Patrones de Diseño

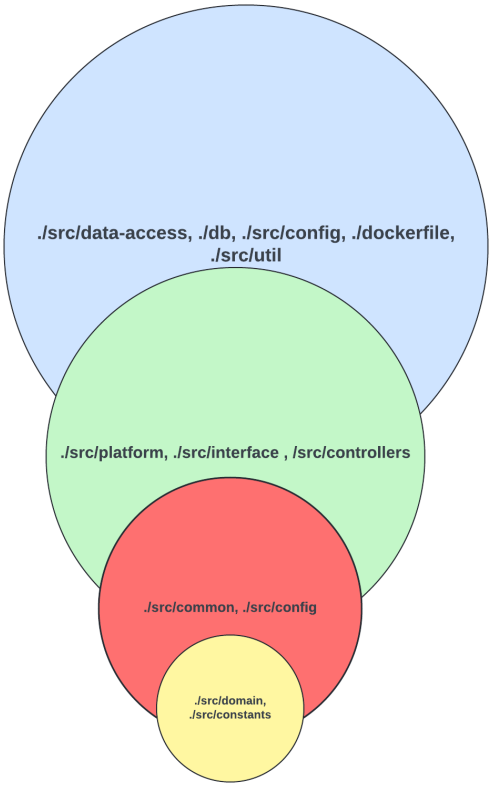
Arquitectura de Software

Proyecto Final

Bryan Gersain Bonilla Nandayapa

Nathalia Dos Santos Bruggemann

1. Clean Architecture



a) Enterprise Bussiness Rules

Encargada de implementar los atributos esenciales de un negocio, es lo que genera ingresos y es independiente de todas las otras capas de arquitectura. Debido a su independencia con otras capas estas no influyen o afectan a esta, es por ello que crea sus propias interfaces y useCases y es invariable su estructura en toda la aplicación. La carpeta ‘./src/domain’ tiene sus propias interfaces y UseCases además de encapsular las dinámicas de userRepository y el servicio de autenticación en la clase de sign in. También define la estructura de un usuario. Por su parte la carpeta ‘./src/constants’ es una serie de constantes aplicables en toda la arquitectura y es invariable.

b) Aplication Bussiness Rules

Debe de proveer cada UseCase necesario para la funcionalidad de la aplicación, determina que controller o Gateway debe ser llamado y los distintos módulos son coordinados para actuar correctamente. En primera instancia se reconoce aquí los servicios de autenticación y cache, uno sirve para asegurarse de que se están encontrando los datos correctos y otro de transportar la información de manera rápida entre las distintas capas de la aplicación, en siguiente, se ubican las rutas que accedan a los métodos de signin y signup, verificando que se estén implementando los UseCases corrrectos. En el caso de ‘./src/config’ se ocupa de proporcionar los accesos necesarios para las operaciones CRUD sobre Redis y SQL además de implementar la biblioteca Inversify para la inversión de dependencias de la aplicación.

c) Interface Adapters

Contiene los métodos que precisa de implementar el UI o la Web para mostrar información o generarla, sostiene las operaciones CRUD que son realizadas por la aplicación e implementadas por la base de datos. También a esta capa se agregan los controllers que cumplen la función de actuar como mediador entre las distintas capas y su entidad correspondiente. Además, también se encuentra aquí ‘./src/interface’ que establece plantillas para errores y la generación de JSON’s. Otro elemento importante es la carpeta ‘./src/platform’ que contiene el server que es encargado de procesar las respuestas y peticiones del cliente.

d) Frameworks and Drivers

Finalmente, esta capa se encarga de lo más externo en la aplicación: la interfaz de usuario, las bases de datos y las requests derivadas de esta parte de la aplicación. Es por eso que aquí se

cree que se puede ubicar la carpeta ‘./src/data-access’ que contiene las queries para el repositorio del usuario. Además se coloca la carpeta ‘./db’ correspondiente a la base de datos, además de ‘./dockerfile’ encargado del deploy de la aplicación, también se coloca a ‘./src/util’ donde se ubican las promises necesarias para el UI, por último, ‘./src/routes’ también se ubica aquí porque permitirá la interacción CRUD de la interfaz de usuario.

2. Clean Code

2.1. Índice de carpetas

a. common.....

