NODE TypeScript Herrera

02 Sección bases NODE_TS

Configuración logger Winston y Typescript en Node

Node Logger - Winston

- Tengo nodemon instalado globalemnte
- Hago el npm init -y para tener el package.json
- Instalo el paquete y copio la configuración de Winston en un archivo, en la carpeta plugins/logger.plugin.js
- A través de una Factory Function que llamo buildLogger implemento el patrón adaptador para el logger

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
 // defaultMeta: { service: 'user-service' },
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});

module.exports = function buildLogger(service){

    return{
        log: (message)=>{
            logger.log('info', {message, service})
        },
        error: (message)=>{
            logger.error('error', {message, service})
        }
    }
}
```

- En app.js

```
const buildLogger = require('./plugins/logger.plugin')

const logger = buildLogger('app.js')
```

```
logger.log("hola mundo")
logger.error("Esto es un error")
```

- Esto me crea automáticamente el combined.log y el error.log
- En el combined.log puedo ver esto

```

```

- En el error.log esto

```
{"level":"error","message":"info Esto es un error","service":"app.js"}
```

- Puedo añadir este código para ver el mensaje en consola. En producción no conviene

```
logger.add(new winston.transports.Console({
    format: winston.format.simple()
}))
```

- Más adelante cuando configuremos las variables de entorno habrá una configuración adicional
- Si puede impactar una DB con Winston, pero también se pueden usar estos archivos .log
- Me interesa la fecha y la hora del evento
- Bien podría crearlo en el objeto

```
error: (message)=>{
    logger.error('error', {
        message,
        service,
        at: new Date().toISOString()
    })
}
```

- Pero Winston tiene su manera de hacerlo
- Para combinar el formato json y que me agregue el timestamp. Desestructuro json, combine y timestamp de winston.format
- En lugar de usar winston.format.json() uso el combine para combinar varios formatos

```
const winston = require('winston');
const {combine, timestamp, json} = winston.format

const logger = winston.createLogger({
  level: 'info',
  format: combine(
```

```
    timestamp(),
    json()
  ),
  defaultMeta: { service: 'user-service' },
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});

{...code}
```

- Los archivos .log no se suben a git
- En .gitignore

```
node_modules/

*.log
```

Typescript (proyecto básico)

> npm init -y

- Creo src/app.js
- Código fácil

```
const heroes=[
    {
        id: 1,
        name: "Spiderman",
        owner: "Marvel"
    },
    {
        id: 2,
        name: "Batman",
        owner: "DC"
    },
    {
        id: 3,
        name: "Superman",
        owner: "DC"
    }
]

const findHeroById= (id)=>{
    return heroes.find(heroe=> heroe.id == id)
}
```

```
const hero = findHeroById(2)

console.log(hero)
```

- Hasta aqui vscode puede inferir en que el tipo puede ser undefined si no encuentra el heroe
- Pero si coloco un console.log de hero.name salta un error porque no puede leer las propiedades de undefined
- Puedo usar **null operator**

```
console.log(hero?.name ?? "Hero not found")
```

- Integremos este código con Typescript

Configuración Typescript en Node

> npm i -D typescript @types/node

- Especifico que la trasnpilación vaya a la carpeta dist/ y que esté pendiente de src
- Se puede configurar directamente en el tsconfig manualmente

> npx tsc --init --outDir dist/ --rootDir src npm i -D ts-node

- Cambio app.js por app.ts (desaparece el error de tsconfig porque el transpilador no encontraba ninún archivo ts)
- Le indico que el id es un número

```
const findHeroById= (id: number)=>{
    return heroes.find(heroe=> heroe.id == id)
}
```

- Puedo usar npx tsc --watch (crea la carpeta dist/)
- Para ejecutar nodemon con **npx nodemon dist/app**
- Creemos los scripts mejor
- Crear archivo de configuración nodemon.json en la raíz

```
{
    "watch": ["src"],
    "ext": ".ts, .js",
    "ignore": [],
    "exec": "npx ts-node ./src/app.ts"
}
```

- En el package.json

```
{
    "dev": "nodemon",

}
```

- Para producción instalar con npm i -D rimraf
- Añadir estos script

```
{
    "build": "rimraf ./dist && tsc",
    "start": "npm run build && node dist/app.js
}
```

- Creo una carpeta src/data/heroes.ts
- Traslado el arreglo de heroes a heroes.ts, lo exporto, creo una interfaz y tipo el arreglo

```
interface Heroe{
    id: number,
    name: string,
    owner: string
}

export const heroes: Heroe[]=[
    {
        id: 1,
        name: "Spiderman",
        owner: "Marvel"
    },
    {
        id: 2,
        name: "Batman",
        owner: "DC"
    },
    {
        id: 3,
        name: "Superman",
        owner: "DC"
    }
]
```

- Creo un nuevo directorio /src/services/hero.service.ts

```
import {heroes} from '../data/heroes'

export const findHeroById= (id: number)=>{
```

```
    return heroes.find(heroe=> heroe.id == id)
}
```

- Importo en app.ts en findHeroById

---

03 Sección testing NODE_TS HERRERA

- Esta sección trata de hacer testing con las funciones y adaptadores de la sección 1

Introducción

- El testing no es una pérdida de tiempo
- Las pruebas unitarias estan enfocadas en pequeñas funcionalidades
- Las pruebas de integración están enfocadas en cómo trabajan varias piezas en conjunto
- Tienen que ser fáciles de escribir, leer, confiables, rápidas
- Principalmente se harán unitarias
- Sigue las tres A's
    - A de Arrange (arreglar)
        - Inicializamos variables, realizamos las importaciones necesarias...
    - A de Act (actuar)
        - Aplicamos acciones, estimulos: llamar métodos, simular clicks, realizar acciones sobre el paso anterior...
    - A de Assert (afirmar)
        - Observar el comportamiento resultante

Configuración testing

> npm i -D jest @types/jest ts-jest supertest

- Crear archivo de configuración de Jest

> npx jest --init

- Le digo que si al coverage, y v8. Clear mocks (normalmente si pero para aprender no)

- En jest.config.js

```
preset: 'ts-jest',
testEnvironment: "jest-environment-node",

//opcional
setupFiles: ['dotenv/config']
```

- Scripts:

```
"test": "jest",
"test:watch":"jest --watch",
"test:coverage": "jest --coverage"
```

- Pequeña prueba: creo la carpeta src/test/app.test.ts
- app.test.ts

```
describe('App', ()=>{
    it('should be true', ()=>{
        expect(true).toBe(true)
    })
})
```

- Ejecutar con npm run test
- Puede ser que en el futuro de un error en el que diga que el jest.config.ts está fuera del rootDir (o similar)
- Añade esto antes del compilerOptions, dentro del objeto JSON del tscongif

```
{
  "include": ["src/**/*"],
  "exclude":  ["node_modules", "**/*/spec.ts", "**/*/test.ts"],
  "compilerOptions": {
 }}
```

Arrange, Act y Assert

- **AAA**
  - **Arrange**: preparación de lo que se va a probar
  - **Act**: procedimiento en el que aplico algún tipo de estímulo, estoy probando algo. La acción
  - **Assert**: la evaluación, que es lo que voy a confirmar

```
test("Should be 30", ()=>{

    //Arrange
    const num1 = 10
    const num2 = 20

    //Act
    const result = num1+num2

    //Assert
    expect(result).toBe(30)
})
```

Pruebas en 01-template

- Creo la misma estructura que mi filesistem en la carpeta test. En lugar de crear otra carpeta src, trato la carpeta test como si fuera src

```
import { emailTemplate } from "../../js-foundation/01-template"

describe("js-foundation/01-template.ts", ()=>{

    test('emailTemplate should contain a greeting', ()=>{

        //Evalúo que tenga la palabra Hi
        expect(emailTemplate).toContain('Hi, ')
    })

    test('emailTemplate should contain {{name}} and {{orderId}} ', ()=>{

        //uso una expresión regular
        expect(emailTemplate).toMatch(/{{name}}/)
        expect(emailTemplate).toMatch(/{{orderId}}/)
    })
})
```

02-destructuring

- Exporto el arreglo de personajes para probar algo
- Pongamos que el orden no me importa, solo quiero determinar que el arreglo contenga Flash

```
import { characters } from "../../js-foundation/02-destructuring";


describe("js-foundation/02-destructuring.ts", ()=>{
    test("characters should contain Flash", ()=>{

        //debe contener Flash
        expect(characters).toContain('Flash')
    })

    test("Flash should be the first character", ()=>{

        //el primero debe ser Flash y el segundo Superman
        const [flash, superman] = characters
        expect(flash).toBe('Flash')
        expect(superman).toBe('Superman')

    })
})
```

03-callbacks

- getUserById recibe un id y un callback
- El callback puede recibir un error y un user

```
export function getUserById( id: number, callback: (err?: string, user?:User) =>
void ) {
  const user = users.find( function(user){
    return user.id === id;
  });

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

  return callback( undefined, user );
}
```

- Puedo probar que si le mando el id que existe me devuelva el user
- Si da error estaría esperando que el usuario sea nulo
- Por el scope, el test termina antes de ejecutar el callbacks
- Debo decirle al test que espere a que tener una resolución positiva o negativa del callback
- Se usa el **done** como argumento que da origen al test
- *NOTA* si coloco un setTimeOut en el getUserById el test esperará a que finalice si pongo **done**
- Hay que llamar al done cuando ya se que tengo los resultados

```
import { getUserById } from "../../js-foundation/03-callbacks"



describe("js-foundation/03-callbacks.ts", ()=>{                          //uso del
done para el callback
    test("getuserById should return an error if user does not exists", (done)=>{
        const id = 10
                        //TS no me obliga a especificarlos porque ambos pueden ser
nulos
        getUserById(id, (err, user)=>{
            expect(err).toBe(`User not found with id ${id}`)
            expect(user).toBe(undefined)

          done() //le dioa a jest no termines la prueba hasta que llame al done
        })

    })

})
```

- En este caso de éxito, para comparar un objeto toBe no sirve porque no apunta al mismo lugar en memoria
- Usaremos toEqual o toStrictEqual

```
test("getUserById should return John Doe", (done)=>{
    const id= 1

    getUserById(id, (err, user)=>{
        expect(err).toBeUndefined()
        expect(user).toEqual({
            id: 1,
            name: "John Doe"
        })
        done()
    })
})
```

Pruebas en 05-Factory

- Por ahora no me interesa que el uuid y la fecha de nacimiento sea la correcta
- Pruebo que devuelva una función

```
import { buildMakePerson } from "../../js-foundation/05-factory";

describe('Factory', () => {

    const getUUID = ()=>'1234'
    const getAge= ()=> 35

    test('buildMakePerson should return a function', () => {

    const makePerson = buildMakePerson({getUUID, getAge})

    expect(typeof makePerson).toBe('function')
    })
})
```

- No te preocupes por tener el 100% del codigo cubierto por el testing
- Probemos que retorna a una persona
- Esto comprueba que en el id de retorno de la persona también retorne el llamado a la función getUUID

```
test('Should return a person', ()=>{

    const makePerson= buildMakePerson({getUUID, getAge}) //pruebo también la
inyección de la dependencia getUUID y getAge
    const JohnDoe= makePerson({name:'John Doe', birthdate: '1985-10-21'})

    //console.log(JohnDoe)

    expect(JohnDoe).toEqual({
```

```
        id:'1234',
        name:'John Doe',
        birthdate:'1985-10-21',
        age:35
    })
})
```

06-Promises

```
import { httpClient as http} from "../plugins";

export const getPokemonById = async( id: string|number ):Promise<string> => {
  const url = `https://pokeapi.co/api/v2/pokemon/${ id }`;

  const pokemon = await http.get( url );

  // const resp = await fetch( url );
  // const pokemon = await resp.json();


  // throw new Error('Pokemon no existe');

  return pokemon.name;

  // return fetch( url )
  //   .then( ( resp ) => resp.json())
  //   // .then( () => { throw new Error('Pokemon no existe') })
  //   .then( ( pokemon ) => pokemon.name );

}
```

- Incluyo el httpClient

```
import axios from 'axios';


export const httpClientPlugin = {

  get: async(url: string ) => {
    const { data } = await axios.get( url );
    return data;
    // const resp = await fetch( url );
    // return await resp.json();
  },

  post: async(url: string, body: any ) => {},
  put: async(url: string, body: any) => {},
  delete: async(url: string ) => {},
```

```
};
```

- Podríamos hacer la prueba con hhtp client, mandarle un mock para simular un mensaje de éxito
- Podemos probar el resultado de la llamada real a pokeapi.co
- A veces voy a querer configurar una db ficticia para crear, borrar, etc
- Cuanto más desacoplado sea el código es más fácil de testear
- Es más fácil probar segmentos pequeños de código
- Probemos primero la situación de existo, debe regresar un pokemon
- Para trabajar con promesas puedo hacer el callback async

```
import { getPokemonById } from "../../js-foundation/06-promises"

describe('06-Promises', ()=>{


    test("Should return a pokemon", async()=>{

        const pokemonId = 1

        const pokemonName = await getPokemonById(pokemonId)

        expect(pokemonName).toBe('bulbasaur')

     })
})
```

- Ahora en caso de que no devuelva un pokemon debe mostrar un error
- Debo manejar el error en 06-promises

```
import { httpClient as http} from "../plugins";

export const getPokemonById = async( id: string|number ):Promise<string> => {

  try {

    const url = `https://pokeapi.co/api/v2/pokemon/${ id }`;

    const pokemon = await http.get( url );

    return pokemon.name;

  } catch (error) {

    throw 'Pokemon no existe!';

  }
  // const resp = await fetch( url );
```

```
  // const pokemon = await resp.json();




  // return fetch( url )
  //   .then( ( resp ) => resp.json())
  //   // .then( () => { throw new Error('Pokemon no existe') })
  //   .then( ( pokemon ) => pokemon.name );

}
```

- Espero que me devuelav la excepción
- Puedo usar un try catch en el test y asegurarme que vaya al catch

```
test('Should return an error if pokemon does not exists', async()=>{
      const pokemonId=12345678


      try {
          const pokemonName= await getPokemonById(pokemonId)
          expect(true).toBe(false) //esto siempre dará error

      } catch (error) {
          expect(error).toBe('Pokemon no existe!')
      }

})
```

Pruebas en getAge plugin

```
export const getAge = ( birthdate: string ) => {

  // return getAgePlugin(birthdate);
  return new Date().getFullYear() - new Date(birthdate).getFullYear();
}
```

- Yo esperaría que esto funcionara

```
import { getAge } from "../../plugins"

describe("get-age.plugin", ()=>{

    test('getAge() should return the age of a person', ()=>{

        const birthDate = '1981-06-30'
```

```
        const age = getAge(birthDate)

        expect(age).toBe(43)
    })

})
```

- Y si, pasa. Pero el año que viene no pasará
- De todas maneras, el código de la función getAge faltaría estar más elaborado
- Por ejemplo hacer una comprobación que el tipo de retorno sea una fecha -Para que el test sea atemporal puedo hacerlo así

```
test('getAge should return current age', ()=>{

    const birthDate = '1981-06-30'

    const age = getAge(birthDate)

    const calculatedAge = new Date().getFullYear() - new
Date(birthDate).getFullYear();

    expect(age).toBe(calculatedAge)
})
```

SpyOn - métodos de objetos

- Puedo simular que el resultado del método getfullYear sea 2020
- Con spyOn puedo cambiar el ADN llamando a la propiedad.protoype y el método que quiero mockear

```
test('getAge() should return 0 years', ()=>{
    const spy = jest.spyOn(Date.prototype, 'getFullYear').mockReturnValue(1995)

    const birthdate = '1995-01-01'

    const age = getAge(birthdate)

    console.log(spy) // para ver los tipos de mock que hay

    expect(age).toBe(0)

    //el spy no solo sirve para hacer un mock
    //puedo comprobar si la función ha sido llamada
    expect(spy).toHaveBeenCalled()
    //expect(spy).toHaveBeenCalledWith() serviría para indicar que getFullyear fue
llamado con algún argumento
})
```

Pruebas en GetUUID Adapter

- No me interesa probar que uuid funciona bien porque ya está altamente probado
- Lo que me interesa probar es que la función funcione como se espera

```
import { getUUID } from "../../plugins"

describe('get-id.plugin', ()=>{

    test('getUUID should return a uuid', ()=>{
        const uuid = getUUID()

        expect(typeof uuid).toBe('string')
        expect(uuid.length).toBe(36)
    })
})
```

HttpClient Adapter

- La mejor estrategia para testing es evaluar pieszas pequeñas para que cuando se vaya a evaluar una pieza grande todas las demás estén evaluadas

```
import axios from 'axios';


export const httpClientPlugin = {

  get: async(url: string ) => {
    const { data } = await axios.get( url );
    return data;
    // const resp = await fetch( url );
    // return await resp.json();
  },

  post: async(url: string, body: any ) => {
    throw new Error('not implemented')
  },
  put: async(url: string, body: any) => {
    throw new Error('not implemented')
  },
  delete: async(url: string ) => {
    throw new Error('not implemented')
  },

};
```

- post, put y delete no están implementados
- Es conveniente lanzar un new Error('not implemented')
- En modo watch pulso w y luego la p para filtrar y pongo http para hacer las pruebas solo en http-client
- Apunto al endpoint con POSTMAN/THUNDERCLIENT y copio el objeto para equipararlo en el test

```
import { httpClientPlugin } from "../../plugins/http-client.plugin"

describe('http-client adapter', ()=>{

    test('httpClientPlugin.get() should return a string', async()=>{

        const data = await
httpClientPlugin.get('https://jsonplaceholder.typicode.com/todos/1')

        expect(data).toEqual({
            "userId": 1,
            "id": 1,
            "title": "delectus aut autem",
            "completed": false
        })
    })
})
```

- Puedo colocar que espere un boolean en completed con

```
"completed": expect.any(Boolean)
```

- Evalúo que tenga las funciones post, put y delete

```
import { httpClientPlugin } from "../../plugins/http-client.plugin"

describe('http-client adapter', ()=>{

    test('httpClientPlugin.get() should return a string', async()=>{

        const data = await
httpClientPlugin.get('https://jsonplaceholder.typicode.com/todos/1')
        expect( data).toEqual({
            "userId": 1,
            "id": 1,
            "title": "delectus aut autem",
            "completed": false
        })
    })

    test('httpClient should have POST, PUT and DELETE methods', async ()=>{

        expect(typeof httpClientPlugin.post).toBe('function')
```

```
        expect(typeof httpClientPlugin.put).toBe('function')
        expect(typeof httpClientPlugin.delete).toBe('function')


    })
})
```

- Podría evaluar si reciben argumentos pero no tiene sentido si no los están usando

---

Logger Adapter

- Lo que tengo que evaluar es el producto resultante

```
import winston, { format } from 'winston';

const { combine, timestamp, json } = format;


const logger = winston.createLogger({
  level: 'info',
  format: combine(
    timestamp(),
    json(),
  ),
  // defaultMeta: { service: 'user-service' },
  transports: [
    //
    // - Write all logs with importance level of `error` or less to `error.log`
    // - Write all logs with importance level of `info` or less to `combined.log`
    //
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});

logger.add(new winston.transports.Console({
  format: winston.format.simple(),
}));

export const buildLogger = (service: string ) => {

  return {
    log: (message: string) => {
      logger.log('info', {message, service});
    },
    error: (message: string ) => {
      logger.error('error', {
        message,
        service,
      });
```

