuando uso promisify
    - Para generar el salt, randomBytes me devuelve un buffer (similar a un array con unos y ceros), con toString('hex') lo transformo a un string hexadecimal (16 caracteres de letras y numeros)
    - Creo el hash, le paso el password, el salt, y 32 son los bytes de salida (standard), pueden ser más
    - Si miro el tipo de retorno de hash, Typescript dice unknown, no tiene ni idea
    - Lo meto entre paréntesis y le digo a Typescript que es un Buffer
    - Junto el salt y el hash separados por un punto
- Creo un nuevo usuario, lo guardo y lo retorno

```
import { BadRequestException, Injectable } from "@nestjs/common";
import { UsersService } from "./users.service";
import { randomBytes, scrypt as _scrypt } from "crypto";
import { promisify } from "util";

const scrypt = promisify(_scrypt)

@Injectable()
export class AuthService{
    constructor(
        private usersService: UsersService
    ){}

    async signup(email: string, password: string){
        const users = await this.usersService.find(email)

        if(users.length) throw new BadRequestException('email in use')

        const salt = randomBytes(8).toString('hex')

        const hash = (await scrypt(password, salt, 32 )) as Buffer

        const result = salt + '.' + hash.toString('hex')

        const user = await this.usersService.create(email, result)

        return user
    }

    signin(){}
}
```

- Importo el AuthService en el controller, lo inyecto
- En el POST, en lugar de usar usersService uso authService y le paso el email y el password

```
import { Controller, Get, Post, Body, Patch, Param, Delete, Query } from
'@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { UserDto } from './dto/user.dto';
import { Serialize } from 'src/interceptors/serialize.interceptor';
import { AuthService } from './auth.service';

@Controller('auth')
@Serialize(UserDto)
export class UsersController {
  constructor(private readonly usersService: UsersService,
              private readonly authService: AuthService) {}

  @Post('/signup')
```

```
  create(@Body() createUserDto: CreateUserDto) {
    return this.authService.signup(createUserDto.email, createUserDto.password)
  }

  (...)
}
```

- Pequeña modificación en el método create del UsersService

```
async create(email: string, password: string) {
  const user = this.userRepository.create({email, password})

  return await this.userRepository.save(user)
}
```

- Si ahora hago ctrl+shift+p SQLite: Open Data Base, car-price, puedo observar que el password está encriptado
- Para el signin es similar. Recibo un email y un password, debo encontrar ese usuario y si no devolver un error
- Mirar si el password hace match, etc

Handling User Sign In

- En usersService tengo findOne que requiere el id
- find requiere el email, pero puede devolver varios users, por lo que devuelve un array
- Uso desestructuración de asrrays para extraer un usuario
- Verifico que existe el usuario
- Tengo el password dividido por un punto del salt y el hash. Uso split y desestructuración para obtenerlos
- Creo de la misma manera el hash con el salt, y hago la comparación de los hashes

```
async signin(email: string, password: string){
    const [user] = await this.usersService.find(email)

    if(!user) throw new NotFoundException('user not found')

    const [salt, storedHash] = user.password.split('.')

    const hash = (await scrypt(password, salt, 32)) as Buffer

    if(storedHash !== hash.toString('hex')){
        throw new BadRequestException('bad password')
    }
      return user
}
```

- Creo otro handler POST en el users.controller para el signin
- Uso el mismo dto de createUserDto, ya que necesito el email y el password

```
@Controller('auth')
@Serialize(UserDto)
export class UsersController {
  constructor(private readonly usersService: UsersService,
              private readonly authService: AuthService) {}

  @Post('/signup')
  createUser(@Body() body: CreateUserDto) {
    return this.authService.signup(body.email, body.password)
  }

  @Post('/signin')
  signIn(@Body() body: CreateUserDto){
    return this.authService.signin(body.email, body.password)
  }
}
```

- Si coloco el email y password correctos me devuelve el id y el email del usuario
- Vamos con el tema de la Cookie-Session para almacenar el id

---

Setting up Sessions

- Vamos a enviar un header llamado Cookie con un string que luce como varias letras y números random
- La librería Cookie-Session mira el header de la cookie, que contiene el string encriptado
- Cookie-Sessions decodifica el string resultando en un objeto (Session Object)
- Tenemos acceso al objeto en el handler usando un decorador
- Podemos añadir, remover, cambiar propiedades en el objeto
- Cookie-Session ve la sesión actualizada y lo vuelve a una string encriptada
- El string encriptado (que incluye la actualización del objeto) es devuelto en la Set-Cookie de los headers en la response
- Instalo la librería

```
npm i cookie-session @types/cookie-session
```

- En el main debo configurar el cookie middleware
- No acepta el import del ECMAS6 por lo que se usa require
- Hago uso de app.use y dentro del objeto de cookieSession le paso la propiedad keys con un array de un string
- Este string se usará para encriptar la información de la cookie. Más adelante se configurará como una variable de entorno
- main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
```

```
import { ValidationPipe } from '@nestjs/common';
const cookieSession = require('cookie-session');

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(cookieSession({
    keys:['lalala']
  }))

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true
    })
  )
  await app.listen(3000);
}
bootstrap();
```

Changing and Fetching Session Data

- El decorador Session funciona con la librería Cookie-Session
- Me permitirá acceder al objeto de la cookie
- Ejemplo, dónde actualizaría el color del objeto Session en la cookie y otro handler donde retorno el color almacenado en la cookie

```
@Get('/colors/:color')
setColor(@Param('color') color: string, @Session() session: any){
  session.color = color
}

@Get('/colors')
getColors(@Session() session: any){
  return session.color
}
```

Signin in a User

- Dentro del signup y signin vamos a extraer el id y almacenarlo como userId en la cookie
- En el handler del controller uso **@Session** para extraer la session. Lo tipo como any de momento
- Vuelvo los handlers async, remuevo el return del servicio para guardar el resultado en una variable user
- Ahora puedo colocar el user.id en la session

```
@Post('/signup')
async createUser(@Body() body: CreateUserDto, @Session() session: any) {
  const user = await this.authService.signup(body.email, body.password)
```

```
  session.userId = user.id
  return user
}

@Post('/signin')
async signIn(@Body() body: CreateUserDto, @Session() session: any){
  const user = await this.authService.signin(body.email, body.password)
  session.userId = user.id
  return user
}
```

Getting the Current User

- Creo un nuevo handler en users.controller de tipo GET con el endpoint whoami

```
@Get('/whoami')
whoAmI(@Session() session: any){
  return this.usersService.findOne(session.userId)
}
```

Signing Out a User

- Creo otro handler POST para el signout

```
@Post('/signout')
signOut(@Session() session: any){
  session.userId = null
}
```

- De esta manera el método findOne de whoami (al recibir null) no actúa de la manera esperada
- Vamos a retocar el método findOne para que en el caso de recibir null actúe adecuadamente

```
async findOne(id: number) {
    if(!id) return null

    const user = await this.userRepository.findOneBy({id});
    if(!user){
      throw new NotFoundException(`User with id ${id} not found`)
    }
    return user
  }
```

- Si hago el POST signout, al hacer el GET whoami me devuelve un 200 pero no retorna nada

Two Automation Tools

- Relacionado con el handler signIn, ciertos handlers deben rechazar la request si el usuario no ha ingresado en el sistema
  - Para este caso usaremos un **Guard**. Protege la ruta del acceso si no se está autenticado
- signOut automaticamente debería decirle al handler quien hay ingresado en el sistema
  - Para este caso usaremos un **Interceptor + Decorator**. Es más complicado por lo que empezaremos por este

---

Custom Param Decorators

- Quiero crear un **Custom Decorator** para extraer el user sin usar @Session y todo el rollo
- Creo en users/decorators/current-user.decorator.ts
- Importo de @nestjs/common:
  - createParamDecorator
  - ExecutionContext
- Esqueleto de mi Custom Decorator

```
import { createParamDecorator, ExecutionContext } from "@nestjs/common";

export const CurrentUser =  createParamDecorator(
    (data: any, ctx: ExecutionContext)=>{

    }
)
```

- Necesito también un interceptor. Porqué?
- Lo que sea que le pase al decorador @CurrentUser('aksdakds') en el handler, va a estar disponible en el parámetro data:any de la función que hay dentro del CurrentUser
- Como mi decorador no necesita ningún parámetro, pongo que data será de tipo never.
- Esto marcará error si le pongo algún parámetro al decorador
- En el Custom Decorator necesito el objeto session y el usersService
- Para extraer el id uso el ctx: ExecutionContext, para esto no hay problema
- Cuando se complica es cuando quiero usar el servicio para encontrar al usuario por el id
- UsersService es inyectable y a la vez usa el userRepository que también es inyectable