Descripción:

Contiene servicios como AuthService y CacheService, que encapsulan dicha lógica respectivamente. Los cuales coordinan y aplican la lógica de negocio utilizando los casos de uso y entidades del dominio.

a.1.- auth.....

a.1.1.- auth-service.ts

```
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';

import { config } from "@example-api/platform/index";
import { IAuthService } from './AuthService';
import { injectable } from 'inversify';
...
@injectable()
export class AuthService implements IAuthService {
  generateToken(data: any): any {
    return jwt.sign({ data }, config.auth.secret, {
      expiresIn: "24h",
    });
  }

  verifyToken(token: string): any {
    try {
      return jwt.verify(token, config.auth.secret);
    } catch (err) {
      return false;
    }
  }

  async matchPassword(password, hash) {
    return await bcrypt.compare(password, hash);
  }

  hashPassword(plainPassword): string {
    const saltRounds = config.auth.salts;
    const salt = bcrypt.genSaltSync(saltRounds);
    const hash = bcrypt.hashSync(plainPassword, salt);

    return hash;
  }
}
```

Principio	Justificación de uso
Dependency Inversion	La clase AuthService implementa la interfaz IAuthService y utiliza el decorador "@injectable" que insinúa una inyección de dependencias, esto permite el desacoplamiento de módulos que facilita también la realización de pruebas unitarias.
Pilar	Justificación de uso
Polimorfismo	La clase AuthService se va a comportar de manera similar a todas las clases que implementen la interfaz IAuthService, ya que contiene los mismos métodos, aunque tenga datos diferentes en cada uno.

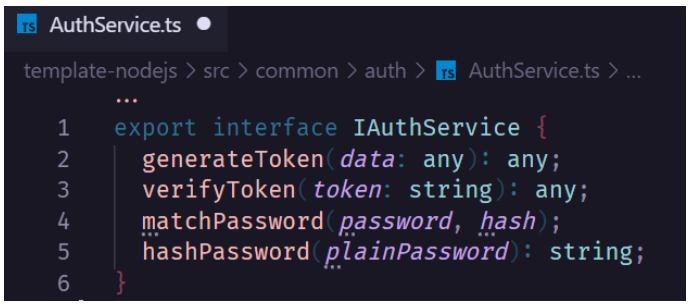
Descripción:

Utiliza jwt y bcrypt para la generación y verificación de tokens, cifrado de contraseñas. Lo cual podría considerarse parte de la capa de Infraestructura.

Singleton:

Se observa que la clase AuthService está anotada con @injectable, indicando su gestión por un contenedor de Inversión de Control (IoC), lo cual sugiere que se espera que AuthService sea una única instancia gestionada en el sistema.

a.1.2.- AuthService.ts



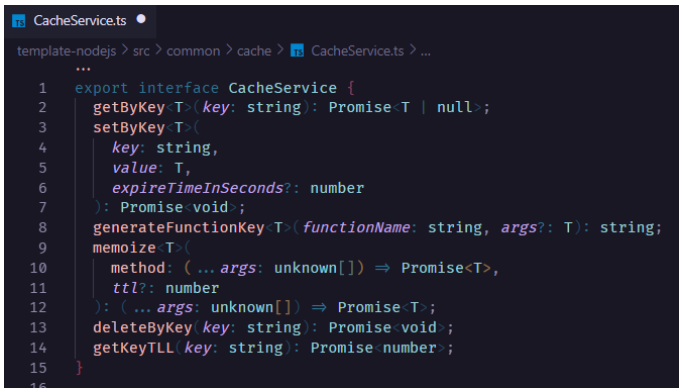
Pilar	Justificación de uso
Abstracción	La interfaz IAuthService abstrae de manera concreta en cada uno de sus atributos los servicios que necesita proporcionar un servicio de autenticación.

Mejoras sugeridas:

Implementar una mejor segregación de interfaces para no sobrecargar una sola interfaz con todas las tareas correspondientes al servicio de autenticación, esto también ayudaría a manejar un principio de Single Responsability en las clases que implementen esta interfaz como lo es AuthService en auth-service.ts.

a.2.- cache.....

a.2.1.- CacheService.ts



Pilar	Justificación de uso
Encapsulación	Solo se puede acceder a los atributos del objeto por medio de métodos designados como lo es getByKey, setByKey, generateFunctionKey, etc.
Abstracción	La interfaz CacheService logra manejar en sus atributos todas las tareas que requiere un servicio de Cache, esto permite su fácil implementación en las clases que requieran de dichos atributos o funcionalidades.

Mejoras sugeridas:

También se sugiere una segregación de interfaz mejor implementada para no sobrecargar una sola, esto evitará errores y hará que la implementación de la misma sea más fácil.

b. config.....

Descripción:

Tal vez en algún caso futuro de que el proyecto siga creciendo separar las responsabilidades de la capa de Infraestructura en módulos o carpetas diferentes (por ejemplo, una carpeta dedicada a la configuración de la DB y otra para la configuración de Redis).

b.1.-db.....