```
    }
  }
}
```

- Probar un logger es dificil
- No esperamos un return si no que se imprima un log
- Hay que probar que se llamen los métodos que se espera
    - Hablando del logger de winston, voy a asegurarme de que el logger haya sido llamado con los argumentos adecuados, o con algún valor
    - No voy a probar que ejecute la construcción, cree los archivos...
    - Lo que me interesa es que haya sido llamado con el servicio esperado y que tenga los métodos log y error en particular
    - Podría inetersarme evaluar otras cosas...
- Evalúo que logger devuelva la funciones log y error

```
import { buildLogger } from "../../plugins"

describe('logger.plugin', ()=>{

    test('buildLogger should return a function logger', ()=>{

        const logger = buildLogger('test')

        expect(typeof logger.log).toBe('function')
        expect(typeof logger.error).toBe('function')
    })
})
```

- Esto nos asegura que siempre tenga la función .log (¡y que sea una función!)
- Para evaluar que el logger ha sido llamado hay muchas formas de hacer un mock al respecto (un mock de winston)
- No me interesa hacer un mock completo, solo que ha sido llamado
- Para ello exporto el logger del archivo original
- Podría evalauar muchas cosas del logger pero lo que me interesa es usar el espía para comprobar que ha sido llamado

```
import { buildLogger, logger as winstonLogger } from "../../plugins"

describe('logger.plugin', ()=>{

    test('buildLogger should return a function logger', ()=>{

        const logger = buildLogger('test')

        expect(typeof logger.log).toBe('function')
        expect(typeof logger.error).toBe('function')
    })
```

```
    test('logger has been called', ()=>{
        //preparación
        const winstonLoggerMock= jest.spyOn(winstonLogger, 'log')
        const message = 'test message'

        const service= 'test service'

        //estímulo
        const logger = buildLogger(service)

        logger.log(message)

        //aserciones
        expect(winstonLoggerMock).toHaveBeenCalled()

    })
})
```

- Podría usar el toHaveBeenCalledWith({}) y pasarle un objeto vacío, en consola me indicará todos los valores como info, otro objeto con contros valores
- Puedo comprobar que ha sido mandado con esos valores copiando de la respuesta de error

---

testing Coverage

> npm run test:coverage

- Da un resumen general del código evaluado
- En la carpeta coverage hay un html dinámico (doble click)
- Es recomendable que esté por encima del 70% cubierto
- Si el testing falla puedo prevenir el build de producción

Build + testing

- Para evitar que haga el build si algún test falla
- Instalo rimraf
- En el package.json

```
{
  "build": "npm run test && rimraf ./dist && tsc",
  "start": "node dist/app.js"
}
```

NODE_TS - NOC

- Instalaciones necesarias (sin nodemon)

> npm init -y npm i -D typescript @types/node ts-node-dev rimraf npx tsc --init --outDir dist/ --rootDir src

- Scripts

```
"dev": "tsnd --respawn src/app.ts",
"build": "rimraf ./dist && tsc",
"start": "npm run build && node dist/app.js"
```

- En el ts-config, fuera del "compilerOptions" excluyo los node_modules y la dist/ e incluyo el src

```
"exclude": [
    "node_modules",
    "dist"
],
"include":[
    "src"
]
```

Main - Server app

- Para esta aplicación vamos a hacer uso de arquitectura limpia y el patrón repositorio
- Creo la carpeta presentation con server.ts
- Creo la clase Server con el método start
- Poner public en el método es redundante ya que por defecto es un método público, pero ayuda a la legibilidad
- Con static no necesito hacer una instancia de la clase y puedo ussar el método directamente con Server
- server.ts

```
export class Server {

    public static start(){

        console.log("Server started!")
    }
}
```

- Pongo en marcha el servidor con .start() en app.ts llamando a main dentro de función autoinvocada (puede ser async)

```
import { Server } from "./presentation/server"
```

```
(async ()=>{
    main()
})()

function main(){
    Server.start()
}
```

CRON Tasks

> npm i cron

- Uso CronJob
- Los asteriscos se refieren a los segundos, minutos, horas, etc (mirar documentación)

```
import { CronJob } from "cron"

export class Server {

    public static start(){
        const job = new CronJob(
            '*/2 * * * * *', //cada 2 segundos
            ()=>{
                const date = new Date()
                console.log('2 seconds', date)
            }
        )

        job.start()
    }
}
```

- De esta manera el código **está fuertemente acoplado**
- Siempre que uses librerías de terceros usa **EL PATRÓN ADAPTADOR**
- Creo la carpeta cron dentro de presentation (capa de presentación, de cara al usuario) con el archivo cron-service.ts
- Creo la clase CronService. Cómo no voy a usar inyección de dependencias (no necesito instancia), solo quiero un método, lo hago estático
- **Retorno el job** por si quiero luego usar el .stop para detenerlo
- Pego el código del método start del server

```
import { CronJob } from "cron"

export class CronService{

    static createJob(): CronJob{
```

```
        const job = new CronJob(
            '*/2 * * * * *', //cada 2 segundos
            ()=>{
                const date = new Date()
                console.log('2 seconds', date)
            }
        )

        job.start()

        return job
    }
}
```

- Lo llamo en el server

```
import { CronService } from "./cron/cron-service";


export class Server {

    public static start(){
        CronService.createJob()
    }
}
```

- **NOTA: con CTRL + click encima de un método o clase voy a la información de las definiciones de TypeScript en index.d.ts**
- Necesito pasarle los parámetros al createJob para configurar el tiempo (**cronTime**) que quiero para que se ejcute cada vez, y la función que es **onTick**
- Necesito tipar estos dos parámetros. Cuando son objetos es mejor **interfaces**, pero cuando es un tipo de dato es mejor con un **type**
- Cuando hay **más de dos argumentos** se recomienda **usar un objeto y mandar un único argumento**

```
import { CronJob } from "cron"


type CronTime = string | Date;
type OnTick = ()=> void

export class CronService{

    static createJob(cronTime: CronTime, onTick: OnTick ): CronJob {

        const job = new CronJob( cronTime, onTick)

        job.start()
```

```
        return job
    }
}
```

- Obviamente en server tengo que pasarle los parámetros al método createJob

```
import { CronService } from "./cron/cron-service";


export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{
            const date = new Date();
            console.log('5 seconds', date)
        })
    }
}
```

- Hemos **aplicado el patrón adaptador**
- cron tiene algo llamado ChildProcess que permite el multihilo

---

CheckService - UseCase

- Los casos de uso los colocaremos en la capa de dominio (src/**domain**/**use-cases**)
- Dentro de use-cases voy a ir creando diferemtes carpetas según el caso de uso, checks, logs, etc
- Creo un método **async** al que le paso la url de mi web (por ejemplo, para chequear que funciona) y devuelve una promesa con un valor booleano
- Si la web no estuviera operativa me gustaría enviar un email o crear un log, notificar de alguna manera, al menos después de 5 intentos (por ejemplo)
- Esto es **un caso de uso**, un código dedicado a una tarea en específico
- Siempre es bueno crear interfaces para determinar cierto comportamiento y ayudar a comprender el código a otras personas
- No hago el código estático porque si voy a hacer inyección de dependencias
- Uso un **try catch**

```
interface CheckServiceUseCase{
    execute(url: string):Promise <boolean>
}

export class CheckService implements CheckServiceUseCase{


    async execute(url: string): Promise <boolean>{

        try {
            const req = await fetch(url)
```

```
            if(!req.ok){
                throw new Error(`Error on check service ${url}`)
            }

            console.log(`${url} is ok!`)
            return true

        } catch (error) {
            console.log(`${error}`)

            return false
        }

    }
}
```

- Ahora llamo este caso de uso en el server

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { CronService } from "./cron/cron-service";


export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

            new CheckService().execute('https://google.com')
        })
    }
}
```

- **Resumen**:
  - Creamos el **Server** en la capa de presentación
  - Creamos el **CronService** para realizar **el patrón adaptador** con la librería cron
    - **Defino los tipos de los argumentos** del método estático
    - **Pongo en marcha el servicio** de cron dentro del método con una nueva instancia pasándole los argumentos como parámetros y llamando a **.start**
    - **Retorno el job** por si quiero hacer algo con el cómo usar .stop
  - Creo el primer **caso de uso en la capa de dominio, CheckService**
    - Implemento una **interfaz** con el único método execute al que le paso una url
    - No lo hago estático porque quiero inyectarlo
    - En un **try catch hago el fetch** de la url pasada
    - Si el fetch no me devuelve el ok en la request mando **un error**. **Devuelvo un true**
    - Si el catch pilla un error, lo imprimo
    - Llamo al método **generando una nueva instancia del CheckService pasándole la url** al método **execute**

JSON-Server

- Para testear el servicio y montar un server REST API usaremos JSON Server
- Creo una nueva carpeta, hago el npm init -y

> npm i json-server

- Creo el db.json y copio el contenido de la web de ejemplo
- Para levantar el server creo el script

```
"start": "json-server --watch db.json"
```

- Cambio la url del servicio (caso de uso) CheckService

```
export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

            new CheckService().execute('http://localhost:3000')
        })
    }
}
```

Inyección de dependencias

- La inyección de dependencias es colocar una dependencia en los casos de uso, repositorios, data sources, etc
- Se suele realizar en un **constructor**
- Tipo los casos de si sale bien y si hay un error
- Voy a recibir estos dos argumentos en el constructor del CheckService
- Uso **private readonly** porque yo no quiero cambiar el SuccessCallback accidentalmente
- LLamo a las funciones en sus lugares correspondientes

```
interface CheckServiceUseCase{
    execute(url: string):Promise <boolean>
}

type SuccessCallback = ()=> void  //tipo de lo que quiero ejecutar si todo sale
bien
type ErrorCallback = (error: string)=> void //tipo si hay algún error

export class CheckService implements CheckServiceUseCase{

    constructor(
```

```
        private readonly successCallback: SuccessCallback,
        private readonly errorCallback: ErrorCallback
        ){}


    async execute(url: string): Promise <boolean>{

        try {
            const req = await fetch(url)

            if(!req.ok){
                throw new Error(`Error on check service ${url}`)
            }

            this.successCallback() //llamo al SuccessCallback


            return true

        } catch (error) {

            this.errorCallback(`${error}`) //llamo al ErrorCallback

            return false
        }

    }
}
```

- Ahora solo tengo que pasarle las funciones al crear la instancia de CheckService

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { CronService } from "./cron/cron-service";


export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

            new CheckService(
                ()=> console.log("Success!"),
                (error)=> console.log(`${error}`)
            ).execute('http://localhost:3000')
        })
    }
}
```

- El objetivo de todo esto es separar responsabilidades

Cierre de sección

- Los logs en consola son útiles pero si tuvieramos varios servicios generaría mucho ruido y tampoco es conveniente
- El caso de uso debería recibir **dónde es que quiero grabar estos logs**
- Vamos a usar una de las funciones (SuccessCallback o ErrorCallback) para grabar en la DB con un sistema personalizado de logs mediante inyección de dependencias

---

NODE_TS NOC - Patrón Repositorio

Introducción

- **Data Source** es el origen de los datos
- Vamos a empezar a trabajar con el filesystem. Vamos a grabar logs en él
- Pero va a dar igual si es en el filesystem, en una DB...
- El objetivo es hacer una app en la que esto no importe, o que sea intercambiable sin dolor
- Aplicando **el patrón repositorio**, voy a crear un repositorio que se conecta al data-source, y mi caso de uso llamará al repositorio.
- De esta manera si necesito cambiar de Data source no hay problema, e implica una capa de seguridad

---

LogEntity

- Toda la lógica de negocio está en el **domain**
- Creo en domain la carpeta entities con log.entity.ts
- Otros ejemplos serían client.entity, product.entity...
- Con esto podré crear instancias de LogEntity

```
export enum LogSeverityLevel{
    low    = 'low',
    medium = 'medium',
    high   = 'high'
}


export class LogEntity{

    public level: LogSeverityLevel
    public message: string
    public createdAt: Date

    constructor(message: string, level: LogSeverityLevel){
        this.level = level
        this.message = message
        this.createdAt = new Date()
    }
}
```

Datasources y Repositorios Abstractos

- Creo la carpeta **datasources** y **repository** en domain
- Es el origen de los datos, una DB, filesystem...
- repository es desde dónde vamos a llamar al datasource
- En el domain solo son las reglas, en domain **no hacemos la implementación**
- Creo datasource/log.datasource.ts
- Utilizo una clase abstracta
- Las clases abstractas **no pueden ser instanciadas**
- Sirve para obligar el comportamiento que quiero definir en este datasource sobre otras clases

```
import { LogEntity, LogSeverityLevel } from "../entities/log.entity";

export abstract class LogDataSource{

    //cualquier origen de datos va a tener que implementar saveLog
    abstract saveLog(log: LogEntity): Promise<void>;
    abstract getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>
}
```

- Esta es básicamente la implementación de nuestras **reglas de negocio para los datasources**
- Es un contrato el cual **todos mis datasources tienen que cumplirlo**
- En el repository copio el código y le cambio el nombre

```
import { LogEntity, LogSeverityLevel } from "../entities/log.entity";

export abstract class LogRepository{

    //Me permite llamar métodos que hay en el datasource
    abstract saveLog(log: LogEntity): Promise<void>;
    abstract getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>
}
```

FileSystem - Datasource

- Creo la carpeta src/**infraestructure/datasources**
- Hay gente que le pone otros nombres.
- De esta manera estan en orden: domain (reglas de negocio), infraestructura y presentacion
- Presentación va a llamar cosas de infraestructura e infraestructura va a seguir las reglas de domain
- En la carpeta datasources tengo **el código que yo me comprometí a cumplir en log.datasource**
- Creo el archivo file-system.datasource.ts
- Una vez escrita la clase, implemento LogDatasource.

NOTA: Con **Ctrl+ .** sobre el nombre de la clase me da la opción de implementar los métodos automáticamente

```
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";

export class FileSystemDatasource implements LogDataSource{

    private readonly logPath = 'logs/' //mis logs se grabarán en este PATH

    saveLog(log: LogEntity): Promise<void> {
        throw new Error("Method not implemented.");
    }
    getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {
        throw new Error("Method not implemented.");
    }
}
```

- Almacenaré los logs según el nivel de severidad
- Necesitamos tener creados los directorios para guardar los logs para que no de error
- **Llamo al método en el constructor** para que al generar una instancia compruebe si existe el directorio logs y si no lo crée
- También quiero crear los archivos en caso de que no existan
- Puedo crear **un arreglo** con los paths de los archivos, y con un **forEach** usar **fs.writeFileSync** para crearlos en caso de que no existan
- Inserto **un string vacío**

```
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";
import fs from 'fs'

export class FileSystemDatasource implements LogDataSource{

    private readonly logPath = 'logs/' //mis logs se grabarán en este PATH
    private readonly allLogsPath = 'logs/logs-all.log'
    private readonly mediumLogsPath = 'logs/logs-medium.log'
    private readonly highLogsPath = 'logs/logs-high.log'


    constructor(){
        this.createLogsFiles() //cuando se genere una instancia llamaremos al
método para que verifique si existe logs/ y si no existe lo crée
    }

    //es private porque no quiero que se use fuera de este Datasource
    private createLogsFiles =()=>{

        if(!fs.existsSync(this.logPath)){
            fs.mkdirSync(this.logPath)
```

```
        }

        [
            this.allLogsPath,
            this.mediumLogsPath,
            this.highLogsPath

        ].forEach(path =>{
            if(fs.existsSync(path)) return
               fs.writeFileSync(path, '') //si no existe creo el archivo y le
inserto un string vacío
        })
    }

    saveLog(log: LogEntity): Promise<void> {
        throw new Error("Method not implemented.");
    }
    getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {
        throw new Error("Method not implemented.");
    }
}
```

- Hacer esto con la db es mucho más fácil!!

FileSystem - SaveLog

- Usaré el método saveLog para guardar según la severidad
- Todos los logs los grabaré en allLogsPath, y luego también separaré por medium y high
- **appendFileSync añade una linea al final**
- **JSON.stringify** serializa un objeto como un JSON (le pone comillas dobles, dos puntos entre propiedad y propiedad)
- Coloco un return Promise.resolve() para acabar con la instrucción y no de error (salta error si coloco solo un return)

```
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";
import fs from 'fs'

export class FileSystemDatasource implements LogDataSource{

    private readonly logPath = 'logs/' //mis logs se grabarán en este PATH
    private readonly allLogsPath = 'logs/logs-all.log'
    private readonly mediumLogsPath = 'logs/logs-medium.log'
    private readonly highLogsPath = 'logs/logs-high.log'


    constructor(){
        this.createLogsFiles() //cuando se genere una instancia llamaremos al
método para que verifique si existe logs/ y si no existe lo crée
```

```
    }

    //es private porque no quiero que se use fuera de este Datasource
    private createLogsFiles =()=>{

        if(!fs.existsSync(this.logPath)){
            fs.mkdirSync(this.logPath)
        }

        [
            this.allLogsPath,
            this.mediumLogsPath,
            this.highLogsPath
        ].forEach(path =>{
            if(fs.existsSync(path)) return
               fs.writeFileSync(path, '') //si no existe creo el archivo y le
inserto un string vacío
        })
    }

    saveLog(newLog: LogEntity): Promise<void> {

        fs.appendFileSync(this.allLogsPath, `${JSON.stringify(newLog)}\n`)

        if(newLog.level === LogSeverityLevel.low) return Promise.resolve()
        if(newLog.level === LogSeverityLevel.medium){
            fs.appendFileSync(this.mediumLogsPath, `${JSON.stringify(newLog)}\n`)
        }else{
            fs.appendFileSync(this.highLogsPath, `${JSON.stringify(newLog)}\n`)
        }
        return Promise.resolve()
    }
    getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {
        throw new Error("Method not implemented.");
    }

}
```

- Para no repetir JSON.stringify(newLog) tantas veces puedo guardarlo en una variable

```
const logAsJson = `${JSON.stringify(newLog)}\n`
```

---

getLogs

- saveLog está guardando los logs en un archivo .log (no JSON)
- luce algo así

```
{"level": "medium", "message": "hola mundo", "createdAt": "21746TZ6546874145"}
{"level": "medium", "message": "hola mundo", "createdAt": "21746TZ6546874145"}
{"level": "medium", "message": "hola mundo", "createdAt": "21746TZ6546874145"}
{"level": "medium", "message": "hola mundo", "createdAt": "21746TZ6546874145"}
```

- Yo necesito regresar un arreglo de LogEntity
- Estos objetos no son lo mismo que un LogEntity, puede que yo tenga un método en LogEntity.
- Este objeto no los va a tener
- Tenemos que transformar este objeto {"level": "medium", "message": "hola mundo", "createdAt": "21746TZ6546874145"} en una entidad
- Voy a crear un switch para filtrar según severidad
- Creo una función para obtener los logs del archivo
- Uso un ternario para evaluar si el json viene vacío