```
import { createParamDecorator, ExecutionContext } from "@nestjs/common";

export const CurrentUser =  createParamDecorator(
    (data: never, ctx: ExecutionContext)=>{

        const request = ctx.switchToHttp().getRequest()

        const id = request.session.userId

        return id // Si lo que quisiera es retornar el id podría hacerlo así
```

```
    }
)
```

- No puedo usar el servicio. Los Param Decorators viven fuera del sistema de inyección de dependencias
- No puedo usar directamente UsersService
- **El Interceptor va a resolver este problema**
- Creo un interceptor que reciba el id para que interactue con UsersService, obtenga el usuario y retornarlo en el decorador

---

Comunicating from Interceptor to Decorator

- Creo users/interceptors/current-user.interceptor.ts
- Importo de @nestjs/common:
  - NestInterceptor
  - ExecutionContext
  - CallHandler
  - Injectable
- Importo el UsersService
- Una vez tengo el usuario, para pasárselo al decorador lo meto en la request

```
import { NestInterceptor, ExecutionContext, CallHandler, Injectable } from
"@nestjs/common";
import { UsersService } from "../users.service";

@Injectable()
export class CurrentUserInterceptor implements NestInterceptor{

    constructor(private usersService: UsersService){}

    async intercept(ctx: ExecutionContext, handler: CallHandler){

        const request = ctx.switchToHttp().getRequest()

        const {userId} = request.session || {} //puede ser que venga vacío, para
que prosiga con el código

        if(userId){
            const user= await this.usersService.findOne(userId) //uso el servicio
para encontrar el usuario
            //para comunicarme con el decorador, coloco el user en la request
            request.currentUser = user
        }

        return handler.handle() //esto es "sigue adelante y ejecuta el handler"
    }
}
```

- Ahora voy al decorador CurrentUser y retorno el user de la request

```
import { createParamDecorator, ExecutionContext } from "@nestjs/common";

export const CurrentUser =  createParamDecorator(
    (data: never, ctx: ExecutionContext)=>{
        const request = ctx.switchToHttp().getRequest()

        return request.currentUser
    }
)
```

- El interceptor que he creado debe correr antes que el decorador
- Puedo implementarlo de dos maneras
- Para usar inyección de dependencias debo declararlo en los providers del módulo

```
@Module({
  imports:[TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [UsersService, AuthService, CurrentUserInterceptor]
})
export class UsersModule {}
```

- Para usarlo en el users.controller importo UseInterceptors de @nestjs/common
- Lo pongo a nivel de controlador

```
@Controller('auth')
@Serialize(UserDto)
@UseInterceptors(CurrentUserInterceptor)
export class UsersController {
  constructor(private readonly usersService: UsersService,
              private readonly authService: AuthService) {}

  @Get('/whoami')
  whoAmI(@CurrentUser() user: User){
    return user
  }
}
```

- Para que funcione he tenido que importar CurrentUserInterceptor, UseInterceptors, añadirlo al controlador...
- Si tuviera quince controladores sería mucho código duplicado
- **Hay otra manera**

Globally Scoped Interceptors

- Aplicaremos el interceptor globalmente

- En users.module importo **APP_INTERCEPTOR** de '@nestjs/core'
- Envuelvo el CurrentUserInterceptor en un objeto y lo coloco dentro de la propiedad useClass
- Le añado el provide: APP_INTERCEPTOR al objeto

```
import { CurrentUserInterceptor } from './interceptors/current-user.interceptor';
import {APP_INTERCEPTOR} from '@nestjs/core'

@Module({
  imports:[TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [
    UsersService,
    AuthService,
    { provide: APP_INTERCEPTOR,
      useClass: CurrentUserInterceptor}
    ]
})
export class UsersModule {}
```

- Ahora falta trabajar con el sign in y rechazar requests si el usuario no se ha loggeado

---

Preventing Access with Authentication Guards

- Los Guards son **una clase** que a través de **canActivate()** devuelve un truthly o falsy value si el usuario puede acceder o no
- Cualquier valor de retorno como null, undefined, etc, retornará un falsy y rechazará la petición automáticamente
- El Guard puede estar a nivel de **aplicación, de controller o de handler**
- Creo en src/guards/auth.guard.ts
- Importo de @nestjs/common
  - CanActivate
  - ExecutionContext
- Implemento la interfaz CanActivate a la clase
- Me pide la función canActivate que puede devolver un boolean, una promesa de tipo boolean, o un Observable de tipo boolean

```
import { CanActivate, ExecutionContext } from "@nestjs/common";
import { Observable } from "rxjs";

export class AuthGuard implements CanActivate{

    canActivate(ctx: ExecutionContext): boolean | Promise<boolean> |
Observable<boolean> {
     const request = ctx.switchToHttp().getRequest()

     return request.session.userId
```

```
    }
}
```

- En users.controller importo el AuthGuard. También el useGuards de @netsjs/common
- Lo uso en un handler

```
@Get('/whoami')
@UseGuards(AuthGuard)
whoAmI(@CurrentUser() user: User){
  return user
}
```

- Si no estoy logeado me devuelve un 403

---

04 NEST UNIT TESTING GRIDER

- Unit testing -> asegurarse de que métodos en una clase funcionan adecuadamente
- Integration Testing -> testear el flow entero de la aplicación
- En el directorio test tengo app.e2e-spec.ts que es un test end-to-end. Comprueba que el servidor devuelva un 200
- Vamos a empezar el testing con el auth.service
- Debemos asegurarnos que signin y signup retornen un output adecuado
    - Para ello necesitamos una copia de UsersService, que depende de UsersRepository que consulta la DB de SQLite
    - Usaremos la inyección de dependencias
    - Vamos a hacer una copia fake de UsersService
        - Será una clase temporal que definiremos en el archivo de testing con los métodos que necesitemos
        - Crearemos una instancia de AuthService para que use este servicio fake
        - Para esto crearemos un Container con inyección de dependencias, con el AuthService y el UsersService fake (una clase con los métodos de UsersService)

Testing Setup

- Importo:
    - Test de @nestjs/testing
    - AuthService
    - UsersService
- Debo crear el módulo DI Container (DI=Dependency Injection)
- Luego debo crear una copia del AuthService
- Debo resolver las dependencias del AuthService (UsersService), pero puedo comprovar si se ha definido adecuadamente
- Luego refactorizaré este código

- users/auth.service.spec.ts

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";

it('can create an instance of auth service', async()=>{
    const module = await Test.createTestingModule({
        providers: [AuthService]
    }).compile()

    const service = module.get(AuthService)

    expect(service).toBeDefined()
})
```

- Para correr el test uso npm run test:watch
- El test falla porque el módulo contiene UsersService (dependencia de AuthService)
- Vamos a crear una copia fake de UsersService con los métodos find y create
- Lo añado al array de providers dentro de un objeto con las propiedades provide y useValue

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";

it('can create an instance of auth service', async()=>{

    const fakeUsersService = {
        find: ()=> Promise.resolve([]),
        create: (email: string, password: string)=> Promise.resolve({id: 1, email,
password })
    }


    const module = await Test.createTestingModule({
        providers: [AuthService,
        {
            provide: UsersService,
            useValue: fakeUsersService
        }]
    }).compile()

    const service = module.get(AuthService) //crea una instancia de AuthService

    expect(service).toBeDefined()
})
```

- Ahora pasa el test porque hemos resuleto la dependencia

NOTA: Para acelerar los tests cambiar el script a "jest --watch --maxWorkers=1"

Testing is a little bit confusing

- Veamos que pasa en el array de providers
- Este array es una lista de clases que queremos inyectar en el DI Container
- Con el objeto le estamos diciendo que cualquiera que pregunte por UsersService dale el objeto fakeUsersService que es {find,create}

Getting Typescript to Help with Mocks

- El UsersService también tiene el método update, remove...aunque find y create son los únicos usados por AuthService on signin y signup
- Usamos Promise.resolve, porque en la vida real esto sería una consulta a la DB. Crea una promesa y la resuelve inmediatamente con el valor dado
- Ahora mismo si hago const arr = await fakeUsersService.fin() me devuelve un array vacío
- Estamos pasando fakeUsersService como UsersService, pero el primero no contiene todos los métodos del segundo
- Typescript no nos está ayudando a implementar todos los métodos como haría una interfaz
- Tampoco me está ayudando con el valor de retorno de create y find de fakeUsersService
- Solucionemoslo!
- Uso el tipado con Partial para no tener que implementar todos los métodos
- Se supone que create debe devolver una instancia de User, debería implementar logInsert, logUpdate y logRemove
- Como no quiero implementar estos métodos en el objeto uso as User

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";
import { User } from "./entities/user.entity";

it('can create an instance of auth service', async()=>{

    const fakeUsersService: Partial<UsersService> = {
        find: ()=> Promise.resolve([]),
        create: (email: string, password: string)=> Promise.resolve({id: 1, email,
password } as User)
    }

    const module = await Test.createTestingModule({
        providers: [AuthService,
        {
            provide: UsersService,
            useValue: fakeUsersService
        }]
    }).compile()
```

```
    const service = module.get(AuthService)

    expect(service).toBeDefined()
})
```

Improving File Layout

- Como voy a usar ek servicio en todos los tests, para no repetir código usor beforeEach
- Para poder usar service (al estar un scope diferente) defino la variable fuera del scope
- Lo envuelvo todo en un bloque describe