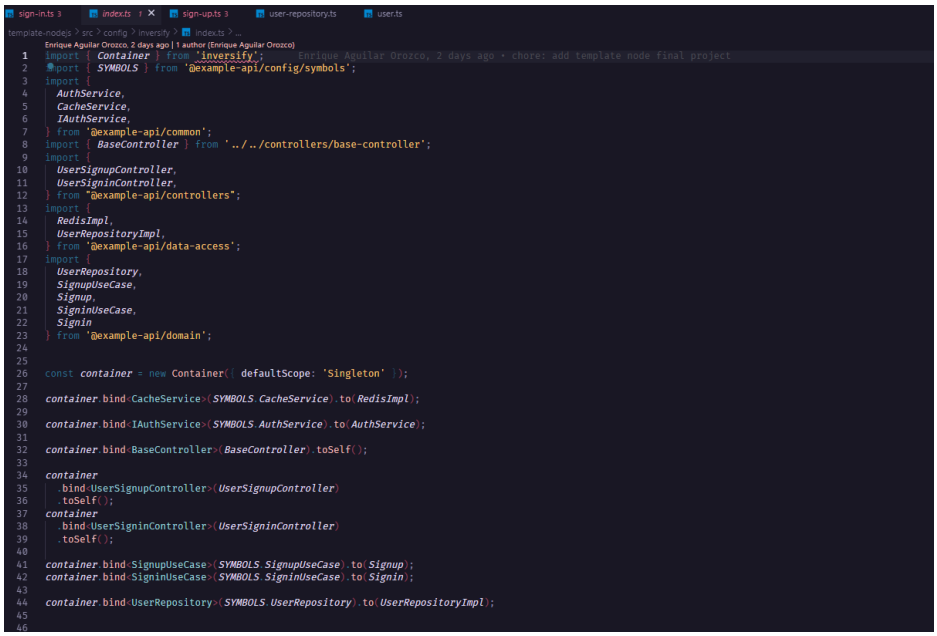
Descripción:

En los archivos de la carpeta DB contiene la implementación concreta para interactuar con el archivo Redis y la DB mediante el archivo SQL.

Estas implementaciones pertenecen a la capa de Interface Adapters, ya que manejan los detalles técnicos de acceso a las fuentes de datos.

b.2.- Inversify.....

b.2.1.- index.ts



Descripción:

Se utiliza la biblioteca inversify para implementar la inyección de dependencias y mantener un desacoplamiento entre los componentes de la arquitectura del programa, esto permite que las distintas clases pueda implementar la inversión de dependencias y que la segregación de interfaces pueda ser implementada de manera segura.

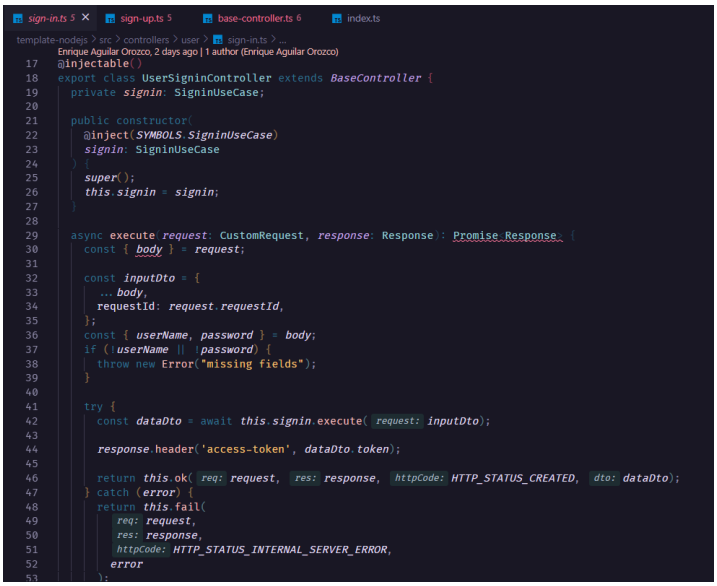
c. constants.....

Principio	Justificación de uso
Single Responsability	Cada archivo dentro de la carpeta Constants tiene una responsabilidad bien definida, como definir códigos de error, mensajes de error, constantes relacionadas con el juego o configuraciones de Redis, cumpliendo con este principio.

d. controllers.....

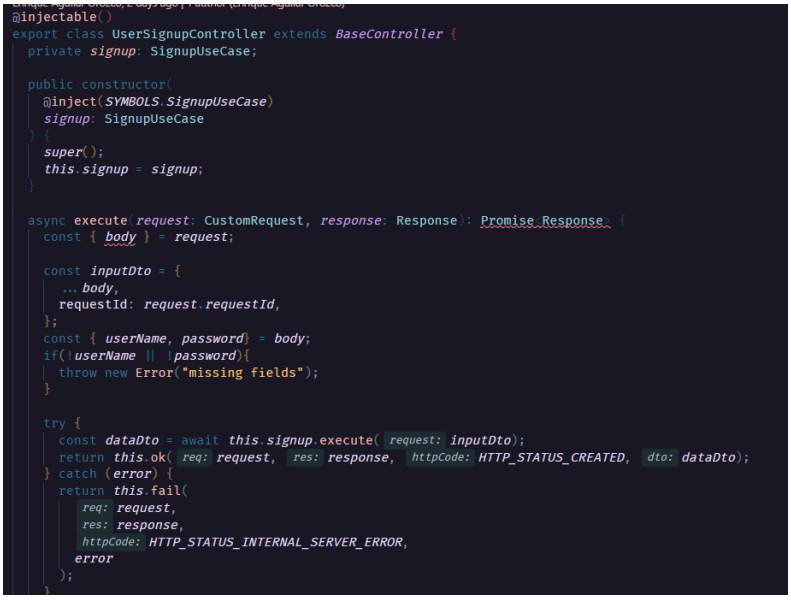
d.1.- user.....