```
 private getLogsFromFile =(path: string): LogEntity[]=>{
        const content = fs.readFileSync(path, 'utf-8')
        if(content=== '') return []
   }

    getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {

        switch (severityLevel) {
            case LogSeverityLevel.low:

                break;
            case LogSeverityLevel.medium:

                break;

            case LogSeverityLevel.high:

                break;

            default:

            throw new Error(`${severityLevel} not implemented`)
                break;
        }
    }
```

- Creo un método **static porque no quiero crear una instancia para poderlo llamar**
- Podríamos verlo como un **Factory Constructor** en log.entity

```
static fromJson = (json: string): LogEntity =>{
    //coloco un ternario por si viene vacío (sin logs) no me cree una entidad con
los valores undefined desde getLogsFromFile
    json = (json === '{}') ? '{}': json
```

```
    const {message, level, createdAt}= JSON.parse(json) //parseo el string del
archivo .log a formato JSON
    if(!message) throw new Error("message is required") //validaciones
    if(!level) throw new Error("message is required")

    const log = new LogEntity(message, level) //genero la instancia con la data de
la desestructuración
    log.createdAt = new Date(createdAt) //le paso la data del log
    return log
}
```

- Para barrer el archivo por lineas desde getLogsFromFile, podemos usar el split y pasarle la separación (\n)
- Luego usar un map para que convierta cada elemento en una entidad **con el método estático formJson**
- **En JS cuando tenemos el mismo argumento como mismo parámetro de la función se puede poner solo la declaración de la función (sin paréntesis)**
- No necesito hacer una nueva instancia ya que es un método estático
    - Es decir, en lugar de **.map(arg=> LogEntity.fromJson(arg)) se puede escribir .map(LogEntity.fromJson)**
    - Si no hay logs devuelvo un array vacío, así no llama al fromJson

```
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";
import fs from 'fs'

export class FileSystemDatasource implements LogDataSource{

    private readonly logPath = 'logs/' //mis logs se grabarán en este PATH
    private readonly allLogsPath = 'logs/logs-all.log'
    private readonly mediumLogsPath = 'logs/logs-medium.log'
    private readonly highLogsPath = 'logs/logs-high.log'


    constructor(){
        this.createLogsFiles() //cuando se genere una instancia llamaremos al
método para que verifique si existe logs/ y si no existe lo crée
    }


    //es private porque no quiero que se use fuera de este Datasource
    private createLogsFiles =()=>{

        if(!fs.existsSync(this.logPath)){
            fs.mkdirSync(this.logPath)
        }

        [
            this.allLogsPath,
```

```
            this.mediumLogsPath,
            this.highLogsPath
        ].forEach(path =>{
            if(fs.existsSync(path)) return
               fs.writeFileSync(path, '') //si no existe creo el archivo y le
inserto un string vacío
        })
    }

    async saveLog(newLog: LogEntity): Promise<void> {

        const logAsJson = `${JSON.stringify(newLog)}\n`
        fs.appendFileSync(this.allLogsPath, logAsJson)

        if(newLog.level === LogSeverityLevel.low) return
        if(newLog.level === LogSeverityLevel.medium){
            fs.appendFileSync(this.mediumLogsPath, logAsJson)
        }else{
            fs.appendFileSync(this.highLogsPath, logAsJson)
        }
        return
    }


   private getLogsFromFile = (path: string): LogEntity[]=>{
        const content = fs.readFileSync(path, 'utf-8')

        //si no hay logs devuelvo un array vacío
        if(content === '') return []

        const logs = content.split('\n').map(LogEntity.fromJson)

        return logs
   }

   async getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {

        switch (severityLevel) {
            case LogSeverityLevel.low:

            return this.getLogsFromFile(this.allLogsPath)

            case LogSeverityLevel.medium:

            return this.getLogsFromFile(this.mediumLogsPath)

            case LogSeverityLevel.high:

            return this.getLogsFromFile(this.highLogsPath)

            default:

            throw new Error(`${severityLevel} not implemented`)
```

```
        }
    }
}
```

LogRepository Implementation

- Nuestra clase abstracta LogRepository debe tener los métodos saveLog y getLogs
- Se llaman igual que los de nuestro Datasource porque el repositorio va a llamar al DataSource
- Creo la carpeta infraestructure/**repository** con LogRepository.ts
- Implementa la clase abstracta LogRepository. **Con Ctrl + . autocompleta los métodos**
- Inyecto el Datasource en el constructor de la forma abreviada con **private readonly**
- Lo uso para llamar a los métodos y le paso el log a saveLog y el severityLevel a getLogs

```
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";
import { LogRepository } from "../../domain/repository/log.repository";

export class LogRepositoryImpl implements LogRepository{

    constructor(
        private readonly logDataSource: LogDataSource
    ){}

    async saveLog(log: LogEntity): Promise<void> {
        return await this.logDataSource.saveLog(log)
    }
    async getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]> {
        return await this.logDataSource.getLogs(severityLevel)
    }

}
```

Inyectar repositorio en caso de uso

- Voy al CheckService. Inyecto el repositorio
- Si todo sale bien puedo guardar con logRepository.saveLog
- Necesito guardar una nueva instancia de LogEntity

```
import { LogEntity, LogSeverityLevel } from "../../entities/log.entity"
import { LogRepository } from "../../repository/log.repository"

interface CheckServiceUseCase{
    execute(url: string):Promise <boolean>
}

type SuccessCallback = ()=> void  //tipo de lo que quiero ejecutar si todo sale
```

```
bien
type ErrorCallback = (error: string)=> void //tipo si hay algún error

export class CheckService implements CheckServiceUseCase{

    constructor(
        private readonly logRepository: LogRepository,
        private readonly successCallback: SuccessCallback,
        private readonly errorCallback: ErrorCallback
        ){}


    async execute(url: string): Promise <boolean>{

        try {
            const req = await fetch(url)

            if(!req.ok){
                throw new Error(`Error on check service ${url}`)
            }

            //Si ha ido bien puedo guardar el log con LogRepository

            const log = new LogEntity(`Service ${url} working`,
LogSeverityLevel.low )
            this.logRepository.saveLog(log)
            this.successCallback() //llamo al SuccessCallback si todo sale bien


            return true

        } catch (error) {

            const errorMessage = `${error}`
            const log = new LogEntity(errorMessage, LogSeverityLevel.low )

            this.logRepository.saveLog(log)
            this.errorCallback(errorMessage) //llamo al ErrorCallback

            return false
        }

    }
}
```

- Falta la inyección de la dependencia en server.ts en la nueva instancia de CheckService
- En el server necesito crear la nueva instancia que van a usar todos los servicios
- Creo una nueva instancia de LogRepositoryImpl fuera del server y le paso el fileSystemDataSource
- Se lo paso a la nueva instancia de CheckService

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";


const fileSystemRepository = new LogRepositoryImpl(
    new FileSystemDatasource()
)


export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

            new CheckService(
                fileSystemRepository,
                ()=> console.log("Success!"),
                (error)=> console.log(`${error}`)
            ).execute('https://google.es')
        })
    }
}
```

Variables de entorno

- Creo el archivo .env en la raíz del proyecto
- Lo añado a gitignore, también el directorio logs/ (no les voy a dar seguimiento desde el repositorio si no con mails)
- En .env creo MAILER_EMAIL, con el password de mi email y el puerto de la app
- Como .env no va a estar en el repositorio, es conveniente crear un .env.template donde dejar las variables vacías o con valores por defecto

```
PORT=3000
MAILER_EMAIL=ismaelberoncastano@gmail.com
MAILER_SECRET_KEY=mi_password_email
PROD=true
```

- Debería haber un proceso que valide que lo que hay colocado en la variable de entorno es un correo válido
- Puedo observar lo que hay en las variables de entorno (hay muchas) con un console.log a **process.env**
    - Aqui no van a aparecer las que yo he definido si no hay configuración (y una conveniente validación)
- Para tener disponibles las variables de entorno instalamos **dotenv**
- Para usarlo necesito usar el patrón adaptador, pero por ahora importo **dotenv/config**

- Ahora ya tengo disponibles las variables de entorno

```
import 'dotenv/config'
import { Server } from "./presentation/server"

(async ()=>{
    main()
})()

function main(){
    Server.start()
    console.log({email: process.env.MAILER_EMAIL})
}
```

- El paquete **env-var** (no tiene dependencias) nos va a permitir hacer validaciones
- Creo la carpeta src/**config**/**plugins** con el archivo envs.plugin.ts
- Importo env y todo de env-var como env
- Ahora puedo usar el .get, decir que el PORT es requerido, y que debe de ser un entero positivo

```
import 'dotenv/config'
import * as env from 'env-var'

const PORT: number = env.get('PORT').required().asIntPositive()
```

- Puedo tambien exportarlo como un objeto para disponer del tipado

```
import 'dotenv/config'
import * as env from 'env-var'

export const envs = {
    PORT: env.get('PORT').required().asPortNumber()
}
```

- En el server

```
import { envs } from "./config/plugins/envs.plugin"
import { Server } from "./presentation/server"

(async ()=>{
    main()
})()

function main(){
    Server.start()
    console.log(envs.PORT)
}
```

- Hago lo mismo con el resto de variables de entorno

```
import 'dotenv/config'
import * as env from 'env-var'

export const envs = {
    PORT: env.get('PORT').required().asPortNumber(),
    MAILER_EMAIL:env.get('MAILER_EMAIL').required().asEmailString(),
    MAILER_SECRET_KEY: env.get('MAILER_SECRET_KEY').required().asString(),
    PROD: env.get('PROD').required().asBool()
}
```

- Podría ponerle un valor por defecto a PROD con .default(false)

README.md

- Con procedimientos no standards (no solo lanzar un npm run start) hay que definirlo en un README.md

```
# Proyecto NOC

- Aplicación de monitoreo usando Arquitectura Limpia con TypeScript

# dev

1. Clonar el archivo .env.template a .env
2. Configurar las variables de entorno

````
PORT=3000
MAILER_EMAIL=
MAILER_SECRET_KEY=
PROD

````
3. Ejecutar npm i
4. ejecutar npm run dev
```

---

# NODE_TS NOC - EMAILS

- Comento el CronService en el Server para evitar que haga mucho ruido durante el ejercicio
- Refactoricemos
- Hay cosas dolorosas, como cuando una entidad cambia

- Hay una forma de prepararse para ello
- Creo una nueva propiedad origin en LogEntity
- Necesito especificar el origen en el constructor. Puede ser el UseCase o el Service, por ejemplo
- Quiero que en mis logs aparezca en que archivo fue que yo llamé ese log (en un caso de uso, el servicio, etc)
- Si en el futuro necesitara añadir otra propiedad debería volver a cambiar el constructor yarreglar todos los inconvenientes generados en cascada
- Una solución es crear una Factory Function que reciba los argumentos que se esperan para generar una nueva instancia de la entidad
- Por ahora solo será añadir origin al constructor
- Según el Clean Code, cuando tienes 3 argumentos en un método es mejor mandar un objeto
- Creo una interfaz para tipar el objeto que mandaré en el constructor
    - createAt lo hago opcional, porque si no lo recibo lo voy a establecer
- Desestructuro de options las propiedades. Si no viene el createdAt va a ser igual a new Date
- Cuando creo la instancia en el método estatico fromJSON me da error porque recibe dos argumentos y se esperaba uno (el objeto options)
- En origin coloco el origin desestructurado, pero aquí iría el string 'log.entity'
- *NOTA*: Para que el fromJson no de error si no hay logs en el archivo, **uso un ternario**
    - También **debo colocar un return** tras evaluar la condicion en getLogsFromFile **para que no llame al fromJson** en caso de que venga vacío
- domain/entities/log.entities.ts
-

```
export enum LogSeverityLevel{
    low    = 'low',
    medium = 'medium',
    high   = 'high'
}

export interface LogEntityOptions{
     level: LogSeverityLevel
     message: string
     createdAt?: Date
     origin: string
}


export class LogEntity{


    public level: LogSeverityLevel
    public message: string
    public createdAt: Date
    public origin: string

    constructor(options: LogEntityOptions){
        const {message, level, createdAt = new Date(), origin}= options

        this.message = message
```

```
        this.level = level
        this.createdAt = new Date()
        this.origin = origin
    }

    static fromJson = (json: string): LogEntity =>{
        json = (json==='{}') ? '{}': json //para que no de error cuando esté vacio

        //para que no cree una entidad con undefined si viene el json vacio(no hay
logs), evito que llame el fromJSOn en getLogsFromFile
        const {message, level, createdAt}= JSON.parse(json)
        if(!message) throw new Error("message is required")
        if(!level) throw new Error("message is required")

        const log = new LogEntity({    //le paso las propiedades al constructor en
un objeto
            message: message,
            level: level,
            createdAt: createdAt,
            origin: origin
        })

        return log
     }
}
```

- En infraestructure/datasources/file-system.datasource

```
private getLogsFromFile = (path: string): LogEntity[]=>{
        const content = fs.readFileSync(path, 'utf-8')

        if(content=== "") return []//aquí si no hay contenido devuelvo un array
vacío

        const logs = content.split('\n').map(LogEntity.fromJson)

        return logs
   }
```

- Esto va a generar una serie de errores en todos los lugares dónde usamos LogEntity.
- En este caso es solo en el caso de uso
- Debo pasarle el objeto a las nuevas instancias de LogEntity
- domain/use-cases/checks/check-service.ts

```
import { LogEntity, LogSeverityLevel } from "../../entities/log.entity"
import { LogRepository } from "../../repository/log.repository"

interface CheckServiceUseCase{
    execute(url: string):Promise <boolean>
```

```
}

type SuccessCallback = ()=> void  //tipo de lo que quiero ejecutar si todo sale
bien
type ErrorCallback = (error: string)=> void //tipo si hay algún error

export class CheckService implements CheckServiceUseCase{

    constructor(
        private readonly logRepository: LogRepository,
        private readonly successCallback: SuccessCallback,
        private readonly errorCallback: ErrorCallback
        ){}


    async execute(url: string): Promise <boolean>{

        try {
            const req = await fetch(url)

            if(!req.ok){
                throw new Error(`Error on check service ${url}`)
            }

            //Si ha ido bien puedo guardar el log con LogRepository

            const log = new LogEntity({
                message:`Service ${url} working`,
                level: LogSeverityLevel.low,
                origin: 'check-service.ts' })
            this.logRepository.saveLog(log)
            this.successCallback() //llamo al SuccessCallback si todo sale bien


            return true

        } catch (error) {

            const errorMessage = `${error}`
            const log = new LogEntity({  //debo pasarle el objeto a la instancia
de LogEntity
                message:errorMessage,
                level: LogSeverityLevel.low,
                origin: ' check.service.ts' })

            this.logRepository.saveLog(log)
            this.errorCallback(errorMessage) //llamo al ErrorCallback

            return false
        }

    }
}
```

- Podemos crear una variable para no poner el string en duro
- Es más facil mandar estos objetos y luego usar desestructuración para refactorizar, especialmente con más de tres argumentos

---

## Preparación de envío de correos

- En lugar de usar mailtrap o similares usaremos la propia cuenta de gmail
- Hay que hacer unas configuraciones en la política de contraseñas y las politicas de seguridad de gmail
- Ocupamos un secret_key que google nos va a dar
- Gmail Keys
    - Hay que habilitar el key y el two factor auth
    - En myAccount/seguridad/verificacion en dos pasos (hay que tener activada la verificación en dos pasos)/contraseña de aplicaciones
    - Selecciono correo y le pongo de nombre NOC
    - Copio el código en .env en MAILER_SECRET_KEY

---

## NodeMailer

- Instalo nodemailer y los tipos con @types/nodemailer
- Creo un nuevo servicio en /presentation/email/email.service.ts
- Teniendo esto es un archivo reutilizable que puedes copiar en cualquier proyecto en el que quieras enviar mails
- Para configurar nodemailer necesitamos establecer el transporter
- Yo puedo querer cambiar estos valores dentro del transporter por lo que uso variabels de entorno
    - Recuerda poner la variable vacía en env.template para saber que hay que introducirla ya que .env no se subirá a GIT
    - Coloco la variable de entorno en env.plugin
- config/plugins/envs.plugins.ts

```
import 'dotenv/config'
import * as env from 'env-var'

export const envs = {
    PORT: env.get('PORT').required().asPortNumber(),
    MAILER_EMAIL:env.get('MAILER_EMAIL').required().asEmailString(),
    MAILER_SERVICE: env.get('MAILER_SERVICE').required().asString(),
    MAILER_SECRET_KEY: env.get('MAILER_SECRET_KEY').required().asString(),
    PROD: env.get('PROD').required().asBool()
}
```

- El servicio
- presentation/email/email.service.ts

```
import nodemailer from 'nodemailer'
import { envs } from '../../config/plugins/envs.plugin'

interface SendEmailOptions{
    to: string
    subject: string
    htmlBody: string
    //TODO:attachments
}

export class EmailService{
    private transporter= nodemailer.createTransport({
        service: envs.MAILER_SERVICE,
        auth:{
            user: envs.MAILER_EMAIL,
            pass: envs.MAILER_SECRET_KEY
        },
        tls: {
            rejectUnauthorized: false
        }
    })

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody
                })

                console.log(sentInformation)

                return true
            } catch (error) {

                console.log(error)

                return false
            }
    }
}
```

- Creo una nueva instancia en el server
- Lo hago así porque voy a usar inyección de dependencias, si no usaría un método estático

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
```

```
system.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";


const fileSystemRepository = new LogRepositoryImpl(
    new FileSystemDatasource()
)


export class Server {

    public static start(){
        //CronService.createJob('*/5 * * * * *', ()=>{

          //  new CheckService(
            //    fileSystemRepository,
            //    ()=> console.log("Success!"),
            //    (error)=> console.log(`${error}`)
            //).execute('https://google.es')
        // })

        const emailService = new EmailService()

        emailService.sendEmail({
            to:'bercast81@gmail.com',
            subject: 'Logs de sistema',
            htmlBody:`
            <h3>Logs de sistema</h3>
            <p>Este es un mail para los logs de sistema</p>`
        })
    }
}
```

## Enviar archivos

- Hay varias maneras
- Puedo crear el archivo, crear un buffer (mirar documentación Nodemailer)
- Vamos a usar el path y el fileName
- Creo otro método en el servicio
- Creo attachments del tipo arreglo de Attachment. Lo desestructuro y lo agrego al transporter

```
import nodemailer from 'nodemailer'
import { envs } from '../../config/plugins/envs.plugin'

interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
```

```
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{
    private transporter= nodemailer.createTransport({
        service: envs.MAILER_SERVICE,
        auth:{
            user: envs.MAILER_EMAIL,
            pass: envs.MAILER_SECRET_KEY
        },
        tls: {
            rejectUnauthorized: false
        }
    })

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
                    attachments
                })

                console.log(sentInformation)

                return true
            } catch (error) {

                console.log(error)

                return false
            }
    }

    async sendemailWithFileSystemLogs(to: string | string[]){
            const subject= 'Logs del servidor'
            const htmlBody=`
            <h3>Logs del sistema</h3>
            <p>Desde sendEmailWithFileSystem</p>
            `

        const attachments: Attachment[]= [
            {filename: 'logs-all.log', path: './logs/logs-all.log'},
            {filename: 'logs-high.log', path: './logs/logs-high.log'},
            {filename: 'logs-medium.log', path: './logs/logs-medium.log'},
```

```
        ]
        return this.sendEmail({to, subject, attachments, htmlBody})
    }
}
```

- Ahora puedo usar este método con la instancia del servicio en el server