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";
import { User } from "./entities/user.entity";


describe('auth service testing', ()=>{

    let service: AuthService;

    beforeEach(async ()=>{

        const fakeUsersService: Partial<UsersService> = {
            find: ()=> Promise.resolve([]),
            create: (email: string, password: string)=> Promise.resolve({id: 1,
email, password } as User)
        }

        const module = await Test.createTestingModule({
            providers: [AuthService,
            {
                provide: UsersService,
                useValue: fakeUsersService
            }]
        }).compile()

        service = module.get(AuthService)
    })

    it('can create an instance of auth service', async()=>{

        expect(service).toBeDefined()
    })
})
```

Ensuring Password Gets Hashed

- signup recibe un email y un password, comprueba si el usuario existe, genera el salt y el hash, crea un nuevo usuario con el email y el password encriptado
- Por último retorna el user
- Queremos que find devuelva un array vacío, porque en este caso, en el signup significa que no hay usuario con ese email
- Como espero que el password esté encriptado, no me debería devolver el mismo password
- Debería poder obtener el salt y el hash haciendo la división por el punto con split

```
it('creates a new user with a hashed and salted password', async ()=>{
       const user =  await service.signup('email@google.com', '123456')

       expect(user).not.toEqual('123456')

       const [salt, hash] = user.password.split('.')

       expect(salt).toBeDefined()
       expect(hash).toBeDefined()
    })
```

- El test pasa

Throws an Error if user signs up

- Hago la misma jugada, para usar fakeUsersService lo declaro en el escope del describe y lo tipo
- En el signup le paso el mismo email y password que he declarado en el find del fakeUsersService
- Uso rejects.toThrow para lanzar BadRequestException

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";
import { User } from "./entities/user.entity";
import { BadRequestException } from "@nestjs/common";

describe('auth service testing', ()=>{

    let service: AuthService;
    let fakeUsersService: Partial<UsersService>
    beforeEach(async ()=>{

        fakeUsersService = {
            find: ()=> Promise.resolve([]),
            create: (email: string, password: string)=> Promise.resolve({id: 1,
email, password } as User)
        }

        const module = await Test.createTestingModule({
            providers: [AuthService,
            {
```

```
                provide: UsersService,
                useValue: fakeUsersService
            }]
        }).compile()

        service = module.get(AuthService)
    })

    it('can create an instance of auth service', async()=>{

        expect(service).toBeDefined()
    })

    it('creates a new user with a hashed and salted password', async ()=>{
       const user =  await service.signup('email@google.com', '123456')

       expect(user).not.toEqual('123456')

       const [salt, hash] = user.password.split('.')

       expect(salt).toBeDefined()
       expect(hash).toBeDefined()
    })

    it('throws an error if user signs up with email that is in use', async()=>{
        fakeUsersService.find = ()=> Promise.resolve([{id:1, email:
"email@google.com", password: '123456'} as User])
        await
expect(service.signup("email@google.com",'123456')).rejects.toThrow(BadRequestExce
ption)
    })
})
```

Throws if signin is called with an unused email

- Como el fakeUsersService pasa por el beforeEach, se reinicia.
- Quiero decir que en el scope global, el find del fakeUsersService devuelve un arreglo vacío y eso es precisamente lo que nos interesa

```
 it('throws if signin is called with an unused email', async()=>{

       await  expect(service.signin('correo@email.com', '123456'))
                            .rejects.toThrow(NotFoundException)

    })
```

Invalid password

```
it('invalid password returns error', async ()=>{
    fakeUsersService.find= ()=> Promise.resolve([{email: 'correo@gmail.com',
password: '123456'} as User])

    expect(service.signin('correo@gmail.com',
'uhiuh122792')).rejects.toThrow(BadRequestException)
})
```

More intelligent mocks

- Para hacer más realista el fakeUsersService, vamos a hacer que el método create guarde el email y password en un array
- Y el find busque en este array
- Así los passwords harán match y voy a poder testar si le doy un password correcto, que me devuelva un user
- Creo una variable que será el arreglo de users
- En el find filtro por el email y devuelvo el usuario filtrado del array de users en el resolve
- En create creo el usuario, lo subo al array con push y lo retorno en el resolve

```
describe('auth service testing', ()=>{

    let service: AuthService;
    let fakeUsersService: Partial<UsersService>
    beforeEach(async ()=>{

        const users: User[] = []

        fakeUsersService = {
            find: (email:string)=> {
                const filteredUsers= users.filter(user=> user.email === email)
                return Promise.resolve(filteredUsers)
            },
            create: (email:string, password: string)=>{
                    const user = {id: Math.floor(Math.random() * 9999), email,
password} as User
                    users.push(user)
                    return Promise.resolve(user)
            }
        }
        //resto del código
     })
 })
```

- Ahora el test de dar un password correcto si da match

```
it('returns a user if a valid password is provided', async()=>{
    await service.signup('correo@gmail.com', '123456')
    const user = await service.signin('correo@gmail.com', '123456')

    expect(user).toBeDefined()
})
```

Refactor to use intelligent mocks

- Debo sustituir el userFakeService por el service.signup como corresponda

```
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";
import { User } from "./entities/user.entity";
import { BadRequestException, NotFoundException } from "@nestjs/common";


describe('auth service testing', ()=>{

    let service: AuthService;
    let fakeUsersService: Partial<UsersService>
    beforeEach(async ()=>{
        const users: User[] = []

        fakeUsersService = {
            find: (email:string)=> {
                const filteredUsers= users.filter(user=> user.email === email)
                return Promise.resolve(filteredUsers)
            },
            create: (email:string, password: string)=>{
                    const user = {id: Math.floor(Math.random() * 9999), email,
password} as User
                    users.push(user)
                    return Promise.resolve(user)
            }
        }

        const module = await Test.createTestingModule({
            providers: [AuthService,
            {
                provide: UsersService,
                useValue: fakeUsersService
            }]
        }).compile()

        service = module.get(AuthService)
    })
```

```
    it('can create an instance of auth service', async()=>{

        expect(service).toBeDefined()
    })

    it('creates a new user with a hashed and salted password', async ()=>{
       const user =  await service.signup('email@google.com', '123456')

       expect(user).not.toEqual('123456')

       const [salt, hash] = user.password.split('.')

       expect(salt).toBeDefined()
       expect(hash).toBeDefined()
    })

    it('throws an error if user signs up with email that is in use', async()=>{
        await service.signup('email@google.com', '123456')
        await
expect(service.signup("email@google.com",'123456')).rejects.toThrow(BadRequestExce
ption)
    })

    it('throws if signin is called with an unused email', async()=>{

       await  expect(service.signin('correo@email.com', '123456'))
                           .rejects.toThrow(NotFoundException)

    })

    it('returns a user if a valid password is provided', async()=>{
        await service.signup('correo@gmail.com', '123456')
        const user = await service.signin('correo@gmail.com', '123456')

        expect(user).toBeDefined()
    })

    it('invalid password returns error', async ()=>{

        await service.signup('correo@gmail.com', '123456')

        await expect(service.signin('correo@gmail.com',
'uhiuh122792')).rejects.toThrow(BadRequestException)
    })
})
```

Unit Testing a Controller

- Vamos a testar UserController
- Testar **decoradores** es un pelin complicado. **No los vamos a testar**, vamos a hacer cómo que no están.
- Vamos a imaginar que no están, y que solo está el controlador con lo que sea que tenga **como argumento y valor de retorno**

- Voy a tener que mockear el AuthService y el UsersService
- Observo en el controlador cuales son los métodos que usa cada handler
- Los defino en el fakeUsersService y el fakeAuthService

```
import { Test, TestingModule } from "@nestjs/testing"
import { UsersController } from "./users.controller"
import { UsersService } from "./users.service"
import { AuthService } from "./auth.service"
import { User } from "./entities/user.entity"

describe('UsersController test', ()=>{
    let controller: UsersController
    let fakeUsersService: Partial<UsersService>
    let fakeAuthService: Partial<AuthService>

    beforeEach(async()=>{
        fakeUsersService={
            findOne: ()=>{},
            find: ()=>{},
            remove: ()=> {},
            update: ()=>{}
        }

        fakeAuthService={
            signup: ()=>{},

            signin: ()=>{}
        }

        const module : TestingModule = await Test.createTestingModule({

            controllers: [UsersController],

        }).compile()

        controller = module.get<UsersController>(UsersController)
    })

})
```

- Si coloco el cursor encima de cada método Typescript me dice qué espera de él
- Dejando el update para después, que incorpora el UpdateUserDto, para satisfacer los tipados del resto quedaría algo así

```
beforeEach(async()=>{
        fakeUsersService={
            findOne: (id: number)=> Promise.resolve({id, email:"correo@gmail.com",
password:'123456'} as User),
            find: (email: string)=> Promise.resolve([{email, password:'123456'} as
User]),
```

```
            remove: (id: number)=> Promise.resolve({id, email:"correo@gmail.com",
password:'123456'} as User),
            //update: (id: number )=>{}
        }

        fakeAuthService={
            signup: (email: string, password: string)=> Promise.resolve({email,
password} as User),

            signin: (email: string, password: string)=> Promise.resolve({email,
password} as User)
        }

        const module : TestingModule = await Test.createTestingModule({

            controllers: [UsersController],

        }).compile()

        controller = module.get<UsersController>(UsersController)
    })
```