d.1.2.- sign-in.ts



Principio	Justificación de uso
Dependency Inversion	En este caso al utilizar SigninUseCase al utilizar inyección de dependencias y depender de dicha abstracción, se cumple con el principio correspondiente.
Pilar	Justificación de uso
Encapsulación	Se observa el uso de variables privadas de y la clase contiene sus métodos de acceso públicos para manejar las solicitudes HTTP de inicio de sesión.
Herencia	La clase SigninController extiende de la clase BaseController heredando su estructura.

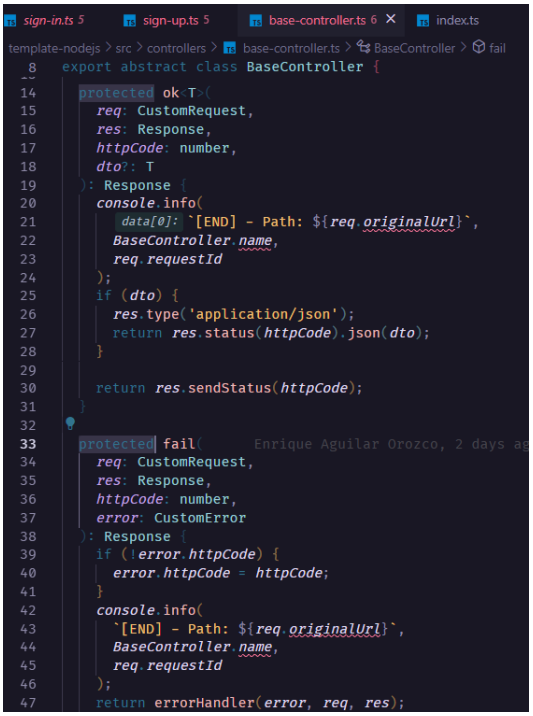
d.1.2.- sign-up.ts



```
@injectable()
export class UserSignupController extends BaseController {
  private signup: SignupUseCase;

  public constructor(
    @inject(SYMBOLS.SignupUseCase)
    signup: SignupUseCase
  ) {
    super();
    this.signup = signup;
  }

  async execute(request: CustomRequest, response: Response): Promise<Response> {
    const { body } = request;

    const inputDto = {
      ...body,
      requestId: request.requestId,
    };
    const { userName, password} = body;
    if(!userName || !password){
      throw new Error("missing fields");
    }

    try {
      const dataDto = await this.signup.execute( request: inputDto);
      return this.ok( req: request,  res: response,  httpCode: HTTP_STATUS_CREATED,  dto: dataDto);
    } catch (error) {
      return this.fail(
        req: request,
        res: response,
        httpCode: HTTP_STATUS_INTERNAL_SERVER_ERROR,
        error
      );
    }
```

Pilar	Justificación de uso
Encapsulación	Declara instancias privadas como lo es *sigin*, y encapsula la lógica de la clase en los métodos heredados.
Herencia	Extiende de la clase abstracta BaseController lo que le permite heredar los atributos de la misma.
Principio	Justificación de uso
Dependency Inversion	En este caso también se utiliza SigninUseCase al utilizar inyección de dependencias y depender de dicha abstracción, se cumple con el principio correspondiente.

d.2.- base-controller.ts



Pilar	Justificación de uso
Abstracción	La clase abstracta BaseController establece un modelo que maneja el flujo de información entre la entidad del servidor y cliente, esta se implementa en las clases UserSigninController y UserSignupController respectivamente.

Notas: Se encontró la implementación de diseño controller, más que definir estructuras de clases, el patrón controller proporciona una estructura organizativa y de flujo de control para la aplicación. Los archivos sign-in y sign-up actúan como intermediarios entre las solicitudes HTTP en los casos de uso del Domain. Se emplea para encapsular la lógica asociada a la manipulación de solicitudes HTTP y la generación de respuestas. En este contexto, funciona como un controlador base del cual otros controladores pueden heredar funcionalidades.