```
const emailService = new EmailService()

emailService.sendemailWithFileSystemLogs(["bercast81@gmail.com"])
```

## Inyectar repositorio

- El mandar un correo electrónico es algo que también debería estar monitoreado
    - Debo poder comunicar si algo salió mal y que quede el registro
- Para ello voy a usar inyección de dependencias
- Primero lo haremos de la manera sencilla y luego la más racional
- Inyecto el LogRepository

```
import nodemailer from 'nodemailer'
import { envs } from '../../config/plugins/envs.plugin'
import { LogRepository } from '../../domain/repository/log.repository'
import { LogEntity, LogSeverityLevel } from '../../domain/entities/log.entity'

interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{

    constructor(private readonly logRepository: LogRepository){

    }
    private transporter= nodemailer.createTransport({
        service: envs.MAILER_SERVICE,
        auth:{
            user: envs.MAILER_EMAIL,
            pass: envs.MAILER_SECRET_KEY
        },
        tls: {
```

```
            rejectUnauthorized: false
        }
    })

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
                    attachments
                })

                console.log(sentInformation)

                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email sent',
                    origin: 'email.service'
                })
                this.logRepository.saveLog(log)

                return true
            } catch (error) {

                console.log(error)
                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email was no sent',
                    origin: 'email.service'
                })
                this.logRepository.saveLog(log)

                return false
            }
    }

    async sendemailWithFileSystemLogs(to: string | string[]){
            const subject= 'Logs del servidor'
            const htmlBody=`
            <h3>Logs del sistema</h3>
            <p>Desde sendEmailWithFileSystem</p>
            `

        const attachments: Attachment[]= [
            {filename: 'logs-all.log', path: './logs/logs-all.log'},
            {filename: 'logs-high.log', path: './logs/logs-high.log'},
            {filename: 'logs-medium.log', path: './logs/logs-medium.log'},
        ]
```

```
        return this.sendEmail({to, subject, attachments, htmlBody})
    }
}
```

- Inyecto el fileSystemLogRepository en la instancia del emailService en el server

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";


const fileSystemRepository = new LogRepositoryImpl(
    new FileSystemDatasource()
)


export class Server {

    public static start(){
        //CronService.createJob('*/5 * * * * *', ()=>{

          //  new CheckService(
            //    fileSystemRepository,
            //    ()=> console.log("Success!"),
            //    (error)=> console.log(`${error}`)
            //).execute('https://google.es')
        // })

        const emailService = new EmailService(fileSystemRepository)

        emailService.sendemailWithFileSystemLogs(["bercast81@gmail.com"])

    }
}
```

---

## SendEmail use-case

- Podría funcionar así tal cual, pero puedo crear un caso de uso
- Creo en domain/use-cases/logs/emails/send-email-logs.ts
- Usualmente son los casos de uso que llaman al repositorio
- Entonces debo inyectar el servicio y el repositorio
- domain/use-cases/emails/send-email.logs.ts

```

```

```
import { EmailService } from "../../../presentation/email/email.service"
import { LogEntity, LogSeverityLevel } from "../../entities/log.entity"
import { LogRepository } from "../../repository/log.repository"

interface SendLogEmailUseCase{
    execute: (to: string | string[])=> Promise<boolean>

}

export class SendEmailLogs implements SendLogEmailUseCase{

    constructor(
        private readonly emailService: EmailService,
        private readonly logRepository: LogRepository
    ){}

    async execute(to: string | string[]){

        try {

           const sent = await this.emailService.sendemailWithFileSystemLogs(to)
//regresa un boolean
           if(!sent){
            throw new Error('Email log not sent')
           }
           return true

        } catch (error) {
            const log = new LogEntity({
                message: `${error}`,
                level: LogSeverityLevel.medium,
                origin: 'send-email-logs'
            })

            this.logRepository.saveLog(log)
            return false
        }


        return true
    }
}
```

- Puedo mandar también un log conforme el mail se mandó exitosamente

```
 async execute(to: string | string[]){

        try {

           const sent = await this.emailService.sendemailWithFileSystemLogs(to)
//regresa un boolean
```

```
            if(!sent){
             throw new Error('Email log not sent')
            }

            const log = new LogEntity({
             message: `Log email sent`,
             level: LogSeverityLevel.medium,
             origin: 'send-email-logs'
        })

        this.logRepository.saveLog(log)
           return true

        } catch (error) {
            const log = new LogEntity({
                message: `${error}`,
                level: LogSeverityLevel.medium,
                origin: 'send-email-logs'
            })

            this.logRepository.saveLog(log)
            return false
        }


        return true
    }
```

- Ahora EmailService no necesita la inyección de dependencias

```
import nodemailer from 'nodemailer'
import { envs } from '../../config/plugins/envs.plugin'
import { LogRepository } from '../../domain/repository/log.repository'
import { LogEntity, LogSeverityLevel } from '../../domain/entities/log.entity'

interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{

    constructor(){  //borro la inyección de dependencias

    }
```

```
    private transporter= nodemailer.createTransport({
        service: envs.MAILER_SERVICE,
        auth:{
            user: envs.MAILER_EMAIL,
            pass: envs.MAILER_SECRET_KEY
        },
        tls: {
            rejectUnauthorized: false
        }
    })

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
                    attachments
                })

                console.log(sentInformation)

                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email sent',
                    origin: 'email.service'
                })
               // this.logRepository.saveLog(log)

                return true
            } catch (error) {

                console.log(error)
                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email was no sent',
                    origin: 'email.service'
                })
                //this.logRepository.saveLog(log)

                return false
            }
    }

    async sendemailWithFileSystemLogs(to: string | string[]){
            const subject= 'Logs del servidor'
            const htmlBody=`
            <h3>Logs del sistema</h3>
            <p>Desde sendEmailWithFileSystem</p>
            `
```

```
        const attachments: Attachment[]= [
            {filename: 'logs-all.log', path: './logs/logs-all.log'},
            {filename: 'logs-high.log', path: './logs/logs-high.log'},
            {filename: 'logs-medium.log', path: './logs/logs-medium.log'},
        ]


        return this.sendEmail({to, subject, attachments, htmlBody})
    }
}
```

- En el Server coloco la instancia del servicio fuera de la clase y llamo a mi caso de uso

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { SendEmailLogs } from "../domain/use-cases/emails/send-email-logs";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";


const fileSystemRepository = new LogRepositoryImpl(
    new FileSystemDatasource()
    )

const emailService = new EmailService()

export class Server {

    public static start(){
        //CronService.createJob('*/5 * * * * *', ()=>{

          //  new CheckService(
            //    fileSystemRepository,
            //    ()=> console.log("Success!"),
            //    (error)=> console.log(`${error}`)
            //).execute('https://google.es')
        // })


        new SendEmailLogs(emailService,
fileSystemRepository).execute(['bercast81@gmail.com'])

    }
}
```

# NOC - Mongo y PostgreSQL

- Conexion con mongo
- Creo la interfaz para evitar dependencias
- data/mongo/init.ts

```
import mongoose from "mongoose";


interface ConnectionOptions{
    mongoUrl: string
    dbName: string
}

export class MongoDatabase{


    static async connect(options:ConnectionOptions){
        const {mongoUrl, dbName} = options

        try {

            return await mongoose.connect(mongoUrl,{
                dbName   //en este objeto se pueden configurar otras cosas
            })

        } catch (error) {
            console.log('Mongo connection error')
            throw error
        }
    }
}
```

- Para conectarme a mongo, en app, antes del server.
- Añado las variables de entorno en el plugin. En .env defino las variables de entorno MONGO_STRING y MONGO_DB
- config/plugins/envs.plugin.ts

```
 MONGO_STRING: env.get('MONGO_STRING').required().asString(),
 MONGO_DB: env.get('MONGO_DB').required().asString()
```

- En app.ts

```
import { envs } from "./config/plugins/envs.plugin"
import { MongoDatabase } from "./data/mongo"
import { Server } from "./presentation/server"

(async ()=>{
```

```
    main()
})()

async function main(){

    await MongoDatabase.connect(
        {
            mongoUrl:envs.MONGO_STRING, dbName: envs.MONGO_DB
        })
    Server.start()
    console.log(envs.PORT)
}
```

---

## Schema & Models

- Creo mongo/models/log.model.ts
- Debo tener la data que se muestra en la interface LogEntity
- domain/entities/log.entity.ts

```
export interface LogEntityOptions{
     level: LogSeverityLevel
     message: string
     createdAt?: Date
     origin: string
}
```

- Creemos el modelo
- data/mongo/models/log.model.ts

```
import mongoose from "mongoose";
import { LogSeverityLevel } from "../../../domain/entities/log.entity";


const LogSchema = new mongoose.Schema({


    level: {
        type: String,
        enum: ['low', 'medium', 'high'],
        default: 'low'
    },

    message: {
        type: String,
        required: true
    },

    createdAt: {
```

```
        type: Date,
        default: new Date()
    },

    origin: {
        type: String,

    },
})
export const LogModel = mongoose.model('Log', LogSchema) //por defecto mongoose le
añadirá la s de plural a Log
```

---

## Crear y leer de mongo

- Creo un registro en el app.ts

```
import { envs } from "./config/plugins/envs.plugin"
import { LogModel, MongoDatabase } from "./data/mongo"
import { Server } from "./presentation/server"

(async ()=>{
    main()
})()

async function main(){

    await MongoDatabase.connect(
        {
            mongoUrl:envs.MONGO_STRING, dbName: envs.MONGO_DB
        })

        //después de este ejemplo borraré este código y dejaré solo la conexión
        const newLog = await LogModel.create({
            message: "test message mongoose",
            origin: "app.ts",
            level: 'low'
        })

        await newLog.save()

        const logs = await LogModel.find()

        console.log({
            newLog,
            logs
        })



    Server.start()
```

```
}
```

- Debemos construir un datasource que implemente la clase abstracta domain/datasources/LogDatasource
- En domain establezco las normas

```
import { LogEntity, LogSeverityLevel } from "../entities/log.entity";

export abstract class LogDataSource{

    //cualquier origen de datos va a tener que implementar saveLog
    abstract saveLog(log: LogEntity): Promise<void>;
    abstract getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>
}
```

---

## MongoLogDatasource

- Creo en infraestructure/datasources/mongo-log.datasouce.ts
- Implemento los dos métodos usando el modelo

```
import { LogModel } from "../../data/mongo";
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";

 export class MongoLogDataSource implements LogDataSource {

    async saveLog(log:LogEntity): Promise<void>{
        const newLog = await LogModel.create(log) //este log no es una instancia
de nuestra entidad
                                                  //es una instancia del modelo de
mongoose
        newLog.save()
        return

    }
    async getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>{
        const logs= await LogModel.find({level:severityLevel})
        return logs //esto marca error porque devuelve logs pero no son instancias
de LogEntity
    }

 }
```

- Creo un método en domain/datasource/LogEntity para crear entidades de un JSON
-

```
static fromObject=(object: {[key:string]:any}):LogEntity=>{
        const {message, level, createdAt, origin} = object

        if(!message) throw new Error("¡Falta el mensaje!")

            const log = new LogEntity({
            message,
            level,
            createdAt,
            origin
        })

        return log
     }
```

- Ahora puedo usarlo sin instanciar la clase (porque es static) y devolver un arreglo de entidades

```
import { log } from "console";
import { LogModel } from "../../data/mongo";
import { LogDataSource } from "../../domain/datasources/log.datasource";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";

 export class MongoLogDataSource implements LogDataSource {

    async saveLog(log:LogEntity): Promise<void>{
        const newLog = await LogModel.create(log) //este log no es una instancia
de nuestra entidad
                                                  //es una instancia del modelo de
mongoose
        newLog.save()
        return

    }
    async getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>{
        const logs= await LogModel.find({level:severityLevel})

        return logs.map(log=> LogEntity.fromObject(log))
    }

 }
```

---

## Grabar logs en Mongo

- Inicio el servidor en app.ts con Server.start()
- Cambio el nombre del fileSystemRepository a logRepository en server.ts

```
//cambio el nombre a logRepository
const fileSystemRepository = new LogRepositoryImpl(
    new FileSystemDatasource()
    )

const emailService = new EmailService()
```

- De esta manera puedo añadirle la nueva instancia de mongo sin alterar el resto del código, ya que le paso el repositorio al caso de uso del envio de mails y al servcio de check más abajo. Con un nombre más genérico es más adecuado

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { SendEmailLogs } from "../domain/use-cases/emails/send-email-logs";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { MongoLogDataSource } from "../infraestructure/datasources/mongo-
log.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";

//cambio el nombre a logRepository
const logRepository = new LogRepositoryImpl(
    //new FileSystemDatasource()

    new MongoLogDataSource()
    )

const emailService = new EmailService()

export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

          new CheckService(
          logRepository,
              ()=> console.log("Success!"),
              (error)=> console.log(`${error}`)
            ).execute('https://google.es')
    })


        new SendEmailLogs(emailService,
logRepository).execute(['ismaelberoncastano@gmail.com'])

    }
}
```

-

---

## PostgreSQL

- Usaremos TypeORM y Docker
- En la raiz del proyecto: docker-compose.yaml

```
version: '3.8'
services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - ./postgres:/var/lib/postgresql/data
    ports:
      - 5432:5432
```

- Coloco las variables de entorno en .env
- potgres es el usuario por defecto

```
POSTGRES_USER=potgres
POSTGRES_PASSWORD=123456
POSTGRES_DB=NOC
POSTGRES_URL=postgresql://potgres:123456@localhost:5432/NOC
```

- Para probar la instalacion usaré tablePlus
    - Name: NOC-PostgreSQL
    - Host: localhost
    - Port: 5432
    - User: postgres
    - Password: 123456
    - Database: NOC
- Para correr la db uso ****
- Usaremos Prisma - ORM

> npm i prisma npx prisma init npx prisma init --datasource-provider PostgreSQL

- Hay que modelar el schema
- Coloco la variable de entorno del string de conexion en el schema.prisma
- Consulto la documentación

- Creo el modelo de Log siguiendo la log.entity
- Prisma recomienda hacer los enums en mayúsculas

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("POSTGRES_URL")
}

enum SeverityLevel{
  LOW
  MEDIUM
  HIGH
}

model LogModel {
  id      Int @id @default(autoincrement())
  message String
  origin  String
  level   SeverityLevel
  createdAt DateTime   @default(now())
}
```

- Haciendo mi modelo simularía que no tengo la DB creada
- Si ya tuvieramos la DB creada podriamos hacer el npx primsa pull para crear todos los objetos
- Como no tenemos Db haremos la migración

> npx prisma migrate dev --name init

- Con la migración se crea el Prisma Client, que es lo que necesitamos para trabajar con la DB
- Si algo sale mal se pueden rvertir las migraciones
- En TablePlus puedo ver la migracion y LogModel

---

## Inserción y lectura

- Probemos en app.ts

```
import { PrismaClient } from "@prisma/client"
import { envs } from "./config/plugins/envs.plugin"
import { LogModel, MongoDatabase } from "./data/mongo"
import { Server } from "./presentation/server"

(async ()=>{
    main()
})()
```

```
async function main(){

    await MongoDatabase.connect(
        {
            mongoUrl:envs.MONGO_STRING, dbName: envs.MONGO_DB
        })

        const prisma = new PrismaClient()

        const newLog = await prisma.logModel.create({
            data:{
            message:"test message",
            origin:"app.ts postgres",
            level:"LOW"}
        })

        const logs = await prisma.logModel.findMany({})


        const logsLOW = await prisma.logModel.findMany({
            where:{
                level: "LOW"
            }
        })
        console.log({
            logs,
            logsLOW
        })
    //Server.start()

}
```

- Pongo en marcha el servidor para que lo inserte en la db

---

## PotgresLogDatasource

- El severityLevel creado en prisma no es compatible con lo que definimos en LogEntity
- Hay que hacer una conversión
- En getLogs si devuelvo dbLogs en un return tal cual, regresa instancias pero no entidades (aunque no está muy lejos de ello)
- Uso el metodo estático de LogEntity fromoObject

```
import { LogModel, PrismaClient, SeverityLevel } from "@prisma/client";
import { LogEntity, LogSeverityLevel } from "../../domain/entities/log.entity";

const prisma = new PrismaClient()

const severityEnum = {
    low: SeverityLevel.LOW,
```

```
    medium: SeverityLevel.MEDIUM,
    high: SeverityLevel.HIGH,
}

export class PostgresLogDataSource{


     async saveLog(log: LogEntity): Promise<void>{

         const level = severityEnum[log.level]

         const newLog = await prisma.logModel.create({

            data:{
                ...log,
                level
            }
        })


     }

     async getLogs(severityLevel: LogSeverityLevel): Promise<LogEntity[]>{
        const level = severityEnum[severityLevel]

        const dbLogs = await prisma.logModel.findMany({
            where:{
                level
            }
        })

        return dbLogs.map(dbLog => LogEntity.fromObject(dbLog))
     }
}
```

- Debo crear una nueva instancia de logRepository en el server

```
import { CheckService } from "../domain/use-cases/checks/check-service";
import { SendEmailLogs } from "../domain/use-cases/emails/send-email-logs";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { MongoLogDataSource } from "../infraestructure/datasources/mongo-
log.datasource";
import { PostgresLogDataSource } from "../infraestructure/datasources/postgres-
log.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";

//cambio el nombre a logRepository
const logRepository = new LogRepositoryImpl(
    //new FileSystemDatasource()
```

```
    //new MongoLogDataSource()

    new PostgresLogDataSource()

)

const emailService = new EmailService()

export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

          new CheckService(
          logRepository,
              ()=> console.log("Success!"),
              (error)=> console.log(`${error}`)
            ).execute('https://google.es')
    })


        new SendEmailLogs(emailService,
logRepository).execute(['ismaelberoncastano@gmail.com'])

    }
}
```

- En app.ts descomento Server.start y pongo en marcha el servidor

---

## Grabar en Mongo, Postgres y FS simultáneamente

- Hagamos un caso de uso en el que usemos todos los repositorios
- domain/use-cases/checks/check-service-multiple.ts
- Lo unico diferente con CheckServiceUseCase es que voy a recibir un arreglo de repositorios
- Creo un metodo privado callLogs para barrer el arreglo de repositorios y guardar en todos ellos

```
import { LogEntity, LogSeverityLevel } from "../../entities/log.entity"
import { LogRepository } from "../../repository/log.repository"

interface CheckServiceMultipleUseCase{
    execute(url: string):Promise <boolean>
}

type SuccessCallback = ()=> void  //tipo de lo que quiero ejecutar si todo sale
bien
type ErrorCallback = (error: string)=> void //tipo si hay algún error

export class CheckServiceMultiple implements CheckServiceMultipleUseCase{
```

```
    constructor(
        private readonly logRepository: LogRepository[],
        private readonly successCallback: SuccessCallback,
        private readonly errorCallback: ErrorCallback
        ){}


        async callLogs(log:LogEntity){
            this.logRepository.forEach((logRepository)=>{
                logRepository.saveLog(log)
            })
        }

    async execute(url: string): Promise <boolean>{

        try {
            const req = await fetch(url)

            if(!req.ok){
                throw new Error(`Error on check service ${url}`)
            }

            //Si ha ido bien puedo guardar el log con LogRepository

            const log = new LogEntity({
                message:`Service ${url} working`,
            level: LogSeverityLevel.low,
        origin: 'check-service.ts' })