- Los handlers findUser y findAll del controller usa los métodos find y findOne del UsersService
- Vamos con ello!
- Si quiero usar fakeUsersService tengo que decírselo al DI Container

```
const module : TestingModule = await Test.createTestingModule({

    controllers: [UsersController],
    providers:[
        {
            provide: UsersService,
            useValue: fakeUsersService
        },
        {
            provide: AuthService,
            useValue: fakeAuthService
        }
    ]

}).compile()
```

NOTA: a veces da error la ruta de importación y es porque en lugar de poner src/ se debe poner ../

- Este test pasa

```
import { Test, TestingModule } from "@nestjs/testing"
import { UsersController } from "./users.controller"
```

```
import { UsersService } from "./users.service"
import { AuthService } from "./auth.service"
import { User } from "./entities/user.entity"

describe('UsersController test', ()=>{
    let controller: UsersController
    let fakeUsersService: Partial<UsersService>
    let fakeAuthService: Partial<AuthService>

    beforeEach(async()=>{
        fakeUsersService={
            findOne: (id: number)=> Promise.resolve({id, email:"correo@gmail.com",
password:'123456'} as User),
            find: (email: string)=> Promise.resolve([{email, password:'123456'} as
User]),
            remove: (id: number)=> Promise.resolve({id, email:"correo@gmail.com",
password:'123456'} as User),
            //update: (id: number )=>{}
        }

        fakeAuthService={
            signup: (email: string, password: string)=> Promise.resolve({email,
password} as User),

            signin: (email: string, password: string)=> Promise.resolve({email,
password} as User)
        }

        const module : TestingModule = await Test.createTestingModule({

            controllers: [UsersController],
            providers:[
                {
                    provide: UsersService,
                    useValue: fakeUsersService
                },
                {
                    provide: AuthService,
                    useValue: fakeAuthService
                }
            ]

        }).compile()

        controller = module.get<UsersController>(UsersController)
    })

    it('UsersController is defined', ()=>{
        expect(controller).toBeDefined()
    })

})
```

Not Super Effective Tests

- En el controller, findAllUsers solo usa el método find de UsersService
- Recuerda que no tenemos la habilidad de testar lo relacionado con los decoradores

```
it('should return all users', async()=>{
    const users = await controller.findAll('email@gmail.com')
        expect(users.length).toEqual(1)
        expect(users[0].email).toEqual('email@gmail.com')
})
```

- Los controladores, obviando los decoradores, tienen una lógica muy simple
- findUser usa findOne del UsersService

```
it('should return one user', async()=>{

    const user = await controller.findUser('1')

    expect(user).toBeDefined()   //NO ENTIENDO PORQUE NO PASA ESTE TEST, devuelve
undefined
})
```

- Este tampoco funciona, devuelve undefined

```
it('throws an error if user given id is not found', async()=>{
    fakeUsersService.findOne = ()=> null
    await expect(controller.findUser('1')).rejects.toThrow(NotFoundException)
})
```

NOTA: El error estaba en el controlador! Tenía solo el await sin el return!!!

```
@Get(':id')
async findUser(@Param('id') id: string) {
return await this.usersService.findOne(+id);
}
```

Testing signin

- Para el signin debemos darle un body de tipo createUserDto y un objeto session (de tipo any)
- Implementa el authService.signin con un email y un password sacados del body
- El id debe asignarse al objeto session

- Devuelve un user

```
@Post('/signin')
async signIn(@Body() body: CreateUserDto, @Session() session: any){
const user = await this.authService.signin(body.email, body.password)
session.userId = user.id
return user
}
```

- No podemos asegurar de que devuelva un usuario
- Y de que la userId sea asignada a la session
- En el fakeAuthService en la hoja de testing me aseguro de harcodear un id

```
fakeAuthService={
        signup: (email: string, password: string)=> Promise.resolve({email,
password} as User),

        signin: (email: string, password: string)=> Promise.resolve({id: 1, email,
password} as User)
    }
```

- Creo un objeto de session vacío en el test
- En el signin le paso el objeto con email y password, lo que sería el body, y el objeto session que he creado vacío
- Para que no me de error de tipado con user.userId en el expect la inicio en el objeto de session

```
it('signin updates session and returns user', async()=>{
    const session = {userId: 10}

    const user = await controller.signIn({email:'correo@gmail.com', password:
'123456'}, session)

    expect(user.id).toEqual(1)
    expect(session.userId).toEqual(1)
})
```

- Es correcto porque en el metodo fake signin le puse id:1

---

05 NEST MANAGE APP

- Vamos a instalar un paquete que nos configure el ConfigService

```
npm i @nestjs/config
```

- Este paquete incluye dotenv
- Podremos usar ConfigService para leer valores guardados en .env
- Vamos a ver como tener diferentes variables para el entorno de producción y el local
- No vamos a seguir las normas que dicta dotenv para la implementación

---

Applying Dotenv for Config

- Creo .env.development y .env.test
- En .env.test escribo:

```
DB_NAME= test.sqlite
```

- En .env.development escribo

```
DB_NAME= development.sqlite
```

- En app.module importo **ConfigModule, ConfigService**
- Configuro el ConfigModule con forRoot
- Con **isGlobal en true** significa que no necesito importar el ConfigModule en otros módulos. Sirve globalmente
- Debo especificar **el path** de los archivos .env que quiero usar
  - Uso un template string para indicarle que quiero usar el que esté utilizando **NODE_ENV**
- Según el entorno que se esté ejecutando buscará .env.development o .env.test
- Para usar el **ConfigService** y usar la variable de entorno para definir la database en el TypeOrmModule **necesito inyectarlo**
- Voy a tener que **refactorizar TypeOrmModule con forRootAsync**
  - Con **inject** inyecto el servicio
  - Con la función **useFactory** puedo retornar el objeto de configuración **usando el servicio**
  - Uso config.get para obtener la variable de entorno de tipo String

```
import { Module } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { ReportsModule } from './reports/reports.module';
import {TypeOrmModule} from "@nestjs/typeorm"
import { User } from './users/entities/user.entity';
import { Report } from './reports/entities/report.entity';
import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({
  imports: [UsersModule, ReportsModule,
    ConfigModule.forRoot({
      isGlobal: true,
      envFilePath: `.env.${process.env.NODE_ENV}`
    }),
    TypeOrmModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (config: ConfigService)=>{
        return {
          type: 'sqlite',
```

```
            database: config.get<string>('DB_NAME'),
            entities: [User, Report],
            synchronize: true
            }
        }
    })],
    controllers: [],
    providers: [],
})
export class AppModule {}
```

- Debo declarar el NODE_ENV para que no reciba .env.undefined
- En el package.json, en start:dev lo seteo
- Hago una instalación necesaria

```
npm install cross-env
```

- Configuro el script:

```
"start:dev": "cross-env NODE_ENV=development nest start --watch"
```

- Puedo hacer lo mismo con start, debug, test, test-watch, test-debug, menos en start:prod
- Excluyo .env.development y .env.test con el .gitignore
- Entonces, con el ConfigService, he configurado una db para development y otra para testing

NOTA: cuando ejecuto npm run test salta un error diciendo SQLITE_BUSY: database is locked

---

Solving SQLite error

- Este error sucede porque Nest usa JEST. Jest va a leer app.e22-spec.ts (test de integración no visto en el curso, que tiene un beforeEach que crea una instancia de la aplicación que quiere acceder a test.sqlite) y los tests unitarios que tambien generan otra instancia de la app que quiere acceder a test.sqlite y SQLite no permite multiples conexiones diferentes, quiere ver solo una conexión .
- Como Jest va a tratar de leer todos los tests al mismo tiempo es un problema
- Vamos a decirle a Jest que solo ejecute un test cada vez, así irá más rápido tambien
- En package.json (para los tests de integración)

```
"test:e2e": "cross-env NODE_ENV=development jest --config ./test/jest-e2e.json --maxWorkers=1"
```

- Borro el archivo de test.sqlite y pongo en marcha de nuevo el servidor con :test y no hay problema

---

It works!