Mejoras sugeridas:

Sería adecuado separar la lógica de manejo de errores y respuestas HTTP en una clase o módulo dedicado, en lugar de tenerla en la clase BaseController.

e. data-access.....

e.1.- cache.....

e.1.1.- redis-impl.ts

```
user-repository.ts 3   redis-impl.ts 6
template-nodejs > src > data-access > cache > redis-impl.ts > RedisImpl > memoize > <function>
export class RedisImpl implements CacheService {
  async getKeyTLL(key: string): Promise<number> {
    const client = getRedisClient();
    return await client.ttl(key);
  }

  async getByKey<T>(key: string): Promise<T | null> {
    const client = getRedisClient();
    const response = await client.get(key);
    if (!response) {
      return null;
    }
    try {
      return JSON.parse( text: response);
    } catch (err) {
      return response as unknown as T;
    }
  }

  async setByKey<T>(key: string, value: T, seconds?: number): Promise<void> {
    const expireTimeInSeconds = seconds ?? 20;
    const client = getRedisClient();
    await client.set(key, JSON.stringify(value), 'EX', expireTimeInSeconds);
  }

  memoize<T>(
    method: (...someArgs: unknown[]) => Promise<T>,
    ttl?: number
  ): (...someArgs: unknown[]) => Promise<T> {
    return async (...args) => {      Enrique Aguilar Orozco, 2 days ago • chore: add templa
      const recordKey = this.generateFunctionKey( functionName: method.name, args);
      const record = await this.getByKey<T>( key: recordKey);
      if (record && typeof record === 'string') {
        try {
          return JSON.parse( text: record);
        } catch (err) {
          return record;
        }
      } else if (record) {
        return record;
      }

      const response = await method.apply( this: this,  thisArg: args);

      if (response) {
        const responseForRedis = JSON.stringify( value: response);
        await this.setByKey( key: recordKey,  value: responseForRedis,  seconds: ttl);
      }

      return response;
    };
  }

  generateFunctionKey<T>(functionName: string, args?: T): string {

    if (Array.isArray( arg: args) && args.length) {
      return `${functionName}-${JSON.stringify(args)}`;
```

Pilar	Justificación de uso
Encapsulación	Los atributos de la clase RedisImpl son únicamente accesibles por medio de los métodos estipulados dentro de la misma.
Polimorfismo	RedisImpl se comporta de la misma manera que CacheService, que es la interfaz implementada en la clase, sin tener los mismos datos.

Principio	Justificación de uso
Dependency Inversion	Permite la inyección de dependencias por medio del decorador “injectable()”, además la clase RedisImpl implementa una interfaz que permite también el desacoplamiento y dependencia entre clases.

Mejoras sugeridas:

Se podría considerar separar las implementaciones de los repositorios en módulos o carpetas diferentes según el dominio o la fuente de datos. Por ejemplo, tener una carpeta para los repositorios relacionados con la base de datos (DB) y otra para los relacionados con Redis.

Abstracción del Acceso a los Datos:

Aunque el archivo user-repository, a través de la clase UserRepositoryImpl, implementa el patrón Repository, aún contiene parte de la lógica de acceso a la base de datos. Esto hace que esté estrechamente acoplado con los detalles de implementación de las consultas SQL.

e.2.- user.....

e.2.1.- user-repository.ts

```
import { injectable } from "inversify";
import { sql } from "@example-api/config/db";
import { UserRepository } from "@example-api/domain";
import { User } from "src/domain/user/user";

Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
@injectable()
export class UserRepositoryImpl implements UserRepository {

  async create(data: User): Promise<void> {
    const { userName, password } = data;
    await sql({ query: {
      query: `
        INSERT INTO
          user_account(usr_act_name, usr_act_password)
        VALUES($1,$2)
      `,
      bind: [userName, password],
    });
  }

  async getUserByUserName(userName: string): Promise<any> {
    const results = await sql({ query: {
      query: `
        SELECT
          usr_act_id as "userId",
          usr_act_name as "userName",
          usr_act_password as password
        FROM user_account
        where usr_act_name = $1
      `,
      bind: [userName],
    }) as any;

    if (results.rowCount) {
      return results.rows[0];
    }

    return null;
  }
}
```

Pilar	Justificación de uso
Encapsulación	Las propiedades de la clase UserRepositoryImpl están encapsuladas en métodos mediante los cuales se pueden construir y consultar datos.
Polimorfismo	UserRepositoryImpl implementa la interfaz UserRepository manteniendo sus métodos y permitiendo que se comporte de la misma manera sin contener los mismos datos.
Principio	Justificación de uso
Dependency Inversion	UserRepositoryImpl implementa la interfaz UserRepository en lugar de depender de otra clase, o, hacer que otras dependan de ella, lo que permite el desacoplamiento de módulos.