        //this.logRepository.saveLog(log)
            this.callLogs(log)
            this.successCallback() //llamo al SuccessCallback si todo sale bien


            return true

        } catch (error) {

            const errorMessage = `${error}`
            const log = new LogEntity({  //debo pasarle el objeto a la instancia
de LogEntity
                message:errorMessage,
                level: LogSeverityLevel.low,
                origin: ' check.service.ts' })

            //this.logRepository.saveLog(log)
            this.callLogs(log)
            this.errorCallback(errorMessage) //llamo al ErrorCallback

            return false
        }
```

```
    }
}
```

- En server.ts necesito mandar un arreglo con todos los repositorios

```
import { CheckServiceMultiple } from "../domain/use-cases/checks/check-service-
multiple";
import { SendEmailLogs } from "../domain/use-cases/emails/send-email-logs";
import { FileSystemDatasource } from "../infraestructure/datasources/file-
system.datasource";
import { MongoLogDataSource } from "../infraestructure/datasources/mongo-
log.datasource";
import { PostgresLogDataSource } from "../infraestructure/datasources/postgres-
log.datasource";
import { LogRepositoryImpl } from "../infraestructure/repository/log.repository";
import { CronService } from "./cron/cron-service";
import { EmailService } from "./email/email.service";

//cambio el nombre a logRepository
const fsLogRepository = new LogRepositoryImpl(
    new FileSystemDatasource()

)

const mongoLogRepository = new LogRepositoryImpl(
    new MongoLogDataSource()

)

const postgresLogRepository = new LogRepositoryImpl(
    new PostgresLogDataSource()

)

const arrayRepos= [fsLogRepository, mongoLogRepository, postgresLogRepository]


new PostgresLogDataSource()

const emailService = new EmailService()

export class Server {

    public static start(){
        CronService.createJob('*/5 * * * * *', ()=>{

          new CheckServiceMultiple(
           arrayRepos,
              ()=> console.log("Success!"),
              (error)=> console.log(`${error}`)
            ).execute('https://google.es')
    })
```

```
        new
SendEmailLogs(emailService,arrayRepos[0]).execute(['ismaelberoncastano@gmail.com']
)
        new
SendEmailLogs(emailService,arrayRepos[1]).execute(['ismaelberoncastano@gmail.com']
)
        new
SendEmailLogs(emailService,arrayRepos[2]).execute(['ismaelberoncastano@gmail.com']
)

    }
}
```

---

# NODE_TS REST SERVER

> npm init -y npm i -D typescript @types/node ts-node-dev rimraf npx tsc --init --outDir dist/ --rootDir src

- Crear scripts

```
"dev": "tsnd --respawn --clear src/app.ts",
"build": "rimraf ./dist && tsc",
"start": "npm run build && node dist/app.js",
```

- Creo src/app.ts

---

## Creemos un Web Server

- Importo http en app.ts

```
import http from 'http'

const server = http.createServer((req,res)=>{
    console.log(req.url)
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

- Si me conecto al localhost 8080 desde el navegador, cualquier dirección a la que vaya se verá en la consola

```
import http from 'http'

const server = http.createServer((req,res)=>{
    console.log(req.url)
    res.write("hola mundo!") //imprime en pantalla hola mundo!
    res.end()
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

---

## Diferentes respuestas

- Puedes escribit el codigo de respuesta con res.writeHead()

```
res.writeHead(404)
```

- Esto sería server side rendering

```
import http from 'http'

const server = http.createServer((req,res)=>{
    console.log(req.url)

    res.write("hola mundo!")
    res.writeHead(200, {'Content-Type': 'text/html'})
    res.write(`<h1>URL ${req.url}</h1>`)
    res.end()
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

- Puedo renderizar JSON

```
import http from 'http'

const server = http.createServer((req,res)=>{
```

```
    console.log(req.url)

    // res.write("hola mundo!")
    // res.writeHead(200, {'Content-Type': 'text/html'})
    // res.write(`<h1>URL ${req.url}</h1>`)

    const data = {name: "Mili Vanili", age: 25}
    res.writeHead(200, {'Content-Type': 'application/json'})
    res.end(JSON.stringify(data))
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

- Creo la carpeta public/index.html
- Psara renderizar el index en /

```
import http from 'http'
import fs from 'fs'

const server = http.createServer((req,res)=>{

    if(req.url === '/'){
        const htmlFile= fs.readFileSync('./public/index.html', 'utf-8')
        res.writeHead(200, {'Content-Type': 'text/html'})
        res.end(htmlFile)
    }else{
        res.writeHead(404)
    }
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

- Si coloco una hoja de estilos CSS y un script de JS en el HTML el servidor no los encuentra **porque no los estoy sirviendo**

---

## Responder demás archivos

- Para servir lso archivos en / (tengo index.css e index.js en la carpeta public)

```
import http from 'http'
import fs from 'fs'
```

```
const server = http.createServer((req,res)=>{

    if(req.url === '/'){
        const htmlFile= fs.readFileSync('./public/index.html', 'utf-8')
        res.writeHead(200, {'Content-Type': 'text/html'})
        res.end(htmlFile)
        return
    }

    if(req.url?.endsWith('.js')){
        res.writeHead(200, {'Content-Type': 'application/javascript'})
    }else if(req.url?.endsWith('.css')){
        res.writeHead(200, {'Content-Type': 'text/css'})
    }

    const responseContent= fs.readFileSync(`./public${req.url}`, 'utf-8') //habria
que comprobar si existe el archivo
    res.end(responseContent)
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

- Esta manera de trabajar es muy tediosa.
- Con **Express** es más sencillo

---

## Http2 OpenSSL

- Clono app.ts y a la copia le pongo app.http.ts
- Renombro http por http2
- Solo con esto no va a funcionar
- No hay navegadores que soporten http2 con servidores no seguros
- Hay que usar .createSecureServer (con las opciones key y certificado) y configurar https
- Para generar el key y el certificado en windows
    - Powershell > openssl (no lo reconoce)
- Si esta instalado Git esta instalado pero hay que actualizar las variables de entorno
- Buscar en Windows env
- Editaremos Path
- Buscar la ruta Program Files/ Git/ usr /bin / openssl
- Copiar ruta en las variables de entorno, en Path
- Cerrar y volver a abrir la powershell
- Ahora usar este comando

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout server.key -out server.crt
```

- Rellenar la info
- Tengo la key y el certificado server.crt y server.key, los m,eto dentro de la carpeta keys en la raiz

- Ahora hay que añadirlos a las opciones de createSecureServer
- Si intento conectarme a localhost:8080 me dice que la conexion no es segura porque el certificado no es valido
- Advanced/proceed
- Si da error con favicon.ico, créalo o usa un try catch

```
import http2 from 'http2'
import fs from 'fs'

const server = http2.createSecureServer({
    key:fs.readFileSync('./keys/server.key'),
    cert:fs.readFileSync('./keys/server.crt')
}, (req,res)=>{

    if(req.url === '/'){
        const htmlFile= fs.readFileSync('./public/index.html', 'utf-8')
        res.writeHead(200, {'Content-Type': 'text/html'})
        res.end(htmlFile)
        return
    }

    if(req.url?.endsWith('.js')){
        res.writeHead(200, {'Content-Type': 'application/javascript'})
    }else if(req.url?.endsWith('.css')){
        res.writeHead(200, {'Content-Type': 'text/css'})
    }

    const responseContent= fs.readFileSync(`./public${req.url}`, 'utf-8')
    res.end(responseContent)
})


server.listen(8080, ()=>{
    console.log('Server is listening on port 8080')
})
```

---

## Usemos Express

- Trata de que los types de express y la version de express sean lo más cercanas posibles (mirar .json)
- Creo un nuevo app.ts
- En una función autoinvocada llamo al metodo main que creo fuera de la función

```
(async()=>{

    main();
})()
```

```
function main (){

}
```

- En presentation creo la clase Server
- Lo crearemos sin hacer estático el método start

```
import express from 'express'


export class Server {

    private app = express()

    async start(){
        this.app.listen(3000, ()=>{
            console.log("corriendo en el 3000!")
        })
    }
}
```

- Creo una instancia del Server en app y llamo a start

```
import { Server } from "./presentation/server";

(async()=>{

    main();
})()


function main (){

    const server = new Server()

    server.start()

}
```

- Creo un middleware para mostrar lo que haya en mi carpeta pública
- Los middlewares son funciones que se ejecutan al pasar por una ruta
- Tanto el puerto como la ruta serán variables, dependencias que deberiamos recibir al crear el servidor

```
import express from 'express'


export class Server {
```

```
    private app = express()

    async start(){


        //middlewares

        //Public Folder
        this.app.use(express.static('public'))

        this.app.listen(3000, ()=>{
            console.log("corriendo en el 3000!")
        })
    }
}
```

---

## Servir SPA con Router

- Usaremos un proyecto de React ya hecho
- Con el paquete http-server y el comando http-server -o levanto la app de Raect
- Sirve para hacer pruebas
- Ahora lo que ocurre es que si yo busco por /marvel y recargo no accede porque no tengo la carpeta marvel en mi carpeta public pero si está en el enrutamiento de React
-

```
import express from 'express'


export class Server {

    private app = express()

    async start(){


        //middlewares

        //Public Folder
        this.app.use(express.static('public'))

        this.app.get('*', (req,res)=>{
          res.send('Hola Mundo')
        })

        this.app.listen(3000, ()=>{
            console.log("corriendo en el 3000!")
        })
    }
}
```

- Si recargo voy a la aplicacion, pero si busco la ruta me devuelve el hola mundo
- Esto pasa porque cuando levanto el server lo encuentra y lo sirve porque React toma el control de la aplicación
- Cuando recargo en una ruta que no es el index, cae al hola mundo porque no encuentra las carpetas /marvel, etc

```
import express from 'express'
import path from 'path'


export class Server {

    private app = express()

    async start(){


        //middlewares

        //Public Folder
        this.app.use(express.static('public'))

        this.app.get('*', (req,res)=>{
          const indexPage = path.join(__dirname, '../../public/index.html')
          res.sendFile(indexPage)
        })

        this.app.listen(3000, ()=>{
            console.log("corriendo en el 3000!")
        })
    }
}
```

## Variables de entorno

- .env

```
PORT=3000
PUBLIC_PATH=public
```

- Instalo

> npm i dotenv env-var

- En src/config/env.ts

```
import 'dotenv/config'
import {get} from 'env-var'

export const envs = {
    PORT: get('PORT').required().asPortNumber(),
    PUBLIC_PATH: get('PUBLIC_PATH').required().default('public').asString()

}
```

- En el server creo una interfaz Options donde pasar el puerto y el path
- El path lo pongo opcional y le asigno uno por defecto
- Podemos declararlas como readonly y asignarles el valor en el constructor
- Luego ya no podremos cambiar ese valor (por ser readonly, pero si es permitido en el constructor)

```
import express from 'express'
import path from 'path'


interface Options{
    PORT : number
    PUBLIC_PATH?: string
}


export class Server {
    private readonly port;
    private readonly publicPath;

    constructor(options:Options){
        const {PORT, PUBLIC_PATH='public'}= options

        this.port = PORT
        this.publicPath = PUBLIC_PATH
    }

    private app = express()

    async start(){


        //middlewares

        //Public Folder
        this.app.use(express.static(this.publicPath))

        this.app.get('*', (req,res)=>{
          const indexPage = path.join(__dirname,
`../../${this.publicPath}/index.html`)
          res.sendFile(indexPage)
        })
```

```
        this.app.listen(this.port, ()=>{
            console.log("corriendo en el 3000!")
        })
    }
}
```

- Ahora debo agregarle las variables a la instancia del server

---

# REST SERVER POSTGRESQL

- Usaremos TMDB
- Creo la conexion con la db usando docker
- .env

```
PORT=3000
PUBLIC_PATH=public

POSTGRES_URL=postgresql://postgres:root@localhost:5432/TODO
POSTGRES_USER=postgres
POSTGRES_DB=TODO
POSTGRES_PORT=5432
POSTGRES_PASSSWORD=123456

NODE_ENV=development
```

- En la raiz del proyecto creo docker-compose.yaml

```
version: '3.8'
services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - ./data:/var/lib/postgresql/data
    ports:
      - 5432:5432
```

- Ignoramos postgres/ en gitignore
- Levantamos la db. La -d es de detouch para despegar la consola de todos los logs

```
docker compose up -d
```

- Instalo prisma. Genero el archiuvo de configuración

```
npx prisma init --datasource-provider postgresql
```

- Cambio el datasource por la variable de entorno
- Defino el modelo

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("POSTGRES_URL")
}


model todo{
  id Int @id @default(autoincrement())
  text String @db.VarChar
  completedAt DateTime? @db.Timestamp()
}
```

- Hagamos la migración

```
npx prisma migrate dev --name init
```

- *Your database is now in sync with your schema*
- Creo la conexión en TablePlus

---

# Crear TODO

- Creo src/data/postgres/index.ts (barril)

```
import { PrismaClient } from "@prisma/client";

export const prisma = new PrismaClient()
```

- Primero vamos a hacer todo el código, validaciones, llamados a la db, en los controladores
- Luego mediante refactorización emplearemos arquitectura limpia
- En createTodo llamo a prisma
- Más adelante usaremos DTOs para validar la data de entrada

```
public async createTodo(req:Request, res:Response){
    const {text} = req.body

    if(!text) return res.status(400).json({error: "Text is required"})

        const newTodo= await prisma.todo.create({
        data: {
            text
        }

    })

    res.json(newTodo)

}
```

- Todo es lo mismo: prisma.todo.update, .delete, .find, .findMany, etc
- Haz la tarea si quieres!

---

## DTOs

- Validemos la data de entrada para createTodo
- Podemos usar un middleware con express-validator
- Pero lo haremos de otra manera
- Creo domain/dtos/todos/todo.dto.ts (en domain porque son reglas)
- Creo un constructor privado. Solo se va a poder llamar dentro de un metodo estático de la clase
- Creo un metodo estático, las properties simularán el objeto del req.body
- Retornará o un string o una instancia del dto
- Lo pongo dentro de un arreglo para usar desestructuración.
- Los hago los dos opcionales porque si tengo el string es que tengo un error y si no tengo la instancia del DTO

```
export class createTodoDto {

    private constructor(
        public readonly text: string
    ){
        this.text = text
    }

    static create(props: {[keys: string]: any}): [string?, createTodoDto?] {
        const {text} = props

        if(!text) return ["Text is required", undefined]



        return [undefined, new createTodoDto(text)]
```

```
    }

}
```

```
    public async createTodo(req:Request, res:Response){

        const createTodoDto= CreateTodoDto.create(req.body)

        if(createTodoDto[0]) return res.status(400).json({error:
createTodoDto[0]})

        const {text} = req.body

        if(!text) return res.status(400).json({error: "Text is required"})

            const newTodo= await prisma.todo.create({
            data: createTodoDto!

        })

        res.json(newTodo)

    }
```

- Otra manera sería usando desestructuración

```
    public async createTodo(req:Request, res:Response){

        //const createTodoDto= CreateTodoDto.create(req.body)
        const [error,createTodoDto] = CreateTodoDto.create(req.body)

        if(error) return res.status(400).json({error})

        const {text} = req.body

        if(!text) return res.status(400).json({error: "Text is required"})

            const newTodo= await prisma.todo.create({
            data: createTodoDto!

        })

        res.json(newTodo)

    }
```

---

## UpdateDto

- En el update haré el texto y la fecha opcionales
- Puede pasarme las dos (texto y fecha), una de las dos o ninguna de las dos
- Debo asegurarme que el texto sea un string y la fecha una fecha
- Como los dos valores son opcionales, creo un getter para obtener el objeto para el dto

```
get values(){
const returnobj:{[key:string]: any} = {}
if(this.text) returnobj.text = this.text
if(this.completedAt) returnobj.completedAt = this.completedAt
return returnobj
}
```

- El dto
- Le paso al constructor los valores opcionales que puede tener el req.body. El id es obligatorio
- Creo un getter que retorna el objeto que asemeja el del req.body para obtener los valopres desde el controlador
- En el metodo create, o retorno el string de error o el dto

```
export class UpdateTodoDto {

    private constructor(
      public readonly id: number,
      public readonly text?: string,
      public readonly completedAt?: Date,
    ){}

    get values() {
      const returnObj: {[key: string]: any} = {};

      if ( this.text ) returnObj.text = this.text;
      if ( this.completedAt ) returnObj.completedAt = this.completedAt;

      return returnObj;
    }


    static create( props: {[key:string]: any} ): [string?, UpdateTodoDto?]  {

      const { id, text, completedAt } = props;
      let newCompletedAt =completedAt; // si viene completedAt la guardo aqui

      if ( !id || isNaN( Number(id)) ) {
        return ['id must be a valid number']; //si da error retornará este string
      }

      if ( completedAt ) {
        newCompletedAt = new Date( completedAt) //la formateo
        //la valido
        if ( newCompletedAt.toString() === 'Invalid Date' ) {
```

```
            return ['CompletedAt must be a valid date']
          }
        }

        return [undefined, new UpdateTodoDto(id, text, newCompletedAt)]; //si no da
error retornará la instancia con los nuevos valores
    }


  }
```

- En el controlador

```
public updateTodo = async( req: Request, res: Response ) => {
    const id = +req.params.id; //casteo el id de la url

    const [error, updateTodoDto] = UpdateTodoDto.create({...req.body, id}); //creo
el dto y extraigo el valor de retorno
    if ( error ) return res.status(400).json({ error }); //si hay error devuelvo
el error

    //busco el todo
    const todo = await prisma.todo.findFirst({
        where: { id }
    });

    if ( !todo ) return res.status( 404 ).json( { error: `Todo with id ${ id } not
found` } );

    const updatedTodo = await prisma.todo.update({
        where: { id },
        data: updateTodoDto!.values //le paso los valores con el getter
    });

    res.json( updatedTodo );

    }
```

- Recuerda agregar la ruta!
- todos/routes.ts

```
export class TodoRoutes{


    static get routes():Router{
        const router = Router();
        const todosController = new TodosController();

        router.get('/', todosController.getTodos) //solo mandamos la referencia a
```

```
la función
        router.get('/:id', todosController.todoById) //solo mandamos la referencia
a la función
        router.post('/', todosController.createTodo)
        router.put('/:id', todosController.updateTodo)
        return router
    }
}
```

---

## Aprovisionar DB

- En railway puedo subir mi DB. Me entregará una cadena de conexión
- Para desplegar mi db cambio el string de conexión por el proporcionado
- Creo un nuevo scripot en el package.json

> "prisma:migrate:prod":"prisma migrate deploy"

- Lo incluyo en el build

> "build":"rimraf ./dist && tsc && npm run prisma:migrate:prod"

---

# REST CLEAN ARCHITECTURE

- Aplicaremos DDD y el patrón repositorio

---

## TodoEntity

- Creo domain/entities/todo.entity.ts

```
export class TodoEntity{

    constructor(
        public id: number,
        public text: string,
        public completedAt?: Date | null
    ){}

    get isCompleted(){
        return !!this.completedAt
    }
}
```

---

## DataSource

- Creo domain/datasources/todo.datasource.ts y domain/repositories

- Creo las normas que regirán en ms datasources