- Debemos borrar el archivo test.sql antes de cada sesión de test para que no de error
- Se crea solo al poner en marcha el servidor
- Se puede crear un beforeEach global en el test, para no tener que borrar la db en cada archivo de test dentro del beforeEach

- En jest-e2e-spec.ts añado

> "setupFilesAfterEnv: ["/setup.ts"]

- Este setup.ts se ejecutará antes de que los tests se ejecuten
- rootDir hace referencia al directorio donde se encuentra el archivo, /test
- Defino el setup.ts
    - Para borrar el archivo test.sqlite cada vez antes de ejecutar los tests uso librerías de Node
        - Uso el __dirname para ubicarme en el directorio, con .. subo un nivel y le digo el archivo a borrar
        - Si el test.sqlite no existe lanzará un error, por ello uso un try y un catch. El error me da igual

```
import { rm } from "fs/promises";
import { join } from "path";


global.beforeEach(async()=>{

    try {
    await rm(join(__dirname, '..', 'test.sqlite'))
    } catch (error) {}

})
```

NOTA: TypeOrm crea el archivo si no existe automáticamente

---

06 NEST RELATIONS IN TYPEORM

- Vamos con reports!
- Un endpoint POST le va a permitir al usuario recibir info del vehículo que ha vendido
- Otro endpoint GET les va a permitir a otros usuarios obtener una valoración del coche
- Un tercer endpoint PATCH que aprueba o rechaza un report hecho por un user

---

Adding Properties to Reports

- Lo que hay que hacer con el módulo reports se asemeja mucho a lo hecho con users
- Vamos con el la entity, a añadir ciertas propiedades que necesito en el report

```
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";


@Entity()
export class Report {
```

```
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    price: number

    @Column()
    make: string //marca

    @Column()
    model: string

    @Column()
    year: number

    @Column()
    lng: number

    @Column()
    lat: number

    @Column()
    mileage: number
}
```

- Ahora que tengo la entidad vayamos con el controlador y el servicio!

A Dto for report creation

- Vamos a centrarnos en tener la habilidad de crear un nuevo reporte

```
@Controller('reports')
export class ReportsController {
  constructor(private readonly reportsService: ReportsService) {}

  @Post()
  create(@Body() body: CreateReportDto) {
    return this.reportsService.create(body);
  }
}
```

- En create-report.dto.ts

```
export class CreateReportDto {
    make: string

    model: string
```

```
    year: number

    lng: number

    lat: number

    mileage: number

    price: number
}
```

- Añado la **validación** con los validators de **class-validator**
- Para el año del coche voy a querer validar que sea un año válido, por lo que usaré Min y Max
- Para la longitud y latitud del punto de venta tengo isLatitude e isLongitude
- Pongo de máximos un millón de kilometraje y un millón de precio

```
import {IsString, IsNumber, Min, Max, isLatitude, IsLongitude, IsLatitude} from
'class-validator'


export class CreateReportDto {

    @IsString()
    make: string

    @IsString()
    model: string

    @IsNumber()
    @Min(1900)
    @Max(2050)
    year: number

    @IsLongitude()
    lng: number

    @IsLatitude()
    lat: number

    @IsNumber()
    @Min(0)
    @Max(1000000)
    mileage: number

    @IsNumber()
    @Min(0)
    @Max(1000000)
    price: number
}
```

- Para asegurarme de que el usuario esté autenticado usaré el AuthGuard que construí por eso lo dejé en /src/guards

```
import { CanActivate, ExecutionContext } from "@nestjs/common";
import { Observable } from "rxjs";

export class AuthGuard implements CanActivate{

    canActivate(ctx: ExecutionContext): boolean | Promise<boolean> |
Observable<boolean> {
     const request = ctx.switchToHttp().getRequest()

     return request.session.userId
    }
}
```

- Importo **UseGuards** de @nestjs/common y el **AuthGuard**
- Lo coloco en el POST. Esto me asegura que la persona esté logeada

```
@Post()
@UseGuards(AuthGuard)
create(@Body() body: CreateReportDto) {
return this.reportsService.create(body);
}
```

- Vamos con el servicio!

Saving a report with the Reports Service

- Primero debo inyectar el repo (la entidad)
- En el ReportsService

```
constructor(@InjectRepository(Report) private  report: Repository<Report>){}
```

- En el método create de ReportsService

```
create(reportDto: CreateReportDto) {
const report = this.repo.create(reportDto)

return this.repo.save(report)
}
```

- Compruebo que todo funciona adecauadamente con ThunderClient o Postman o los archivos .http

- Puedo hacerlo también creando un archivio http dentro de reports
- Sirve para documentar. No tiene porqué estar en reports, puede estar a nivel de aplicación
- Debo tener instalado REST Client
- Recuerda que debo estar logueado para poder postear!!
- requests.http

```
POST http://localhost:3000/reports
content-type: application/json

{
    "price": 1000,
    "make": "Honda",
    "model": "Civic",
    "year": 1993,
    "lng": 0,
    "lat": 0,
    "mileage": 1000

    }
```

Building Associations

- Vamos a **asociar el user con con el report** que crea
- En la tabla reports necesitamos una columna de user_id
- Hay varios tipos de asociación, entre ellas una es **OneToMany** (de uno a muchos) y **ManyToOne** (de muchos a uno)
- Muchos productos pueden ser de un usuario
- Un usuario puede tener muchos productos
- Tambien hay OneToOne (ej:pasaporte-persona), ManyToMany (ej:clases-estudiantes)
- Puedo usar los **decoradores OneToMany y ManyToOne** de typeorm
- En user.entity uso **OneToMany**, el callback devuelve la entidad y otro callback donde le paso la entidad y puedo devolver el user
- Entonces primero le paso la entidad y luego el campo con el que quiero la relación
    - Marca error porque todavía no lo he definido el user con **ManyToOne** en report.entity
- user.entity.ts

```
@OneToMany(()=> Report, (report)=> report.user)
reports: Report[]
```

- En report hago lo mismo pero con **MayToOne** y la entidad User

```
@ManyToOne(()=> User, (user)=> user.reports)
user: User
```

NOTA: al modificar con ManyToOne debo borrar la development.sqlite

---

More on Decorators

- **INFO ADICIONAL**
- La asociación de los reports cuando buscamos un usuario no está automáticamente definida, no me lo devolverá en la respuesta
- Lo mismo pasa cuando buscamos un report. No me va a devolver el user al que pertenece en la respuesta
- Puedo colocar un console.log(User) en la entidad Report y otro con Report en la entidad User
- User devuelve undefined y el otro devuelve class Report
- Por el hecho de tener una relación circular, esto indica que la entity Report se ejecuta primero
- Significa que en el punto de la entity Report, User todavía no se ha ejecutado
    - Entonces no puedo hacer referencias directamente a User dentro Report
    - Por eso la función que devuelve la entidad, **para solventar este problema**
- En el segundo callback, coje la entidad y hace referencia al campo especificado

---

Associations with Nest and TypeORM

- En POST /reports recibo la cookie y el body que debe validar CreateReportDto
- Para extraer el usuario vamos a usar el decorador **@CurrentUser**, con lo que recibiremos una instancia de User
- Con todo ello se crea una nueva instancia de Report y se salva con el metodo save del repo Reports
- En reports.controller

```
@Post()
@UseGuards(AuthGuard)
create(@Body() body: CreateReportDto, @CurrentUser() user: User) {
  return this.reportsService.create(body, user);
}
```

- En el reports.service le asigno el user a la instancia que he creado con el body

```
create(reportDto: CreateReportDto, user: User) {
  const report = this.repo.create(reportDto)
  report.user = user
  return this.repo.save(report)
}
```

- Ahora cuando creo un report me devuelve el usuario que lo ha creado, pero también me devuelve el password en la respuesta
- Vamos a arreglarlo!

Formatting de Report Response

- Vamos a aplicar el **Serializer**(interceptor)
- Quiero evitar enviar el password en la respuesta
- Para esto usaré el serialize.interceptor creado anteriormente
- Necesito crear un Dto que represente cómo quiero que luzca la respuesta
- Lo que quiero es que solo aparezca una propiedad llamada userId con el id del usuario, no un objeto con toda la info del user
- Entonces, voy a **añadir la propiedad userId** con el id del user en la respuesta **en lugar del objeto user** entero

---

Transformig properties with a Dto

- Creo en /reports/dto/reports.dto.ts
- Importo Export y Transform de class-transformer
- También la entity User

```
import { Expose, Transform} from 'class-transformer'
import { User } from 'src/users/entities/user.entity'


export class ReportDto{
    @Expose()
    id: number

    @Expose()
    price: number

    @Expose()
    year: number

    @Expose()
    lng: number

    @Expose()
    lat: number

    @Expose()
    make: string

    @Expose()
    model: string

    @Expose()
    mileage: number
}
```

- Lo importo en el controlador y lo uso con **Serialize** (el interceptor que creé)
- Le digo que quiero serializar la respuesta acorde al ReportDto

```
@Post()
@UseGuards(AuthGuard)
@Serialize(ReportDto)
create(@Body() body: CreateReportDto, @CurrentUser() user: User) {
  return this.reportsService.create(body, user);
}
```

- Para cambiar la respuesta y añadir una nueva propiedad uso **@Transform**
- Desestructuro **obj**, que **es una referencia a la entidad Report original**
- Uso obj para acceder a user.id
- reports.dto

```
@Transform(( {obj} )=> obj.user.id)
  @Expose()
  userId: number
```

07 NEST BASIC PERMISSION SYSTEM

- Vamos a implementar la idea de aprovar o rechazar un report posteado por un user
- Será un PATCH /reports/:id con un body donde esté el aprovado o no
- Por defecto el report estará no-aprobado
- Solo admin podrá aprobar o rechazar los reports
- Agrego la propiedad a la entity
- report.entity

```
@Column({default: false})
    approved: boolean
```