Repository:

El archivo user-repository dentro de esta carpeta implementa el patrón de diseño Repository a través de la clase UserRepositoryImpl. Este patrón aísla la capa de datos del resto de la aplicación (es decir, la interfaz de usuario), actuando como una abstracción para el acceso a los datos relacionados con los usuarios y encapsulando la lógica de interacción con la base de datos.

f. domain.....

f.1.- user.....

f.1.1.- sign-in.ts

```
Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
interface requestDto {
  requestId: string;
  userName: string;
  password: string;
}

Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
interface responseDto {
  token: string;
}

export type SigninUseCase = UseCase<requestDto, responseDto>;

Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
@injectable()
export class Signin implements SigninUseCase {
  private readonly userRepository: UserRepository;
  private readonly authService: AuthService;

  public constructor(
    @inject(SYMBOLS.UserRepository) userRepository: UserRepository,
    @inject(SYMBOLS.AuthService) authService: AuthService
  ) {
    this.userRepository = userRepository;
    this.authService = authService;
  }

  public async execute(requestDto: requestDto): Promise<responseDto> {
    try {
      const { userName, password } = requestDto;
      const user = await this.userRepository.getUserByUserName(userName);
      await this.authService.matchPassword(password, hash: user.password);
      const generateToken = this.authService.generateToken( data: user);

      return Promise.resolve({
        token: generateToken,
      });
    } catch (error) {
      throw new Error(error?.message);
    }
  }
}
```

Pilar	Justificación de uso
Encapsulación	Este código hace un buen uso de la encapsulación al definir las propiedades y métodos privados de la clase Signin.
Polimorfismo	Implementa SignInUseCase, extendiendo únicamente un constructor a la interfaz, se comporta de manera similar a la otra clase derivada de SignInUseCase que es SignUp.
Principio	Justificación de uso
Interface Segregation	Este código implementa el principio de Inversión de Control (IoC) e Inyección de Dependencias (DI) al usar la librería "inversify" y decoradores como @inject y @injectable.

Descripción:

Aquí implementamos el patrón a través de la clase Sign In. Esta clase encapsula la lógica de negocio relacionada con los casos de uso del inicio de sesión.

Este código implementa el principio de Inversión de Control (IoC) e Inyección de Dependencias (DI) al usar la librería "inversify" y decoradores como @inject y @injectable.

Strategy:

La interfaz SigninUseCase establece un contrato para el inicio de sesión, mientras que la clase Signin implementa este contrato y ofrece una estrategia específica para llevar a cabo el inicio de sesión.

f.1.2.- sign-up.ts

Pilar	Justificación de uso
Encapsulación	Encapsula en propiedades privadas de solo lectura las interfaces de userRepository y authService que impide la modificación de sus propiedades, además, contiene la lógica de la clase contenida en métodos específicos.
Polimorfismo	Adquiere los mismos métodos que la clase CacheService y por ende se comporta de manera similar teniendo distintos datos.
Principio	Justificación de uso
Interface Segregation	Maneja interfaces separadas para manejar las peticiones y respuestas, esto permite la disminución de errores y sea más fácil manejar pruebas.

Caso de Uso:

Aquí implementamos el patrón a través de la clase Signup. Esta clase encapsula la lógica de negocio relacionada con los casos de uso del registro de usuarios.

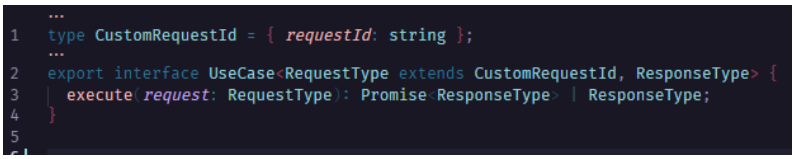
f.1.3.- user-repository

Pilar	Justificación de uso
Abstracción	La interfaz define la función básica de crear y leer datos de una BD, esto de manera concreta y estructurada.
Principio	Justificación de uso
Interface Segregation	Permite que las clases que la implementen solo hereden los métodos que les conciernen y en este caso sirve precisamente como punto de interacción con los datos registrados en una BD.

Repository:

Este archivo define una interfaz UserRepository, que representa la abstracción del acceso a los datos relacionados con los usuarios.

f.2.- UseCase.ts

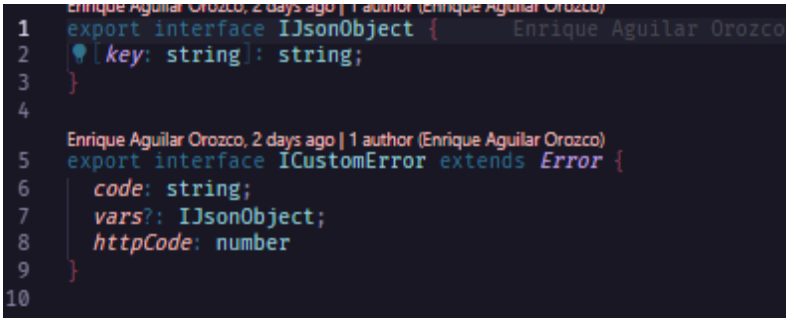


```
...
type CustomRequestId = { requestId: string };
...
export interface UseCase<RequestType extends CustomRequestId, ResponseType> {
  execute(request: RequestType): Promise<ResponseType> | ResponseType;
}
```