```
import { CreateTodoDto } from "../dtos/todos/todo.dto";
import { UpdateTodoDto } from "../dtos/todos/update.dto";
import { TodoEntity } from "../entities/todo.entity";

export abstract class TodoDataSource{
    abstract create(createTodoDto: CreateTodoDto): Promise<TodoEntity>

    //todo:paginación
    abstract getAll(): Promise<TodoEntity[]>

    abstract findById(id: number): Promise<TodoEntity | null>

    abstract updateById(updateTodoDto: UpdateTodoDto): Promise<TodoEntity | null>

    abstract deleteById(id: number): Promise<TodoEntity | null>
}
```

- Vamos a tener que añadir algunas modificaciones cuando agreguemos paginaciones
- El repositorio es la misma implementación ya que voy a usar los mismos métodos

```
import { CreateTodoDto } from "../dtos/todos/todo.dto";
import { UpdateTodoDto } from "../dtos/todos/update.dto";
import { TodoEntity } from "../entities/todo.entity";

export abstract class TodoRepository{
    abstract create(createTodoDto: CreateTodoDto): Promise<TodoEntity>

    //todo:paginación
    abstract getAll(): Promise<TodoEntity[]>

    abstract findById(id: number): Promise<TodoEntity | null>

    abstract updateById(updateTodoDto: UpdateTodoDto): Promise<TodoEntity | null>

    abstract deleteById(id: number): Promise<TodoEntity | null>
}
```

- En otro panorama, podría crearme un servicio en el que añadir toda mi lógica.
- La idea es tener un solo logar para poder acceder a toda la info, para que si eso cambia, cambiar el servicio y listo
- Sigamos con el DDD y los casos de uso

---

## TodoDataSource implementation

- Las implementaciones las haré en infraestructure, en este caso /datasource

- Escribo export class ... implements TodoDataSource y Ctrl . para que autoomplete
- Copio el código de cada método correspondiente que hay en el controlador y lo traslado a los métodos del datasource implementation
- *NOTA*: esto es una refactorización a arquitectura limpia

```
async getAll(): Promise<TodoEntity[]> {
    const todos= await prisma.todo.findMany()

    return todos
}
```

- Esto me marca **ERROR** porque estoy retornando un objeto, no instancias de mi clase que tienen un metodo getter
- Para solucionarlo necesito crear un **mapper** que mapée los objetos y los convierta a instancias de mi clase
- Puedo crear el mapper en un archivo aparte o puedo crear un método en TodoEntity

```
import { CreateTodoDto } from "../dtos/todos/todo.dto";

export class TodoEntity{

    constructor(
        public id: number,
        public text: string,
        public completedAt?: Date | null
    ){}

    get isCompleted(){
        return !!this.completedAt
    }

    public static formJson(object: {[key:string]: any}): TodoEntity{
        const {id, text, completedAt} = object

        if(!id){
            throw new Error('Id is required')
        }

        if(!text){
            throw new Error('Text is required')
        }

        let newCompletedAt;

        if(completedAt){
            newCompletedAt = new Date(completedAt)
        }

        if(isNaN(newCompletedAt.getTime())){
```

```
            throw new Error('Invalid date')
        }

       return new TodoEntity(id, text, newCompletedAt)
    }
}
```

- En infraestructure/datasource/todo.datasource.impl.ts

```
async getAll(): Promise<TodoEntity[]> {
    const todos= await prisma.todo.findMany()

    return todos.map(todo=> TodoEntity.formJson(todo))
}
```

- En infraestructure/repositories/TodoRepositoryImpl
- Le inyecto en el constructor el datasource (no la implementación del datasource)

```
export class TodoRepositoryImpl implements TodoDataSource{

    constructor(
        private readonly datasource: TodoDataSource
    ){}


    create(createTodoDto: CreateTodoDto): Promise<TodoEntity> {
        return this.datasource.create(createTodoDto); //lo mismo con el resto de
métodos
    }
}
```

- La implementeación del repositorio es parecida a lo que había en el controlador

```
import { prisma } from "../../data/postgres";
import { TodoDataSource } from "../../domain/datasources/todo.datasources";
import { CreateTodoDto } from "../../domain/dtos/todos/todo.dto";
import { UpdateTodoDto } from "../../domain/dtos/todos/update.dto";
import { TodoEntity } from "../../domain/entities/todo.entity";
import { TodoRepository } from "../../domain/repositories/todo.repo";


export class TodoRepositoryImpl implements TodoRepository{

    constructor(
        private readonly todoDatasource: TodoDataSource
    ){}
```

```
    create(createTodoDto: CreateTodoDto): Promise<TodoEntity> {
        return this.todoDatasource.create(createTodoDto);
    }

    deleteById(id: number): Promise<TodoEntity> {
        return this.todoDatasource.deleteById(id)
    }

    async getAll(): Promise<TodoEntity[]> {
        return this.todoDatasource.getAll()
    }

    findById(id: number): Promise<TodoEntity> {
        return this.todoDatasource.findById(id)
    }

    updateById(updateTodoDto: UpdateTodoDto): Promise<TodoEntity> {
        return this.todoDatasource.updateById(updateTodoDto)
    }
}
```

- En infraestructure/datasource/TodoDatasourceImpl

```
import { prisma } from "../../data/postgres";
import { TodoDataSource } from "../../domain/datasources/todo.datasources";
import { CreateTodoDto } from "../../domain/dtos/todos/todo.dto";
import { UpdateTodoDto } from "../../domain/dtos/todos/update.dto";
import { TodoEntity } from "../../domain/entities/todo.entity";

export class TodoDatasourceImpl implements TodoDataSource{

    async create(createTodoDto: CreateTodoDto): Promise<TodoEntity> {
        const todo = await prisma.todo.create({
            data: createTodoDto
        })

        return  TodoEntity.formJson(todo)

    }

    async deleteById(id: number): Promise<TodoEntity> {
         await this.findById(id)
        const deleted = await prisma.todo.delete({
            where: {id}
        })

        return TodoEntity.formJson(deleted)
    }

    async getAll(): Promise<TodoEntity[]> {
        const todos= await prisma.todo.findMany()
```

```
        return todos.map(todo=> TodoEntity.formJson(todo))
    }

    async findById(id: number): Promise<TodoEntity> {
       const todo = await prisma.todo.findFirst({
        where: {id}
       })

       if(!todo) throw `todo with id ${id} notfound`

       return TodoEntity.formJson(todo)
    }

    async updateById(updateTodoDto: UpdateTodoDto): Promise<TodoEntity> {
        const todo = await this.findById(updateTodoDto.id)
        const updatedTodo = await prisma.todo.update({
            where:{id:updateTodoDto.id},
            data: updateTodoDto.values
        })

        return TodoEntity.formJson(updatedTodo)
    }
}
```

---

## Uso del repositorio en los controladores

- En presentation/todos/todos.controller bien podría inyectar un servicio y es en el servicio dónde irían todas las implementaciones
- Pero aqui lo que quiero hacer, es antes de implementar los casos de usos, hacer uso del repositorio (lo inyecto)
- Podría mandar la implementación pero eso me obligaría a que siempre fuera esa implementación. Le mando el repo "genérico"
- Si voy a presentation/todos/routes me marca error porque necesito proveer a la instancia de TodosController el todoRepository

```
import { Router } from "express";
import { TodosController } from "./todos.controller";
import { TodoRepositoryImpl } from
"../../infraestructure/repositories/todo.repo.impl";
import { TodoDatasourceImpl } from
"../../infraestructure/datasource/todo.datasource.impl";

export class TodoRoutes{


    static get routes():Router{
        const router = Router();
```

```
        const todoDatasource = new TodoDatasourceImpl()
        const todoRepository = new TodoRepositoryImpl(todoDatasource)
        const todosController = new TodosController(todoRepository) //necesito
proveer el repositorio




        router.get('/', todosController.getTodos) //solo mandamos la referencia a
la función
        router.get('/:id', todosController.todoById) //solo mandamos la referencia
a la función
        router.post('/', todosController.createTodo)
        router.put('/:id', todosController.updateTodo)
        router.delete('/:id', todosController.deleteTodoById)
        return router
    }
}
```

- En el controller

```
import { Request, Response } from "express"
import { CreateTodoDto } from "../../domain/dtos/todos/todo.dto"
import { UpdateTodoDto } from "../../domain/dtos/todos/update.dto"
import { TodoRepository } from "../../domain/repositories/todo.repo"




export class TodosController{

        constructor(
            private readonly todoRepository: TodoRepository
        ){

        }

       public getTodos= async(req: Request, res: Response)=>{
            const todos= await this.todoRepository.getAll()

            return res.json(todos)
        }

    public async todoById(req: Request, res: Response){
        //const id = +req.params.id
        try{
            const todo = await this.todoRepository.findById(Number(req.params.id))
            return res.json(todo)
        }catch(error){
            res.status(400).json({error})
```

```
        }
    }

    public async createTodo(req:Request, res:Response){
        const [error, createTodoDto] = CreateTodoDto.create(req.body)
        if(error) return res.status(400).json({error})
        const todo= await this.todoRepository.create(createTodoDto!)
        return res.json(todo)

    }

    public updateTodo = async( req: Request, res: Response ) => {
        const [error,updateTodoDto] = UpdateTodoDto.create(req.body)
        if(error) return res.status(400).json({error})
        const updatedTodo= await this.todoRepository.updateById(updateTodoDto!)
        return res.json(updatedTodo)
      }


    public deleteTodoById(req:Request, res:Response){
        return this.todoRepository.deleteById(Number(req.params.id))
    }
}
```

---

## Casos de uso

- Hago una copia del controlador como backup
- Creo en domain/use-cases
    - create-todo.ts
    - update-todo.ts
    - delete-todo.ts
    - get-todos.ts
    - get-todo.ts
- Creo la interfaz. Es útil tenerla en el caso de que cambie el día de mañana
-

```
import { CreateTodoDto } from "../../dtos/todos/todo.dto";
import { TodoEntity } from "../../entities/todo.entity";
import { TodoRepository } from "../../repositories/todo.repo";

export interface CreateTodoUseCase{
    execute(dto:CreateTodoDto): Promise<TodoEntity>
}

export class CreateTodo implements CreateTodoUseCase{

    constructor(
        private readonly repository:TodoRepository
    ){}
```

```
    public execute(dto: CreateTodoDto){
        return this.repository.create(dto)
    }
}
```

- Hago lo mismo con elñ restoi de casos de uso (/copio, pego y modifico)
- delete

```
import { CreateTodoDto } from "../../dtos/todos/todo.dto";
import { TodoEntity } from "../../entities/todo.entity";
import { TodoRepository } from "../../repositories/todo.repo";

export interface DeleteTodoUseCase{
    execute(id:number): Promise<TodoEntity>
}

export class DeleteTodo implements DeleteTodoUseCase{

    constructor(
        private readonly repository:TodoRepository
    ){}

    public execute(id:number){
        return this.repository.deleteById(id)
    }
}
```

- update

```
import { CreateTodoDto } from "../../dtos/todos/todo.dto";
import { UpdateTodoDto } from "../../dtos/todos/update.dto";
import { TodoEntity } from "../../entities/todo.entity";
import { TodoRepository } from "../../repositories/todo.repo";

export interface UpdateTodoUseCase{
    execute(dto:UpdateTodoDto): Promise<TodoEntity>
}

export class UpdateTodo implements UpdateTodoUseCase{

    constructor(
        private readonly repository:TodoRepository
    ){}

    public execute(dto:UpdateTodoDto){
        return this.repository.updateById(dto)
    }
}
```

- get

```
import { CreateTodoDto } from "../../dtos/todos/todo.dto";
import { UpdateTodoDto } from "../../dtos/todos/update.dto";
import { TodoEntity } from "../../entities/todo.entity";
import { TodoRepository } from "../../repositories/todo.repo";

export interface GetTodoUseCase{
    execute(id:number): Promise<TodoEntity>
}

export class GetTodo implements GetTodoUseCase{

    constructor(
        private readonly repository:TodoRepository
    ){}

    public execute(id:number){
        return this.repository.findById(id)
    }
}
```

- get-todos

```
import { CreateTodoDto } from "../../dtos/todos/todo.dto";
import { UpdateTodoDto } from "../../dtos/todos/update.dto";
import { TodoEntity } from "../../entities/todo.entity";
import { TodoRepository } from "../../repositories/todo.repo";

export interface GetTodosUseCase{
    execute(): Promise<TodoEntity[]>
}

export class GetTodos implements GetTodosUseCase{

    constructor(
        private readonly repository:TodoRepository
    ){}

    public execute(){
        return this.repository.getAll()
    }
}
```

---

## Consumir los casos de uso

- Hago un archivo de barril de los casos de uso

- Consumo los casos de uso en el controlador
- Express recomienda no usar métodos asíncronos en el controlador. Usaremos .then

```
export class TodosController{

        constructor(
            private readonly todoRepository: TodoRepository
        ){

        }

       public getTodos=(req: Request, res: Response)=>{
            new GetTodos(this.todoRepository)
                .execute()
                .then(todos=> res.json(todos))
                .catch(error=> res.status(400).json({error}))
        }
}
```

- Vamos con el resto de casos

```
import { Request, Response } from "express"
import { CreateTodoDto } from "../../domain/dtos/todos/todo.dto"
import { UpdateTodoDto } from "../../domain/dtos/todos/update.dto"
import { TodoRepository } from "../../domain/repositories/todo.repo"
import { CreateTodo, DeleteTodo, GetTodo, GetTodos, UpdateTodo } from
"../../domain/use-cases"


export class TodosController{

        constructor(
            private readonly todoRepository: TodoRepository
        ){

        }

       public getTodos=(req: Request, res: Response)=>{
            new GetTodos(this.todoRepository)
                .execute()
                .then(todos=> res.json(todos))
                .catch(error=> res.status(400).json({error}))
        }

    public todoById=(req: Request, res: Response)=>{
        const id = +req.params.id
        new GetTodo(this.todoRepository)
        .execute(id)
        .then(todo=> res.json(todo))
        .catch(error=> res.status(400).json({error}))
```

```
    }

    public createTodo=(req:Request, res:Response)=>{
        const [error, createTodoDto] = CreateTodoDto.create(req.body)
        if(error) return res.status(400).json({error})
            new CreateTodo(this.todoRepository)
            .execute(createTodoDto!) //aqui no puede ser undefined
            .then(todo=> res.json(todo))
            .catch(error=> res.status(400).json({error}))
    }

    public updateTodo = ( req: Request, res: Response ) => {
    const id = +req.params.id
    const [error,updateTodoDto] = UpdateTodoDto.create({...req.body, id})
    if(error) return res.status(400).json({error})
    new UpdateTodo(this.todoRepository)
    .execute(updateTodoDto!)
    .then(todo=> res.json(todo))
    .catch(error=> res.status(400).json({error}))
    }


    public deleteTodoById=(req:Request, res:Response)=>{
        const id = +req.params.id
        new DeleteTodo(this.todoRepository)
        .execute(id)
        .then(todo=> res.json(todo))
        .catch(error=>res.status(400).json({error}))
    }
}
```

---

## Buenas prácticas

- Se puede usar compression para que gzip mejore el tiempo de respuesta

> npm i compression

```
this.app.use(compression())
```

---

# NODE TS Autenticacion

- Plantilla node configurado con typescript

> https://github.com/Klerith/node-ts-express-shell

> npm i

- Usaremos mongo
- En docker-compose descomento el primer bloque
- Compongo la db

> docker compose up -d

- En .env coloco el puerto, PORT=3000
- Con npm run dev corro la aplicación, en localhost:3000 en el navegador debo ver el mensaje "Tu eres increíble"
- *NOTA* ya tenemos la configuración del server básica hecha

---

## Módulo Auth - Rutas y controladores

- Creo presentation/auth/controller.ts y routes.ts
- El controlador solo da la respuesta al cliente
- Para las rutas puedo copiar lo que tengo en el archivo de routes y modificarlo

```
import { Router } from 'express';

export class AuthRoutes {


  static get routes(): Router {

    const router = Router();

   router.post('/login') //falta añadir el controlador
   router.post('/register')
   router.get('/validate-email/:token')

    return router;
  }
}
```

- Para usarlo en AppRoutes

```
import { Router } from 'express';
import { AuthRoutes } from './auth/routes';


export class AppRoutes {

  static get routes(): Router {

    const router = Router();

    router.use('/api/auth', AuthRoutes.routes)
```

```
    return router;
  }
}
```

- El controlador no va a ser más que una clase que me permita hacer inyección de dependencias

```
import { Request, Response } from "express"

export class AuthController{

    constructor(){

    }

    registerUser=(req:Request,res:Response)=>{
        res.json('registerUser')
    }

    loginUser=(req:Request,res:Response)=>{
        res.json('loginUser')
    }

    validateEmail=(req:Request,res:Response)=>{
        res.json('validateEmail')
    }
}
```

- Los coloco en las rutas. En AuthRoutes creo una instancia del controlador

```
import { Router } from 'express';
import { AuthController } from './controller';

export class AuthRoutes {

  static get routes(): Router {
    const router = Router();
    const controller = new AuthController();

   router.post('/login', controller.loginUser)
   router.post('/register', controller.registerUser)
   router.get('/validate-email/:token', controller.validateEmail)

    return router;
  }
}
```

# Conectar MongoDB

- Creo src/data/mongo/mongo-connection.ts
- Instalo mongoose

> npm i mongoose

```
import mongoose from "mongoose"

interface connectionOptions{
    mongoUrl: string
    dbName: string
}

export class MongoDBConnection{

    static async connect(options: connectionOptions){
        const {mongoUrl, dbName} = options

        try {
            await mongoose.connect(mongoUrl, {
                dbName
            })
            console.log('Mongo connected')
            return true

        } catch (error) {
            console.log('Mongo connection error')
            throw error
        }

    }
}
~~~-

- Debo definir mongoUrl y dbName en las .env
```

PORT=3000

MONGO_STRING=mongodb://mongo-user:123456@localhost:27017 MONGO_DB_NAME=mystore

```
- Coloco las variables de entorno en src/config/envs.ts

~~~js
import 'dotenv/config';
import { get } from 'env-var';
```

```
export const envs = {

  PORT: get('PORT').required().asPortNumber(),
  MONGO_STRING: get('MONGO_STRING').required().asString(),
  MONGO_DB_NAME: get('MONGO_DB_NAME').required().asString(),

}
```

- Para usarlas en MongoDbConnection en el main de app.ts

```
import { envs } from './config/envs';
import { MongoDBConnection } from './data/mongo/mongo-connection';
import { AppRoutes } from './presentation/routes';
import { Server } from './presentation/server';


(async()=> {
  main();
})();


async function main() {

  await MongoDBConnection.connect({
    mongoUrl: envs.MONGO_STRING,
    dbName: envs.MONGO_DB_NAME

  })

  const server = new Server({
    port: envs.PORT,
    routes: AppRoutes.routes,
  });

  server.start();
}
```

- *NOTA* En gitignore el / de mongo y progres es al inicio
- Ahora hay que hacer el modelo!

---

## User Model

- Creo /data/mongo/models/user.model.ts

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
```

```
    name: {
        type: String,
        required: [true, 'Name is required']
    },
    email: {
        type: String,
        required: [true, 'Email is required'],
        unique: true
    },
    emailvalidated:{
        type: Boolean,
        default: false
    },
    password: {
        type: String,
        required: [true, 'Password is required']
    },
    role:{
        type: [String],
        enum: ['ADMIN_ROLE', 'USER_ROLE'],
        default: 'USER_ROLE'
    },
    img:{
        type: String
    }
})


export const UserModel = mongoose.model('User', userSchema)
```