- Añado a ReportsDto la propiedad approved

```
@Expose()
approved: boolean
```

- En el controller necesito extraer el id y cambiar a true el approved que hay en el body

```
@Patch('/:id')
approveReport(@Param('id') id: string, @Body() body: ApprovedReportDto) {
return this.reportsService.changeApproval(+id, body.approved);
}
```

- Creo el Dto

```
import { IsBoolean } from "class-validator";

export class ApprovedReportDto{

    @IsBoolean()
    approved: boolean
}
```

- En el servicio, para encontrar el report uso async await
- Compruebo que el repo existe

```
  async changeApproval(id: number, approved: boolean) {

    const report = await this.repo.findOneBy({id})

    if(!report) throw new NotFoundException('Report not found')

    report.approved = approved
    return this.repo.save(report)
  }
```

- Puedo comprobar que funciona con ThunderClient con PATCH /reports/id_del_report y pasarle approved: true en el body
- Falta implementar de que solo el admin pueda aprobar estos reports

---

Authorization vs Authentication

- Authentication == saber quien está haciendo la request
- Authorization == saber si la persona que está haciendo la request está autorizada para ello
- Con CurrentUserInterceptor, con su método intercept extraemos el userId de la cookie y sabemos qué user y lo guardamos en la request con la variable request.currentUser
- Entonces, dispongo del user en la request
- current-user-interceptor.ts

```
import { NestInterceptor, ExecutionContext, CallHandler, Injectable } from
"@nestjs/common";
import { UsersService } from "../users.service";

@Injectable()
export class CurrentUserInterceptor implements NestInterceptor{

    constructor(private usersService: UsersService){}

    async intercept(ctx: ExecutionContext, handler: CallHandler){
```

```
        const request = ctx.switchToHttp().getRequest()

        const {userId} = request.session || {}

        if(userId){
            const user= await this.usersService.findOne(userId)

            request.currentUser = user
        }

        return handler.handle()
    }
}
```

- Vamos a incorporar un **AdminGuard** donde vamos a preguntar si request.currentUser es admin
- Si es admin retornará true lo que dará acceso a la ruta
- En lugar de roles podemos manejar admin **con un boolean**

Adding an Authorization Guard

- En user.entity le añado la propiedad admin como boolean

```
@Column({default: false})
admin: boolean
```

- Le pongo el default en **true** para propósitos de **testing**
- Creo en /src/guards/admin.guard.ts
- Será muy similar a auth.guard
- Importo **CanActivate** y **ExecutionContext** de '@nestjs/common'
- CanActivate implica emplear **el método canActivate** y ExecutionContext **contiene la request**
- Recuerda que CurrentUserInterceptor guarda el user en la request, dentro de currentUser
- Valido si está definido el user
- Retornando request.currentUser.admin si existe dará true, si no false
    - **Los Guards trabajan con valores truthly o falsy**

```
import {CanActivate, ExecutionContext} from '@nestjs/common'

export class AdminGuard implements CanActivate{

    canActivate(context: ExecutionContext){
        const request = context.switchToHttp().getRequest()

        if(!request.currentUser) return false

        return request.currentUser.admin
```

```
    }
}
```

Algo no funciona

- En reports.controller importo **AdminGuard** y **UseGuards**

```
@Patch('/:id')
@UseGuards(AdminGuard)
approveReport(@Param('id') id: string, @Body() body: ApprovedReportDto) {
return this.reportsService.changeApproval(+id, body.approved);
}
```

- Para probar creo un nuevo usuario que será admin
- Pero cuando uso el endpoint PATCH para aprobar un report recibo un error 403 "Forbidden resource"
- **ERROR!**

Middlewares, Guards, and Interceptors

- Request --> Middlewares ---> Guards --> Interceptor ---> RequestHandler ---> Interceptor ---> Response
- En el middleware tengo el cookie-session midleware que coloca el user en el objeto session
- El problema está en que **AdminGuard se ejecuta antes que el CurrentUserInterceptor** que me dice que usuario es
- Se soluciona **transformando el CurrentUserInterceptor en un middleware**
- Cogeremos el current-user.interceptor y lo transformaremos en un middleware global como hicimos con cookieSession
- Creo /src/users/middlewares/cuurent-user.middleware.ts
- Importo
    - **Injectable, NestMiddleware** de @nestjs/common
    - **Request, Response, NextFunction** de express
    - **UsersService**
- Al **implementar NestMidlleware** en la clase **debemos usar** el método **use** que **puede ser async**
- Necesito tener acceso al UsersService, por lo que **lo inyecto usando el decorador @Injectable en la clase**

```
import {Injectable, NestMiddleware} from '@nestjs/common'
import { Request, Response, NextFunction } from 'express'
import { UsersService } from '../users.service'

@Injectable()
export class CurrentUserMiddleware implements NestMiddleware{
    constructor(private usersService: UsersService){}
```

```
    async use(req: any, res: any, next: NextFunction) {
        const { userId } = req.session || {}

        if(userId){
            const user = await this.usersService.findOne(userId)
            req.currentUser = user
        }
        next()
    }
}
```

- Para **configurarlo globalmente**, llamo a la función **configure** en AppModule
- Recuerda que **cookieSession solo se puede importar con require**

```
import { Module, MiddlewareConsumer } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { ReportsModule } from './reports/reports.module';
import {TypeOrmModule} from "@nestjs/typeorm"
import { User } from './users/entities/user.entity';
import { Report } from './reports/entities/report.entity';
import { ConfigModule, ConfigService } from '@nestjs/config';
const cookieSession = require('cookie-session');

@Module({
  imports: [UsersModule, ReportsModule,
    ConfigModule.forRoot({
      isGlobal: true,
      envFilePath: `.env.${process.env.NODE_ENV}`
    }),
    TypeOrmModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (config: ConfigService)=>{
        return {
          type: 'sqlite',
          database: config.get<string>('DB_NAME'),
          entities: [User, Report],
          synchronize: true
          }
      }
  })],
  controllers: [],
  providers: [],
})
export class AppModule {
  configure(consumer: MiddlewareConsumer){
    consumer
      .apply(
        cookieSession({
          keys:['lelele']
        }),
      )
```

```
      .forRoutes('*')
  }
}
```

- Hay que hacer lo mismo en UsersModule
- Importo **CurrentUserMiddleware** y también **MiddlewareConsumer** de @nestjs/common
- uso el configure igual, para todas las rutas
- Puedo borrar el objeto de interceptor

```
import { Module, MiddlewareConsumer } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { AuthService } from './auth.service';
//import { CurrentUserInterceptor } from './interceptors/current-
user.interceptor';
//import {APP_INTERCEPTOR} from '@nestjs/core'
import { CurrentUserMiddleware } from './middlewares/current-user.middleware';

@Module({
  imports:[TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [
    UsersService,
    AuthService,
    //{ provide: APP_INTERCEPTOR,
     // useClass: CurrentUserInterceptor}
    ]
})
export class UsersModule {
  configure(consumer: MiddlewareConsumer){
    consumer
      .apply(CurrentUserMiddleware).forRoutes('*')
  }
}
```

Nota: si req.currentUser diera error de tipado puedo decirle a express que la request puede tener currentUser de tipo User

```
declare global{
  namespace Express{
    interface Request{
      currentUser?: User
    }
  }
}
```

```
//lo añado fuera  de la clase CurrentUserMiddleware
```

Validating Query String Values

- GET /reports tiene en su query strings make, model, year, mileage, longitude, latitude y devuelve el valor estimado del coche
- Necesitamos validar esta información con un dto como hacemos con el body
- Creo en /reports/dto/get-estimated.dto.ts
- No incluyo el precio en el dto porque eso es lo que estoy buscando

```
import { IsLatitude, IsLongitude, IsNumber, IsString, Max, Min } from "class-validator"

export class GetEstimateDto{

    @IsString()
    make: string

    @IsString()
    model: string


    @IsLatitude()
    @Min(0)
    lat: number

    @IsLongitude()
    @Min(0)
    lng: number

    @IsNumber()
    @Min(0)
    @Max(1000000)
    mileage: number

    @IsNumber()
    @Min(1939)
    @Max(2050)
    year:number
}
```

- Voy al reports.controller, necesito el decorador Query

```
@Get()
getEstimate(@Query() query: GetEstimateDto) {
  return this.reportsService.getEstimate(query);
}
```

- El problema es que cuando voy al endpoint con las query, los query-parameters siempre son strings y el dto espera números

> http://localhost:3000/reports?make=toyota&model=corolla&lat=0&lng=0&mileage=1000&year=1991