Pilar	Justificación de uso
Abstracción	La interfaz UseCase abstrae los elementos principales que componen las tareas que realiza un use case y esto se refiere a que maneja las interacciones entre los distintos miembros de la arquitectura en este caso definidas por RequestType y ResponseType.
Herencia	RequestType extiende del tipo CustomRequestId, heredando el atributo requestId.
Principio	Justificación de uso
Interface Segregation	La interfaz UseCase contiene solo los elementos que conciernen a la tarea que busca implementar y lo hace de manera compacta.

g. interfaces……

g.1.-custom-error.ts



Pilar	Justificación de uso
Abstracción	Ambas interfaces se componen de atributos concretos que las definen, la interfaz IJsonObject habla de un par key/value que podrá ser implementado y extendido de ser necesario. Por su parte, ICustomError establece la estructura de un error.
Herencia	ICustomError extiende de la interfaz Error definida en la librería de typescript y hereda la misma estructura.
Principio	Justificación de uso
Interface Segregation	Ambas interfases son concretas y cerradas a su tarea o razón de ser en la arquitectura.

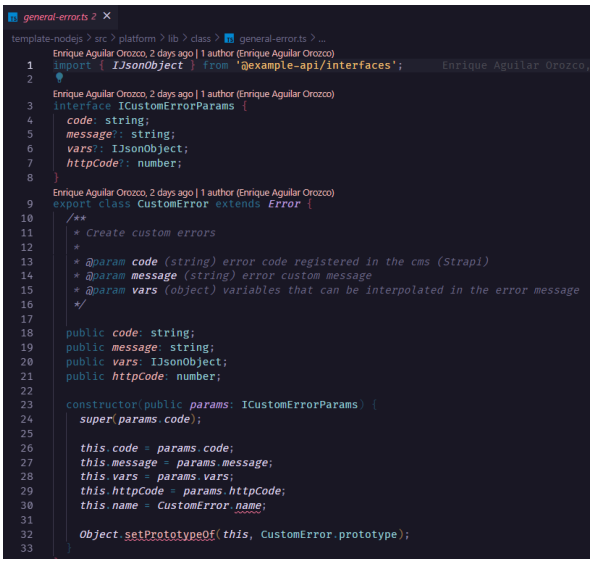
Mejoras sugeridas:

Deberíamos considerar separar los casos de uso en módulos o carpetas diferentes según el dominio o la funcionalidad. Por ejemplo, tener una carpeta para los casos de uso relacionados con usuarios y otra para los casos de uso relacionados con productos. Aunque ya definimos UserRepository, no se observa la definición de interfaces o clases abstractas para Signin y Signup. La inclusión de estas promovería la modularidad y facilitaría las pruebas.

h. platform.....

h.1.- lib.....

h.1.1.- general-errors.ts



```
general-errors.ts 2 ×
template-nodejs > src > platform > lib > class > general-errors.ts > ...
Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
1  import { IJsonObject } from '@example-api/interfaces';
2
Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
3  interface ICustomErrorParams {
4    code: string;
5    message?: string;
6    vars?: IJsonObject;
7    httpCode?: number;
8  }
Enrique Aguilar Orozco, 2 days ago | 1 author (Enrique Aguilar Orozco)
9  export class CustomError extends Error {
10   /**
11    * Create custom errors
12    *
13    * @param code (string) error code registered in the cms (Strapi)
14    * @param message (string) error custom message
15    * @param vars (object) variables that can be interpolated in the error message
16    */
17
18   public code: string;
19   public message: string;
20   public vars: IJsonObject;
21   public httpCode: number;
22
23   constructor(public params: ICustomErrorParams) {
24     super(params.code);
25
26     this.code = params.code;
27     this.message = params.message;
28     this.vars = params.vars;
29     this.httpCode = params.httpCode;
30     this.name = CustomError.name;
31
32     Object.setPrototypeOf(this, CustomError.prototype);
33   }
```