- Coloco el modelo el archivo de barril de /data

```
export * from './mongo/mongo-connection'
export * from './mongo/models/user.model'
```

---

## Custom Error

- Creo domain/errors/cutom.error.ts
- Hago el método privado para que solo sean los métodos estáticos quienes creen y presenten el CustomError

```
export class CustomError extends Error{
    private constructor(
        public readonly statusCode: number,
        public readonly message: string
    ){
        super(message)
```

```
    }

    static badRequest(message: string): CustomError {
        return new CustomError(400, message)
    }
}
```

- Creo los otros

```
export class CustomError extends Error{
   private constructor(
        public readonly statusCode: number,
        public readonly message: string
    ){
        super(message)
    }

    static badRequest(message: string): CustomError {
        return new CustomError(400, message)
    }

    static unauthorized(message: string): CustomError {
        return new CustomError(401, message)
    }
    static forbidden(message: string): CustomError {
        return new CustomError(404, message)
    }
    static internalServer(message: string): CustomError {
        return new CustomError(500, message)
    }
}
```

- Creo un archivo de barril para exportar los errores en /domain

```
export * from './errors/custom.error'
```

---

- Creemos la entidad de Usuario en domain/entities para que rija mi DB

---

## User Entity

- Yo no quiero regresar un modelo de mongoose, porque si algo cambia, y no uso mongoose es una dependencia fuerte que debo resolver
- Si pones una propiedad opcional en el constructor debe ir al final

```

```

```
import { CustomError } from "../../errors/custom.error"

export class UserEntity{
    constructor(
        public id:string,
        public name: string,
        public email: string,
        public emailValidated: boolean,
        public password: string,
        public role: string[],
        public img?: string,

    ){}

    static fromObject(obj:{[key:string]:any}): UserEntity{

        const {id, _id, name, email, emailValidated, password, role, img} = obj

        if(!id && !_id) throw CustomError.badRequest('Missing id')
        if(!name) throw CustomError.badRequest('Missing name')
        if(!email) throw CustomError.badRequest('Missing email')
        if(emailValidated=== undefined) throw CustomError.badRequest('Missing
emailValidated')
        if(!password) throw CustomError.badRequest('Missing password')
        if(!role) throw CustomError.badRequest('Missing role')

        //hay que ponerlo en el mismo orden que el constructor
        return new UserEntity(
            id || _id,
            name,
            email,
            password,
            role,
            emailValidated,
            img
        )
    }
}
```

---

## RegisterUserDto

- Creo el DTO en domain/dtos/auth/register-user.dto con la información que espero que me manden para registrar un usuario
- Espero name, email y password
- Hay que validar que sea un email válido. Coloco la expresión regular en config/regular-exp.ts

```
export const regularExps = {

  // email
```

```
  email: /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/,

}
```

- El dto

```
import { regularExps } from "../../../config/regular-exp";

export class RegisterUserDto{
    constructor(
        public name: string,
        public email: string,
        public password: string
    ){}

    static create(object:{[key:string]: any}): [string?, RegisterUserDto?]{
        const {name, email, password} = object;
        if(!name || !email || !password){
            return ['Invalid input', undefined];
        }

        if(!regularExps.email.test(email)) return ['Invalid email', undefined]
        if(password.length < 6) return ['Password must be at least 6 characters',
undefined]

        return [undefined, new RegisterUserDto(name, email, password)];

    }
}
```

---

## AuthService

- Creo presentation/services/auth.service.ts

```
import { UserModel } from "../../data";
import { CustomError } from "../../domain";
import { RegisterUserDto } from "../../domain/dtos/auth/register-user.dto";

export class AuthService{
    constructor(){

    }

    public async registerUser(registerUserDto: RegisterUserDto){

        const existUser = await UserModel.findOne({email: registerUserDto.email});
        if(existUser) throw CustomError.badRequest('User already exists');
```

```
        return 'todo ok!'
    }
}
```

- Para inyectar el servicio en el controller puedo hacer un singleton para que solo haya una instancia del servicio
- O puedo crear la instancia en el controller dentro del mismo constructor

```
export class AuthController{

    constructor(
        public readonly authService: AuthService = new AuthService()
    ){

    }
}
```

- O puedo instanciarla en /auth/routes que es dónde lo voy a usar

```
import { Router } from 'express';
import { AuthController } from './controller';
import { AuthService } from '../services/auth.service';


export class AuthRoutes {


  static get routes(): Router {
    const router = Router();
    const authService = new AuthService()
    const controller = new AuthController(authService);

   router.post('/login', controller.loginUser)
   router.post('/register', controller.registerUser)
   router.get('/validate-email/:token', controller.validateEmail)



    return router;
  }

}
```

- Hago uso del servicio

```
export class AuthController{
```

```
    constructor(
        public readonly authService: AuthService
    ){

    }

    registerUser=(req:Request,res:Response)=>{
        const [error, registerUserDto] = RegisterUserDto.create(req.body);
        if(error) return res.status(400).json({error});

        this.authService.registerUser(registerUserDto!)
            .then((user)=>res.json(user))
            .catch((error)=>res.status(400).json({error}))

    }
}
```

---

## Crear usuario y manejo de errores

- El controlador delega en el servicio la creación del usuario
- auth.service

```
import { UserModel } from "../../data";
import { CustomError } from "../../domain";
import { RegisterUserDto } from "../../domain/dtos/auth/register-user.dto";

export class AuthService{
    constructor(){

    }

    public async registerUser(registerUserDto: RegisterUserDto){

        const existUser = await UserModel.findOne({email: registerUserDto.email});
        if(existUser) throw CustomError.badRequest('User already exists');

        try {
            const user = new UserModel(registerUserDto);
            await user.save();

            return user

        } catch (error) {
            throw CustomError.internalServer(`${error}`)
        }

        return 'todo ok!'
    }
}
```

- Si lo hago asi, el return user me regresa el password sin encriptar y el _id
- Vamos a manejar el error del servicio en el AuthController

```
private handleError = (error:unknown, res: Response)=>{
    if(error instanceof Error){
        return res.status(400).json({error: error.message})
    }
    res.status(500).json({error: 'Internal Server Error'})
}

registerUser=(req:Request,res:Response)=>{
    const [error, registerUserDto] = RegisterUserDto.create(req.body);
    if(error) return res.status(400).json({error});

    this.authService.registerUser(registerUserDto!)
        .then((user)=>res.json(user))
        .catch((error)=>this.handleError(error, res))

}
```

- Creamos un usuario correctamente, nos conectamos con mongoCompass para comprobar que todo esté bien
- No quiero regresar el password, ni el _id, ni __v
- Podría coger el objeto en AuthService y crear con mi entidad para que se encargue de validar todo

---

- *NOTA*: si no tienes configurada la autenticación en MONGO deberás crear un usuario y habilitarla. Si tienes corriendo el servicio de mongod y docker a la vez, es posible que MongoCompass no te muestre la db,
- Para detener el servicio en windows, abrir PowerShell con privilegios > net stop MongoDB

---

```
public async registerUser(registerUserDto: RegisterUserDto){

    const existUser = await UserModel.findOne({email: registerUserDto.email});
    if(existUser) throw CustomError.badRequest('User already exists');

    try {
        const user = new UserModel(registerUserDto);
        await user.save();

        const {password, ...rest} = UserEntity.fromObject(user) //no quiero
retornar el password

        return {user: rest, token: 'ABC'}



    } catch (error) {
```

```
            throw CustomError.internalServer(`${error}`)
        }


    }
```

- También podría inyectar el repositorio en el servicio para hacerlo más sostenible

---

## Encriptar contraseñas

- Usemos bcrypt.js y el patrón adaptador
- En srx/config/bcrypt.adapter.ts. Bien podría hacerlo con métodos estáticos de una clase

```
import {compareSync, genSaltSync, hashSync} from 'bcryptjs'


export const bcryptAdapter= {
    hash: (password:string)=>{
        const salt = genSaltSync()
        return hashSync(password,salt)
    },
    compare: (password: string, hashed: string)=>{
        return compareSync(password,hashed)
    }
}
```

- Encripto el password

```
public async registerUser(registerUserDto: RegisterUserDto){

    const existUser = await UserModel.findOne({email: registerUserDto.email});
    if(existUser) throw CustomError.badRequest('User already exists');

    try {
        const user = new UserModel(registerUserDto);

        user.password = bcryptAdapter.hash(registerUserDto.password) //encripto el
password antes de guardar

        await user.save();

        const {password, ...rest} = UserEntity.fromObject(user)
        return {user: rest, token: 'ABC'}

    } catch (error) {
        throw CustomError.internalServer(`${error}`)
    }
}
```

---

- Hagamos el login y luego evaluemos el mail de confirmación con el token

---

## Login de usuario

- LoginUserDto

```
import { regularExps } from "../../../config/regular-exp"

export class LoginUserDto{
    constructor(
        public readonly email: string,
        public readonly password: string
    ){}

    static create(object:{[key:string]:any}):[string?, LoginUserDto?] {
        const {email, password} = object

        if(!email) return ['Email is required', undefined]
        if(!regularExps.email.test(email)) return ['Invalid email', undefined]
        if(!password) return ['Password is required', undefined]
        if(password.length >6) return ['Password too short']

        return [undefined, new LoginUserDto(email, password)]
    }
}
```

- AuthController

```
import { regularExps } from "../../../config/regular-exp"

interface Options{
    email: string | null
    password: string | null
}


export class LoginUserDto{
    constructor(options:Options){}

    static create(options:{[key:string]:any}):[string?, LoginUserDto?] {
        const {email, password}= options

        if(!email) return ['Email is required', undefined]
        if(!regularExps.email.test(email)) return ['Invalid email', undefined]
        if(!password) return ['Password is required', undefined]
        if(password.length >6) return ['Password too short']
```

```
            return [undefined, new LoginUserDto({email, password})]
        }
    }
```

- AuthService

```
public async loginUser(loginUserDto: LoginUserDto){
    const user = await UserModel.findOne({email:loginUserDto.email })

    if(!user) throw CustomError.badRequest("User don't exists!")

    const hashMatch = bcryptAdapter.compare(loginUserDto.password, user.password)

    if(!hashMatch) throw CustomError.unauthorized("Password is not valid")

    const {password, ...userEntity} = UserEntity.fromObject(user)

    return {
        user: userEntity,
        token: 'ABC'
    }
}
```

---

## Generar JWT

- Instalamos jsonwebtoken

> npm i jsonwebtoken

- Siempre que uses una librería de terceros usa un adaptador
- En config/jwt.adapter.ts la idea es que solo en este archivo voy a tener la dependencia directa a la librería
- Si no necesito inyección de dependencias puedo usar un método estático
- jwt.sign funciona con callbacks, puedo volver el método async y envolverlo todo en una promesa
- El SEED es la semilla que vamos a usar para firmar los tokens

```
import jwt from 'jsonwebtoken'

export class JwtAdapter{


   static async generateToken(payload:any, duration:string= '2h'){

    return new Promise(resolve=>{

        jwt.sign(payload, "SEED", {expiresIn: duration}, (err, token)=>{
            if(err) return resolve(null)
```

```
                resolve(token)
            })
        })

0
    }

    static validateToken(token:string){

    }

}
```

- En el AuthService uso ek JwtAdapter
- Como uso el resolve siempre resuelve de una manera exitosa
- Si no lo hace (la generación del token puede fallar) devuelve un InternalServerError

```
public async loginUser(loginUserDto: LoginUserDto){
    const user = await UserModel.findOne({email:loginUserDto.email })

    if(!user) throw CustomError.badRequest("User don't exists!")

    const hashMatch = bcryptAdapter.compare(loginUserDto.password, user.password)

    if(!hashMatch) throw CustomError.unauthorized("Password is not valid")

    const {password, ...userEntity} = UserEntity.fromObject(user)

    const token= await JwtAdapter.generateToken({id: user.id})
    if(!token) throw CustomError.internalServer("Error generating token")

    return {
        user: userEntity,
        token: token
    }
}
```

- Añado también la generación de token en registerUser y lo paso en lugar de 'ABC'

---

## Jwt SEED

- La semilla no debe estar comprometida
- Creemos una variable de entorno

```
JWT_SEED=seed
```

- /config/envs.ts

```
import 'dotenv/config';
import { get } from 'env-var';


export const envs = {

  PORT: get('PORT').required().asPortNumber(),
  MONGO_STRING: get('MONGO_STRING').required().asString(),
  MONGO_DB_NAME: get('MONGO_DB_NAME').required().asString(),
  JWT_SEED: get('JWT_SEED').required().asString(),

}
```

- En el JwtAdapter podria usar un constructor (y dejar de usar métodos estáticos)
- También puedo mandar el seed en el método generarToken
- Para indicar que va a haber una dependencia lo indico con una constante
- *NOTA*: sería mejor usar el constructor

```
import jwt from 'jsonwebtoken'
import { envs } from './envs'

const JWT_SEED = envs.JWT_SEED


export class JwtAdapter{


   static async generateToken(payload:any, duration:string= '2h', seed= JWT_SEED){

    return new Promise(resolve=>{

        jwt.sign(payload, seed, {expiresIn: duration}, (err, token)=>{
            if(err) return resolve(null)

            resolve(token)
        })
    })

0
   }

   static validateToken(token:string){

   }

}
```

# NODE TS Envio Correo + Validación Token

## Email Service

- **NOTA**: para generar la contraseña de aplicación de gmail acceder desde la cuenta en *https://myaccount.google.com/u/0/apppasswords*

- Creo presentation/services/email.service.ts

```
import nodemailer from 'nodemailer'

interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{

    constructor(){

    }
    private transporter= nodemailer.createTransport({
        service: envs.MAILER_SERVICE, //NO TENGO LAS VARIABLES DE ENTORNO
        auth:{
            user: envs.MAILER_EMAIL,
            pass: envs.MAILER_SECRET_KEY
        },
        tls: {
            rejectUnauthorized: false
        }
    })

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
```

```
                    attachments
                })

                console.log(sentInformation)

                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email sent',
                    origin: 'email.service'
                })
               // this.logRepository.saveLog(log)

                return true
            } catch (error) {

                console.log(error)
                const log = new LogEntity({
                    level: LogSeverityLevel.low,
                    message: 'Email was no sent',
                    origin: 'email.service'
                })
                //this.logRepository.saveLog(log)

                return false
            }
    }

    async sendemailWithFileSystemLogs(to: string | string[]){
            const subject= 'Logs del servidor'
            const htmlBody=`
            <h3>Logs del sistema</h3>
            <p>Desde sendEmailWithFileSystem</p>
            `

        const attachments: Attachment[]= [
            {filename: 'logs-all.log', path: './logs/logs-all.log'},
            {filename: 'logs-high.log', path: './logs/logs-high.log'},
            {filename: 'logs-medium.log', path: './logs/logs-medium.log'},
        ]


        return this.sendEmail({to, subject, attachments, htmlBody})
    }
}
```

- No tengo las variables de entorno
- Quiero evitar esta dependencia oculta
- En lugar de hacerlo asi, declaro transporter de nodemailer pero no lo inicializo
- Lo hago en el constructor
- No tengo LogEntity, lo borro por el momento
- Pasándole las variables por el constructor, me deshago de la dependencia oculta

```
import nodemailer, { Transporter } from 'nodemailer'



interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{

    private transporter: Transporter;

    constructor(
        mailerService: string,
        mailerEmail: string,
        senderEmailPassword: string
    ){
        this.transporter= nodemailer.createTransport({
            service: mailerService,
            auth:{
                user: mailerEmail,
                pass: senderEmailPassword
            },
            tls: {
                rejectUnauthorized: false
            }
        })
    }

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments} = options

            try {

                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
                    attachments
                })

                return true
            } catch (error) {
                return false
```

```
            }
        }

}
```

- Hay que rellenar las variables de entorno
- *NOTA* para obtener la secret_key hay que activar el 2 factor auth, y generarla en contraseña de aplicaciones

```
MAILER_SERVICE=gmail
MAILER_EMAIL=ismaelberoncastano@gmail.com
MAILER_SECRET_KEY=kludyhcnbiuecrfby
```

- Hay que configurarlas en config/envs.ts

```
import 'dotenv/config';
import { get } from 'env-var';


export const envs = {

  PORT: get('PORT').required().asPortNumber(),
  MONGO_STRING: get('MONGO_STRING').required().asString(),
  MONGO_DB_NAME: get('MONGO_DB_NAME').required().asString(),
  JWT_SEED: get('JWT_SEED').required().asString(),
  MAILER_SERVICE: get('MAILER_SERVICE').required().asString(),
  MAILER_EMAIL:get('MAILER_EMAIL').required().asString(),
  MAILER_SECRET_KEY:get('MAILER_SECRET_KEY').required().asString()

}
```

- Cuando registramos un usuario debemos enviar un correo de confirmación
- Inyecto el emailService (hay varias maneras de hacerlo)
- Creo un método privado para este caso concreto
- Debo generar el token para la validación
- Creo el link (lo añado como variable de entorno)
- Creo el html del email
- Configuro el objeto de opciones que le pasaré a sendEmail
- Empleo el servicio para enviar el email
- *NOTA* en proximas clases crearemos un túnel para poder autenticarnos con el teléfono sin tener desplegada la aplicación
- presentation/services/auth.service

```
import { Request, Response } from "express";
import { bcryptAdapter } from "../../config/bcrypt.adapter";
```

```
import { UserModel } from "../../data";
import { CustomError } from "../../domain";
import { LoginUserDto } from "../../domain/dtos/auth/login-user.dto";
import { RegisterUserDto } from "../../domain/dtos/auth/register-user.dto";
import { UserEntity } from "../../domain/entities/user/user.entity";
import { JwtAdapter } from "../../config/jwt.adqapter";
import { EmailService } from "./email.services";
import { envs } from "../../config/envs";

export class AuthService{
    constructor(
        private readonly emailService: EmailService
    ){

    }

    public async registerUser(registerUserDto: RegisterUserDto){

        const existUser = await UserModel.findOne({email: registerUserDto.email});
        if(existUser) throw CustomError.badRequest('User already exists');

        try {
            const user = new UserModel(registerUserDto);

            user.password = bcryptAdapter.hash(registerUserDto.password)
//encripto el password antes de guardar

            await user.save();

            await this.sendEmailValidationLink(user.email) //ENVIO EL MQAIL DE
CONFIRMACIÓN!!

            const {password, ...userEntity}= UserEntity.fromObject(user)
            const token= await  JwtAdapter.generateToken({id: user.id})
            if(!token) throw CustomError.internalServer("Error generating token")
            return {user: userEntity, token: token}

        } catch (error) {
            throw CustomError.internalServer(`${error}`)
        }


    }

    public async loginUser(loginUserDto: LoginUserDto){
        const user = await UserModel.findOne({email:loginUserDto.email })

        if(!user) throw CustomError.badRequest("User don't exists!")

        const hashMatch = bcryptAdapter.compare(loginUserDto.password,
user.password)

        if(!hashMatch) throw CustomError.unauthorized("Password is not valid")
```

```
        const {password, ...userEntity} = UserEntity.fromObject(user)

        const token= await  JwtAdapter.generateToken({id: user.id})
        if(!token) throw CustomError.internalServer("Error generating token")

        return {
            user: userEntity,
            token: token
        }
    }

      private sendEmailValidationLink =async(email:string)=>{
        const token = await JwtAdapter.generateToken(email)

        if(!token) throw CustomError.internalServer('Error generating token')

        const link = `${envs.WEBSERVICE_URL}/auth/validate-email/${token}`

        const html=`
        <h1>Validar Email</h1>
        <p>Este es un correo para validar tu cuenta</p>
        <a href=${link}>Valida tu email: ${email}</a>
        <p>por favor, si no has sido tu ignóralo</p>
        <p>Gracias</p>
        `

        const options= {
            to: email,
            subject: 'Valida tu email',
            htmlBody: html
        }

        const isSent= await this.emailService.sendEmail(options)

        if(!isSent) throw CustomError.internalServer("Error sending email")

        return true
    }
}
```