Transforming Query String in Data

- En el get-estimated.dto importo **Transform** de class-transformer
- Lo coloco encima de los otros decoradores
- Desestructuro la propiedad value de la propiedad en si en el callback
- Para longitude y latitude en lugar de ParseInt usaré ParseFloat

```
@Transform(({value})=>parseInt(value))
```

```
import { IsLatitude, IsLongitude, IsNumber, IsString, Max, Min } from "class-
validator"
import { Transform } from "class-transformer"

export class GetEstimateDto{

    @IsString()
    make: string

    @IsString()
    model: string


    @Transform(({value})=>parseFloat(value))
    @IsLatitude()
    @Min(0)
    lat: number

    @Transform(({value})=>parseFloat(value))
    @IsLongitude()
    @Min(0)
    lng: number

    @Transform(({value})=>parseInt(value))
    @IsNumber()
    @Min(0)
    @Max(1000000)
    mileage: number

    @Transform(({value})=>parseInt(value))
    @IsNumber()
    @Min(1939)
    @Max(2050)
```

```
    year:number
}
```

- Ahora no obtengo errores de validación

How will we generate the estimate

- Para generar el valor estimado, se buscarán los mismos modelos y marcas
- Se buscará ne la misma latitude y langitude con una variabilidad de 5 grados
- Según el año se buscará en un rango de 3 años
- Se ordenará según el que menos kilometraje tenga y se mostrarán los 3 resultados más cercanos
- No es una SUPER LÓGICA, pero servirá para ver los **Query Builders**

Creating a Query Builder

- Podemos usar find y findOne como filtro sobre los reports
- Hay varios pasos descritos anteriormente que quiero hacer con la query
- Para eso usaré createQueryBuiler en el servicio
- reports.controller

```
@Get()
getEstimate(@Query() query: GetEstimateDto) {
  return this.reportsService.createEstimate(query);
}
```

- Vamos con el createQueryBuilder
- Con select asterisco elijo todos los registros
- En el where hago el filtrado, le digo que el valor make será igual a un valor make que yo le voy a indicar y se lo paso en un objeto
- Con getRawMany obtengo el resultado
- **NOTA**: Podría desestructurar las propiedades con createEstimate({make, model,lng,lat}) pero lo hago así para que quede más claro
- reports.service

```
  createEstimate(estimateDto: GetEstimateDto) {
    return this.repo.createQueryBuilder()
    .select('*') //selecciono todo
    .where('make = :make', {make: estimateDto.make} )  //filtro el resultado
    .getRawMany() //obtengo el resultado
  }
```

- Si volviera a escribir otro .where sobreescribiría el anterior
- Para encadenar where uso andWhere

- Para hacer el rango de 5º de lng y lat uso BETWEEN
- Para ordenar por kilometraje le resto el mileage dado a la propiedad mileage
    - Si le pudeo indicar el orden en valores descendientes con DESC. Uso ABS para obtener el valor absoluto y no me de un posible valor negativo
    - OrderBy no me permite pasarle en un objeto mileage, uso setParameters
    - Para obtener solo 3 resultados uso limit(3)
    - En el select, en lugar de buscar con * uso AVG(de average, promedio, del precio de los 3 resultados) y le paso el precio como string
    - En lugar de getRawMany uso getRawOne

```
createEstimate({make, model, lat, lng, year, mileage}) {
  return this.repo.createQueryBuilder()
                  .select('AVG(price)', 'price')
                  .where('make = :make', {make} )
                  .andWhere('model = :model', {model})
                  .andWhere('lat - :lat BETWEEN -5 AND 5', {lat})
                  .andWhere('lng - :lng BETWEEN -5 AND 5', {lng})
                  .andWhere('year - :year BETWEEN -3 AND 3', {year})
                  .orderBy('ABS(mileage -:mileage)', 'DESC' )
                  .setParameters({mileage})
                  .limit(3)
                  .getRawOne()
}
```

- Para probar esta logica debo crear algunos reports con la misma marca y modelo
- Si creo 3 modelos SET Panda enntre los años 1990-1995 y apunto a este endpoint me da un precio promedio

> http://localhost:3000/reports?make=Seat&model=Panda&lat=0&lng=0&mileage=1000&year=1991

- Ahora falta añadir que los reports estén aprovados para poder contar con ellos
- Uso andWhere en el queryBuilder

> .andWhere('approved IS TRUE')

- Si no hay resultados devuelve null

---

08 NEST PRODUCTION

- Para producción vamos a setear variables de entorno ya cambiar a Postgres
- Empezaremos por guardar el string que codifica la cookie en una variable de entorno
- Añado COOKIE_KEY a los archivos .env
- ConfigModule me da acceso a ConfigService
- Para usar ConfigService dentro de AppModule voy a usar inyección de dependencias
- Coloco la variable de entorno en app.module

```
import { Module, MiddlewareConsumer } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { ReportsModule } from './reports/reports.module';
import {TypeOrmModule} from "@nestjs/typeorm"
import { User } from './users/entities/user.entity';
import { Report } from './reports/entities/report.entity';
import { ConfigModule, ConfigService } from '@nestjs/config';
const cookieSession = require('cookie-session');

@Module({
  imports: [UsersModule, ReportsModule,
    ConfigModule.forRoot({
      isGlobal: true,
      envFilePath: `.env.${process.env.NODE_ENV}`
    }),
    TypeOrmModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (config: ConfigService)=>{
        return {
          type: 'sqlite',
          database: config.get<string>('DB_NAME'),
          entities: [User, Report],
          synchronize: true
          }
      }
  })],
  controllers: [],
  providers: [],
})
export class AppModule {
  constructor(private configService: ConfigService){}

  configure(consumer: MiddlewareConsumer){
    consumer
      .apply(
        cookieSession({
          keys:[this.configService.get('COOKIE_KEY')]
        }),
      )
      .forRoutes('*')
  }
}
```

Understanding the Synchronize flag

- Vamos a usar sqlite en development y testing, y Postgres en producción
- Una de las propiedades dentro del TypeOrmModule es la de syncronize: true **IMPORTANTE!!**
- Lo que hace synchronize es que si elimino alguna propiedad de la entity User, por ejemplo, la borrará también de la DB automáticamente

- Lo mismo si añado una propiedad a la entity
- Pero **no todo son cosas buenas del synchronize**
- Si **por error borro alguna propiedad, pierdo la data** de manera irrecuperable (a no ser que tenga un backup)
- Por eso, en development es recomendable tenerla en true, pero **no en producción**
- Cambiar a synchronize: false y no volverlo a usar en true

The Theory behind Migrations

- **Migration** es un archivo con dos funciones dentro
    - up --> describe cómo actualizar la estructura de la DB
        - Añade la tabla users
        - Da a la tabla la columna mail
        - Da a la tabla la columna password
    - down --> describe cómo deshacer los pasos de up
        - Elimina la tabla users
- Podemos crear varios archivos Migration, otro para reports, por ejemplo, que cree la tabla reports con make, model,etc
- Es un pelín desafiente aplicar esta lógica

Headaches with Config Management

- Nest y TypeORM funcionan realmente bien, pero con las migraciones no es lo mejor del mundo
- Con el servidor parado
    - Uso el CLI de TypeORM para generar un archivo migration vacío
    - Añado código al migration file
    - Uso el CLI de TypeORM para aplicar la migración a la DB
        - TypeORM CLI ejecutará solo los entity files + migration file, entonces conectará con la DB y hará los cambios, pero no tiene ni idea de lo que es NEST, ni ConfigModule, ni ConfigService, ni siquiera lo que hay dentro de AppModule
        - Pero AppModule es lo que define como me conecto a la DB!
        - Entonces, el CLI no tiene ni idea de cómo obtener el objeto {type: 'sqlite', database: 'db.sqlite', entities: [User, report]} necesario para la conexión
        - **Debemos configurar la conexión en un lugar para que lo pueda usar tanto NEST como el CLI de TypeORM**
        - Este es el punto

TypeORM and Nest Config is Great

- Tenemos que decirle al CLI cómo conectarse a la DB
- Configurar NEST y TYpeORM puede ser un poco **NIGHTMARE**, estás avisado
- Acudir a la documentación:
    - Como crear una conexión
    - Crear un ormconfig.json para pasarle la info. TypeORM cargará automáticamente este archivo
        - Puede ser ormconfig.js o .ts .yml o .xml también

    - En el json no tenemos habilidades de scripting, tampoco en .yml o .xml por lo que solo me queda .ts y .js
    - TypeORM espera variables con nombres específicos (mirar docu)
    - El sitio dónde sea que vayamos a publicar la app, va a darnos unas variables de conexión automáticamente que no tienen porqué coincidir en el nombre
    - Así que usar variables o variables de entorno para decirle al CLI lo que debe hacer no es una opción
- Esto nos deja solo ormconfig.js y ormconfig.ts
- **AQUI ES DONDE LAS COSAS SE PONEN FEAS**
- En app.module comento el TypeOrmModule.forRootAsync y lo reemplazo por un forRoot sin ninguna configuración

```
TypeOrmModule.forRoot()
```

- Esto me devuelve un error en consola porque no tiene ningún parametro de configuración
- En la raíz del proyecto creo ormconfig.ts

NOTA: el uso de ormconfig está deprecado

```
export ={
    type: 'sqlite',
    database: 'db.sqlite',
    entities: ['**/*.entity.ts'], //le paso el path de las entities
    synchronize: false
}
```