- En las rutas del auth voy a tener un error porque el authService está esperando que le pase el emailService
- Como son tres argumentos podría crear un objeto en la clase para pasar al constructor con una interfaz y luego generar aquí el objeto con las variables
- Recuerda que tienen que estar en orden

```
import { Router } from 'express';
import { AuthController } from './controller';
import { AuthService } from '../services/auth.service';
import { envs } from '../../config/envs';
import { EmailService } from '../services/email.services';
```

```
export class AuthRoutes {


  static get routes(): Router {
    const router = Router();

    const emailService = new
EmailService(envs.MAILER_SERVICE,envs.MAILER_EMAIL,envs.MAILER_SECRET_KEY)
    const authService = new AuthService(emailService)
    const controller = new AuthController(authService);

   router.post('/login', controller.loginUser)
   router.post('/register', controller.registerUser)
   router.get('/validate-email/:token', controller.validateEmail)



    return router;
  }


}
```

---

## Probar envío de correos

- Coloco console.logs en el servicio para ver que puede salir mal
- En el body envio un correo válido para poder chequear el mail

---

## Validar Token

- En auth.controller voy a validateEmail

```
validateEmail=(req:Request,res:Response)=>{
    const {token} = req.params

    this.authService.validateEmail(token)
        .then((user)=>res.json('Email validated'))
        .catch((error)=>this.handleError(error, res))
}
```

- En auth.service creo el método validateEmail pero primero CREO EL METODO PARA VERIFICAR EL TOKEN EN EL JWT.ADAPTER
- JwtAdapter

```
static validateToken(token:string){
return new Promise(resolve=>{
    jwt.verify(token, JWT_SEED, (err, decoded)=>{
        if(err) return resolve(null)
        resolve(decoded)
    })
})

}
```

- En auth.service

```
public validateEmail= async(token:string)=>{
    const payload = await JwtAdapter.validateToken(token);

    if (!payload) throw CustomError.badRequest('Invalid token');

    const { email } = payload as { email: string };
    if (!email) throw CustomError.internalServer('Email not in token');

    const user = await UserModel.findOne({ email });

    if (!email) throw CustomError.internalServer('Error getting email');

    if (!user) throw CustomError.internalServer('User not found');

    user.emailValidated = true;

    await user.save()
    return true

}
```

- Ahora puedo probar con el link que hay en mi correo

---

# Node TS Protecciónde rutas, middlewares y relaciones

- Creo una variable en false para evitar seguir mandando correos electrónicos (eso ya lo probamos)

```
SEND_EMAIL=false
```

- La configuro en envs.ts
- Le pongo false por defecto. Las variables de entorno siempre tienen que ser strings

```
SEND_EMAIL: get('SEND_EMAIL').default('false').asBool()
```

- Creo una nueva propiedad en el EmailService
- Uso la forma corta con private readonly en el constructor

```
import nodemailer, { Transporter } from 'nodemailer'



interface SendEmailOptions{
    to: string | string[]
    subject: string
    htmlBody: string
    attachments?: Attachment[]
}

interface Attachment{
    filename?: string
    path?: string
}

export class EmailService{

    private transporter: Transporter;

    constructor(
        mailerService: string,
        mailerEmail: string,
        senderEmailPassword: string,
        private readonly postToProvider: boolean //añado la propiedad
    ){
        this.transporter= nodemailer.createTransport({
            service: mailerService,
            auth:{
                user: mailerEmail,
                pass: senderEmailPassword
            },
            tls: {
                rejectUnauthorized: false
            }
        })
    }

    async sendEmail(options: SendEmailOptions): Promise<boolean>{

        const {to, subject,htmlBody, attachments=[]} = options

            try {

                    if(!this.postToProvider) return true //retorno true simulando
el envio de correo
```

```
                const sentInformation = await this.transporter.sendMail({
                    to,
                    subject,
                    html: htmlBody,
                    attachments
                })

                console.log(sentInformation)
                return true
            } catch (error) {
                console.log(error)
                return false
            }
    }
}
```

- En AuthRoutes le paso la variable de entorno a la instancia de EmailService

```
export class AuthRoutes {


  static get routes(): Router {
    const router = Router();

    const emailService = new
EmailService(envs.MAILER_SERVICE,envs.MAILER_EMAIL,envs.MAILER_SECRET_KEY,
envs.SEND_EMAIL)
    const authService = new AuthService(emailService)
    const controller = new AuthController(authService);

   router.post('/login', controller.loginUser)
   router.post('/register', controller.registerUser)
   router.get('/validate-email/:token', controller.validateEmail)

    return router;
  }
}
```

---

## Preparación de los modelos restantes

- Vamos a usar el token cuando alguna persona quiera crearse algo
- Cual es la diferencia entre el modelo y la entidad?
    - El modelo está amarrado a la base de datos
- Creo category.model
- La categtoría estará asociada a un usuario
- Creo la relación

- En la referencia pongo el nombre que le puse como primer parámetro en la creación del modelo
    - mongoose.model("**Este_nombre**", Schema)

```
import mongoose, { Schema } from "mongoose";

const categorySchema = new mongoose.Schema({

    name: {
        type: String,
        required: [true, 'Name is required']
    },
    available:{
        type: Boolean,
        default: true
    },

    user:{
        type: Schema.Types.ObjectId,
        ref: 'User',
        required: true
    }

})


export const CategoryModel = mongoose.model('Category', categorySchema)
```

- Creo también el modelo de producto
- Si no indico que la propiedad es requerida y no le pongo ningún valor por defecto, ni aparecerá en el documento
    - La hace opcional automáticamente

```
import mongoose, { Schema } from "mongoose";

const productSchema = new mongoose.Schema({

    name: {
        type: String,
        required: [true, 'Name is required']
    },
    available:{
        type: Boolean,
        default: true
    },
    price: {
        type: Number,
        default: 0
    },
    description:{
        type: String
```

```
    },
    user:{
        type: Schema.Types.ObjectId,
        ref: 'User',
        required: true
    },
    category:{
        type: Schema.Types.ObjectId,
        ref: 'Category',
        required: true
    }

})


export const ProductModel = mongoose.model('Product', productSchema)
```

---

## Category - Rutas y Controlador

- De momento, si no existe la ruta devuelve lo que hay en la carpeta pública
- Creo presentation/categories/routes.ts, controller.ts y services/categories.service.ts
- La ruta llama al controlador que maneja las rutas, este llama al servicio dónde tengo toda mi lógica e interactúo con la db
- Básicamente copio lo que hay en AppRoutes y lo modifico
- Lo mismo con el controlador

```
import { Request, Response } from "express"
import { RegisterUserDto } from "../../domain/dtos/auth/register-user.dto";
import { AuthService } from "../services/auth.service";
import { LoginUserDto } from "../../domain/dtos/auth/login-user.dto";

export class CategoryController{

    constructor(

    ){}

    private handleError = (error:unknown, res: Response)=>{
        if(error instanceof Error){
            return res.status(400).json({error: error.message})
        }
        res.status(500).json({error: 'Internal Server Error'})
    }

    public getCategories(){

    }

    public createCategory(){
```

```
    }
}
```

- En las rutas

```
import { Router } from 'express';
import { CategoryController } from './controller';
import { getEnabledCategories } from 'trace_events';


export class CategoryRoutes {


  static get routes(): Router {
    const categoryController = new CategoryController()
    const router = Router();

    router.get('/', categoryController.getCategories)
    router.post('/', categoryController.createCategory)

    return router;
  }
}
```

- Añado el router en AppRoutes

```
import { Router } from 'express';
import { AuthRoutes } from './auth/routes';
import { CategoryRoutes } from './categories/routes';


export class AppRoutes {


  static get routes(): Router {

    const router = Router();

    router.use('/api/auth', AuthRoutes.routes)
    router.use('/api/categories', CategoryRoutes.routes)

    return router;
  }
}
```

---

## CreateCategoryDTO

- Necesito mandar el usuario pero vamos a obviarlo por ahora

```
export class CategoryDto{
   private constructor(
       public readonly name: string,
       public readonly avaliable: boolean
    ){}

    static create(object: {[key:string]:any}):[string?, CategoryDto?] {

        const {name, avaliable}=object
        if(!name) return [`Name ${name} don't exists`, undefined]
        let avaliableBoolean= avaliable
        if(typeof avaliable !== 'boolean'){
            avaliableBoolean = (avaliable === true)
        }

        return [undefined, new CategoryDto(name, avaliableBoolean)]
    }
}
```

- Usémoslo en el controller
- presentation/categories/controller.ts

```
public createCategory(req:Request, res: Response){
    const [error, categoryDto] = CategoryDto.create(req.body)
    if(error) return res.status(400).json({error})

    res.json(categoryDto)
}
```

---

## Auth Middleware - proteger rutas

- Cómo obtengo el usuario?
- Cuando hacemos un login, el token que se crea solo tiene el id del usuario
- Hay que tomar el token y mandarlo mediante los headers
- Seleccionamos Bearer Token en POSTMAN o similares y pego el token del login
- El middleware no es una regla de negocio, va en la capa de presentación
- El token está en la Response, en header Authorization
- Le añado el tipado genérico a valideateToken jwt.adapter
- Puede retornar un genérico o null
- Le digo que trate el decoded de tipo genérico

```
static validateToken<T>(token:string): Promise<T | null>{
    return new Promise(resolve=>{
        jwt.verify(token, JWT_SEED, (err, decoded)=>{
```

```
            if(err) return resolve(null)
            resolve(decoded as T)
        })
    })
```

- Después de hacer las validaciones y encontrar el usuario, lo paso al req.body
- Quiero que este user sea una instancia de mi entidad, para que no esté amarrado a la db y a mongoose
- Uso fromObject de UserEntity
- presentation/middlewares/auth.middleware.ts

```
import { NextFunction, Request, Response } from "express";
import { JwtAdapter } from "../../config/jwt.adqapter";
import { UserModel } from "../../data";
import { UserEntity } from "../../domain/entities/user/user.entity";

export class AuthMiddleware{

    static async validateJWT(req:Request,res:Response,next: NextFunction ){
        const authorization= req.header('Authorization')

        if(!authorization) return res.status(401).json({error:"No token
provider"})

        if(!authorization.startsWith('Bearer ')) return
res.status(401).json({error: "Invalid Bearer Token"})

        const token = authorization.split(' ').at(1)  || ""//es lo mismo que poner
[1]

        try {
            const payload = await JwtAdapter.validateToken<{id:string}>(token)
            if(!payload) return res.status(401).json({error: "Invalid Token"})

            const user = await UserModel.findById(payload.id)
            if(!user) return res.status(401).json({error:"Invalid Token- user"})

            //todo: validar si el usuario está activo
            req.body.user = UserEntity.fromObject(user)

            next()

        } catch (error) {
            res.status(500).json({error:"Internal Server error"})
        }

    }
}
```

---

## Probar AuthMiddleware

- Como solo debo aplicarlo al posteo de categoria voy al CategoryRoute
- Para colocar el middleware lo pongo como segundo argumento. Si hay varios puedo colocarlos en un arreglo

```
import { Router } from 'express';
import { CategoryController } from './controller';
import { getEnabledCategories } from 'trace_events';
import { AuthMiddleware } from '../middlewares/auth.middleware';


export class CategoryRoutes {


  static get routes(): Router {
    const categoryController = new CategoryController()
    const router = Router();

    router.get('/', categoryController.getCategories)
    router.post('/', [AuthMiddleware.validateJWT],
categoryController.createCategory)

    return router;
  }
}
```

- El usuario lo tengo en el req.body
- Creo el CategoryService

```
import { CategoryModel } from "../../data/mongo";
import { CustomError } from "../../domain";
import { CategoryDto } from "../../domain/dtos/auth/categories/category.dto";
import { UserEntity } from "../../domain/entities/user/user.entity";

export class CategoryService{


    constructor(){}

    async createCategory(categoryDto:CategoryDto, user:UserEntity){
        const categoryExists = await CategoryModel.findOne({name:
categoryDto.name})
        if(categoryExists) throw CustomError.badRequest("Category exists")

            try {
                const category= new CategoryModel({
                    ...categoryDto,
                    user: user.id
                })

                await category.save()
```

```
                return {
                    id: category.id,
                    name: category.name,
                    available: category.available
                }

            } catch (error) {
                throw CustomError.internalServer(`${error}`)
            }
    }
}
```

- En el controlador necesito usar el servicio. Debo inyectarlo

```
import { Request, Response } from "express"
import { RegisterUserDto } from "../../domain/dtos/auth/register-user.dto";
import { AuthService } from "../services/auth.service";
import { LoginUserDto } from "../../domain/dtos/auth/login-user.dto";
import { CategoryDto } from "../../domain/dtos/auth/categories/category.dto";
import { CustomError } from "../../domain";
import { CategoryService } from "../services/category.service";

export class CategoryController{

    constructor(
        private readonly categoryService: CategoryService

    ){}

    private handleError = (error:unknown, res: Response)=>{
        if(error instanceof Error){
            return res.status(400).json({error: error.message})
        }
        res.status(500).json({error: 'Internal Server Error'})
    }

    public getCategories(){

    }

    public createCategory=(req:Request, res: Response)=>{
        const [error, categoryDto] = CategoryDto.create(req.body)
        if(error) return res.status(400).json({error})

        this.categoryService.createCategory(categoryDto!, req.body.user)
            .then(category =>res.json(category))
            .catch(error=>res.json(`${error}`))
    }
}
```

- Debo pasarle una instancia del servicio a la instancia del controller en routes

```
import { Router } from 'express';
import { CategoryController } from './controller';
import { getEnabledCategories } from 'trace_events';
import { AuthMiddleware } from '../middlewares/auth.middleware';
import { CategoryService } from '../services/category.service';


export class CategoryRoutes {


  static get routes(): Router {

    const categoryController = new CategoryController(new CategoryService())
    const router = Router();

    router.get('/', categoryController.getCategories)
    router.post('/', [AuthMiddleware.validateJWT],
categoryController.createCategory)

    return router;
  }
}
```

---

## Retornar todas las categorias

- En el servicio

```
async getCategories(){
    try {
        const categories = await CategoryModel.find()
        return categories.map(category=>({
            id: category.id,
            name: category.name,
            available: category.available
        }))

    } catch (error) {
        throw CustomError.internalServer(`${error}`)
    }
}
```

- En el controller

```
public getCategories = (req:Request, res:Response)=>{
    this.categoryService.getCategories()
```

```
        .then(categories=>res.json(categories))
        .catch(error=>this.handleError(error, res))
}
```

---

## Paginación (DTO)

- Los query parameters suelen ser opcionales

```
export class PaginationDto{


    constructor(
        public readonly page:number,
        public readonly limit:number
    ){}

    static create(page:number =1,limit:number = 10 ):[string?, PaginationDto?]{

        if(typeof page !== 'number' || typeof limit !== 'number') return ['Invalid
data', undefined]
        if(page < 0 || limit < 0) return ['Page and  limit must be greater than
0', undefined]


        return [undefined, new PaginationDto(page, limit)]
    }
}
```

- En el controller

```
public getCategories = (req:Request, res:Response)=>{

    const {page=1, limit=10} = req.query
    const [error, paginationDto] = PaginationDto.create(Number(page),
Number(limit))
    if(error) return res.status(400).json({error})

    this.categoryService.getCategories(paginationDto)
        .then(categories=>res.json(categories))
        .catch(error=>this.handleError(error, res))
}
```

- Si pongo 1 * limit sería skip 10, pasaría a la página 2
- Si pongo 0 * limit, le estoy diciendo que haga un skip de 0 por lo que estaríamos en la página 1

```
async getCategories(paginationDto:PaginationDto){
    const {page, limit} = paginationDto

    try {
        const categories = await CategoryModel.find()
        .skip((page-1) * limit)
        .limit(limit)

        return categories.map(category=>({
            id: category.id,
            name: category.name,
            available: category.available
        }))

    } catch (error) {
        throw CustomError.internalServer(`${error}`)
    }
}
```

- Para hacer un recuento de todos los documentos uso

```
const total = await CategoryModel.countDocuments()
```

- Puedo disparar las dos promesas de manera simultánea

```
async getCategories(paginationDto:PaginationDto){

    const {page, limit} = paginationDto

    try {

        const [total, categories] = await Promise.all([
            CategoryModel.countDocuments(),
            CategoryModel.find()
            .skip((page-1) * limit)
            .limit(limit)
        ])

        return{
            page,
            limit,
            total,
            next: (page-1 >0)?`/api/v1/categories?
page=${page+1}&limit=${limit}`:null,
            prev: `/api/v1/categories?page=${page-1}&limit=${limit}`,


            categories: categories.map(category=>({
```

```
                id: category.id,
                name: category.name,
                available: category.available
            }))
        }


    } catch (error) {
        throw CustomError.internalServer(`${error}`)
    }
}
```

---

# Node TS File Upload

- Creo la carpeta presentation/file-upload con su controlador y sus rutas
- controller

```
import { Request, Response } from "express"

export class FileUploadController{

    constructor(){}

    private handleError = (error:unknown, res: Response)=>{
        if(error instanceof Error){
            return res.status(400).json({error: error.message})
        }
        res.status(500).json({error: 'Internal Server Error'})
    }

    public uploadFile=(req:Request, res: Response)=>{
        res.json({message: 'File uploaded successfully'})
    }
}
```

- routes

```
import { Router } from 'express';
import { getEnabledCategories } from 'trace_events';
import { AuthMiddleware } from '../middlewares/auth.middleware';
import { FileUploadController } from './file-upload.controller';


export class FileUploadRoutes {


  static get routes(): Router {
```

```
    const fileUploadController = new FileUploadController()
    const router = Router();


    router.post('/', [AuthMiddleware.validateJWT],
fileUploadController.uploadFile)

    return router;
  }
}
```

- En AppRoutes

```
import { Router } from 'express';
import { AuthRoutes } from './auth/routes';
import { CategoryRoutes } from './categories/routes';
import { FileUploadRoutes } from './file-upload/routes';


export class AppRoutes {


  static get routes(): Router {

    const router = Router();

    router.use('/api/auth', AuthRoutes.routes)
    router.use('/api/categories', CategoryRoutes.routes)
    router.use('/api/file-upload', FileUploadRoutes.routes)

    return router;
  }
}
```

- (Faltan webhooks y websockets)