- Con esta configuración no es suficiente. Salta un error que dice unexpected token 'export'
- El orden de ejecución de NEST es primero todo lo que hay en src, lo pasa a JavaScript y lo coloca en dist
- Entonces Node corre el main.js
- TypeORM intenta correr ormconfig.ts como JavaScript pero se encuentra typescript, por eso falla
- Entonces, **no puedo usar .ts**
- **Solo me queda una opción: ormconfig.js**
- Cambio a .js. Al ser js debo usar js plano para el export

```
module.exports ={
    type: 'sqlite',
    database: 'db.sqlite',
    entities: ['**/*.entity.ts'],
    synchronize: false
}
```

- Pero la cosa no acaba aquí. Ahora me sale el error de "Can not use imports statement outside a module" y apunta a report.entity

- En entities tengo archivos typescript y estos no se ejecutan hasta que son transformados a JavaScript
- Pasa lo mismo que antes, TypeORM intenta cargarlos antes de ser JavaScript
- Entonces, debo decirle que vaya a **/dist** para tener la versión de las entities en JavaScript
- **Debo cambiar el .ts de las entities por .js**

```
module.exports ={
    type: 'sqlite',
    database: 'db.sqlite',
    entities: ['**/*.entity.js'],
    synchronize: false
}
```

- Para que funcione en development y test
- Ahora si uso npm run test:e2e me salta error porque usa ts-jest que lee archivos typescript y está recibiendo las entities en js
- Vamos a decirle a ts-jest que no importa si encuentra un archivo js, lo ejecute igual
- **En ts.config le añado "allowJs": true**
- No es suficiente. No encuentra User
- El test runner no es capaz de leer los .js que estan en dist como le indico en ormconfig

```
module.exports= {
    type: 'sqlite',
    database: 'development.sqlite',
    entities: process.env.NODE_ENV === "development"
    ? ['**/*.entity.js']
    : ['**/*.entity.ts'],
    synchronize: false
}
```

Env-Specific database Config

- Vamos a extraer algo de configuración del app.module y ponerlo dentro de ormconfig.js

```
var dbConfig = {
  synchronize: false  //puedo colocar aqui el synchronize y no en cada caso del
switch
}

switch (process.env.NODE_ENV){
  case 'development':
    Object.assign(dbConfig,{
      type: 'sqlite',
      database: 'development.sqlite',
      entities: ['**/*.entity.js'],
      //synchronize: false
    });
```

```
      break;
    case 'test':
      Object.assign(dbConfig,{
        type: 'sqlite',
        database: 'test.sqlite',
        entities: ['**/*.entity.ts'],
        //synchronize: false
      });
      break;
    case 'production':
      break;
      default:
        throw new Error('Unknown environment')
}

module.exports = dbConfig
```

Installing the TypeORM CLI

- Para instalar y poner a punto el CLI es mejor acudir a la docu
- TypeORM y el CLI no saben como interpretar archivos typescript, hay que configurarlo
- Añado un nuevo script a package.json

> "typeorm": "node --require tsnode/register ./node_modules/typeorm/cli.js

- Vamos a usar este comando para generar un migration file
- Hay que añadir un par de opciones al objeto dbConfig al ormconfig.js [deprecated]

```
var dbConfig = {
  synchronize: false,
  migrations: [`migrations/*.js`],
  cli:{
    migrationsDir: 'migrations'
  }
}
```

- Se creará un archivo .ts pero CLI solo trabaja con .js, por lo que algo habrá que transpilar la migración antes de ejecutarla
- -o genera un archivo .js

> npm run typeorm migration:generate -- -n initial-schema -o

- Ahora en la carpeta migrations hay un archivo .js
- El CLI consulta las entidades de la app y hace los queryRunners automáticamente

NOTA: falta código, es para hacerse una idea. El archivo lo genera automáticamente

```
const {MigrationInterface, QueryRunner} = require("typeorm")

module.exports = class initialSchema19736173623 {
  name = 'initialSchema19736173623'

  async up(queryRunner){
    await queryRunner.query(`CREATE TABLE "user" ("id" integer PRIMARY KEY
AUTOINCREMENT NOT NULL, "email" varchar,)`)
    await queryRunner.query(`CREATE TABLE "report" ("id" integer PRIMARY KEY
AUTOINCREMENT NOT NULL, "approved",)`)
    await queryRunner.query(`CREATE TABLE "temporary_report" ("id" integer PRIMARY
KEY AUTOINCREMENT NOT NULL, )`)
    await queryRunner.query(`INSERT INTO "temporary_report" ("id" "approved",
"price", "make", "model", "year")`)
    await queryRunner.query(`DROP TABLE "report"`)
    await queryRunner.query(`ALTER TABLE "temporary_report" RENAME TO "report"`)
  }

  async down(queryRunner){
    await queryRunner.query(`ALTER TABLE "temporary_report" RENAME TO "report"`)
    (..etc...)
  }
}
```

NOTA: el archivo generado automáticamente no funciona. Es reemplazado por este

```
const { MigrationInterface, QueryRunner, Table } = require('typeorm');

module.exports = class initialSchema1625847615203 {
  name = 'initialSchema1625847615203';

  async up(queryRunner) {
    await queryRunner.createTable(
      new Table({
        name: 'user',
        columns: [
          {
            name: 'id',
            type: 'integer',
            isPrimary: true,
            isGenerated: true,
            generationStrategy: 'increment',
          },
          {
            name: 'email',
            type: 'varchar',
          },
          {
            name: 'password',
```

```
            type: 'varchar',
          },
          {
            name: 'admin',
            type: 'boolean',
            default: 'true',
          },
        ],
      }),
    );

    await queryRunner.createTable(
      new Table({
        name: 'report',
        columns: [
          {
            name: 'id',
            type: 'integer',
            isPrimary: true,
            isGenerated: true,
            generationStrategy: 'increment',
          },
          { name: 'approved', type: 'boolean', default: 'false' },
          { name: 'price', type: 'float' },
          { name: 'make', type: 'varchar' },
          { name: 'model', type: 'varchar' },
          { name: 'year', type: 'integer' },
          { name: 'lng', type: 'float' },
          { name: 'lat', type: 'float' },
          { name: 'mileage', type: 'integer' },
          { name: 'userId', type: 'integer' },
        ],
      }),
    );
  }

  async down(queryRunner) {
    await queryRunner.query(`DROP TABLE ""report""`);
    await queryRunner.query(`DROP TABLE ""user""`);
  }
};
```

- Para ejecutar

```
typeorm migration:run
```

- Tiene que salir en consola executed successfully
- Si diera error de db asegurate de borrar la development.sqlite

---

Running Migration with e2e

- Da error 'no such table user'
- Para asegurarme que la migración se ejecute también con los tests e2e necesito añadir código al ormconfig [deprecated]
- Cada test que ejecuto automáticamente borra la db
- Entonces cada test necesita re-migrar la db
- Debe estar habilitado en tsconfig allowJs en true
- En ormconfig añado migrationsRun al switch, en el case de test

```
case 'test':
    Object.assign(dbConfig,{
      type: 'sqlite',
      database: 'test.sqlite',
      entities: ['**/*.entity.ts'],
      //synchronize: false
      migrationsRun: true
    });
```

Production DB Config

- Para usar Heroku subo la app a gitHub
- Para conectar con la db se usará process.env.DATABASE_URL proveida por Heroku
- Necesitamos crear la DB de Postgres
- En el ormconfig, en el case de production del switch

```
case 'production':
    Object.assign(dbCOnfig,{
      type: 'postgres',
      url: process.env.DATABASE_URL,
      migrationsRun: true, //para asegurarnos de ejecutar las migraciones
      entities: ['**/*.entity.js'], //uso .js
      ssl: {     //propiedad especifica de Heroku
        rejectUnathorized: false
      }
    })
    break;
```

- Me aseguro de instalar el driver de postgres

```
npm i pg
```

- Instalar Heroku CLI según la documentación

Heroku Specific Project Config

- Compruebo que estoy en heroku

> heroku auth:whoami

- Hay algo que cambiar en el main.ts
- Heroku da su propio puerto

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';
const cookieSession = require('cookie-session');

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

 // app.use(cookieSession({
 //   keys:['lalala']
 // }))

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true
    })
  )
  await app.listen(process.env.PORT || 3000);
}
bootstrap();
```

- En el directorio raíz creo Procfile (sin extensión)
- El comando para iniciar en producción en NEST es start:prod
- Antes hay que hacer el build pero Heroku lo hace automáticamente
- Procfile

```
web:  npm run start:prod
```

- En el tsconfig.build.json debo excluir el ormconfig y migrations
- Para segurarme que typescript no intenta compilar estos archivos .js

```
{
  "extends": "./tsconfig.json",
  "exclude"; ["ormconfig.js", "migrations","node_modules", "test","dist",
"**...more_code"]
}
```

Deploying the app

> git add . git commit - m "production commit" heroku create

- Creo la db de postgres

> heroku addons:create heroku-postgresql:name_db

- Seteo la cookie_key

> heroku config:set COOKIE_KEY=uoehyrbeiury879605 heroku config:set NODE_ENV=production git push heroku master