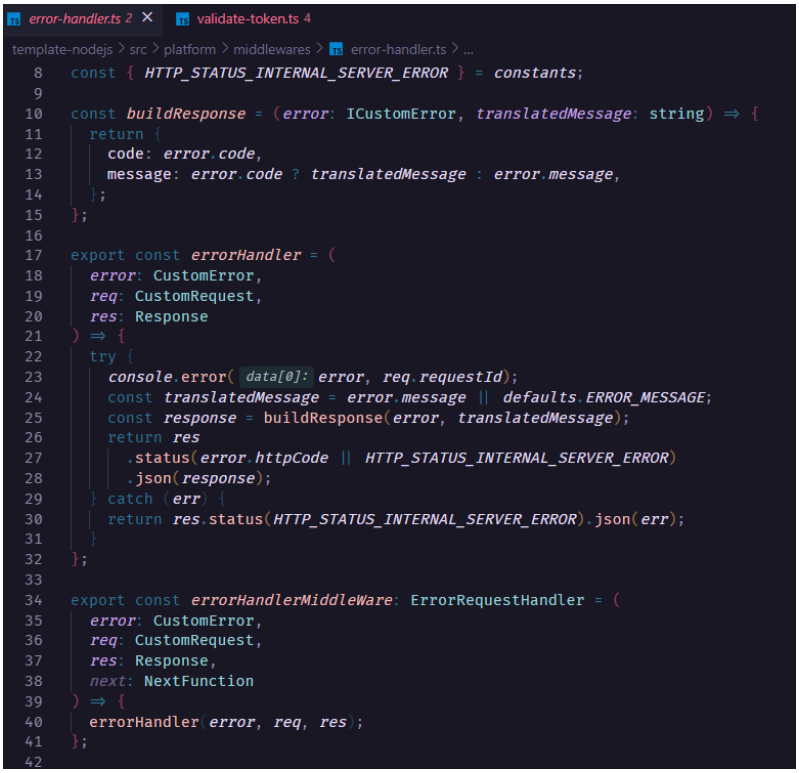
Pilar	Justificación de uso
Abstracción	Construye una estructura para presentar un error implementando interfaces para su constitución que ejecutará su tarea especifica y esa es la de hacer conocer errores.
Herencia	CustomError extiende de Error una interfaz nativa de typescript, extiende más métodos y atributos de esta, pero conserva los atributos heredados de manera principal.
Principio	Justificación de uso
Interface Segregation	CustomError únicamente implementa la interfaz que ocupa y esta es concisa y permite el desacoplar tareas de la clase principal de este archivo.

Mejoras sugeridas:

Hacer que el constructor de CustomError sea privado para prevenir el acceso por medio de el operador *new* y añadir un método estático para obtener la instancia de clase de una manera más controlada.

h.2.- middlewares.....

h.2.1.- error-handler.ts



Descripción:

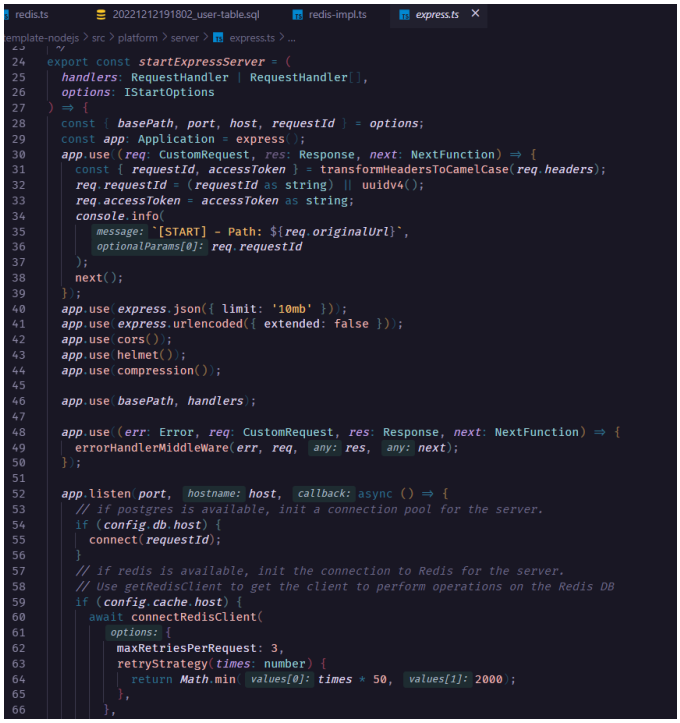
Encapsula la lógica de la gestión de errores que se ejecuta durante el procesamiento de solicitudes HTTP.

Chain of Responsibility:

En esencia, permitimos que múltiples middlewares manejen errores en una solicitud de forma secuencial.

h.3.- server.....

h.3.1.-express.js



Singleton:

Implementamos una única instancia de configuración y servicios mediante el uso de connectRedisClient y connect. Este enfoque garantiza que determinadas clases dispongan de una única instancia compartida en toda la aplicación.

Factory Method:

Las funciones connectRedisClient y connect funcionan como fábricas para crear y configurar instancias de Redis y SQL, respectivamente.

Facade:

La función startExpressServer simplifica la inicialización y configuración del servidor, al tiempo que oculta los detalles complejos de su configuración interna.

h.3.2.- types.ts

Pilar	Justificación de uso
Abstracción	Cada interfaz tiene atributos propios de su tarea a realizar o razón de existencia en la arquitectura. CustomRequest permite el acceso y personalización de una request nativa de typescript, mientras que IStartOptions establece sus atributos generales.
Herencia	CustomRequest hereda de Request, que es una interfaz nativa a express, su misma estructura de atributos.
Principio	Justificación de uso
Interface Segregation	Las interfaces están segregadas con sus respectivas tareas y atributos a contener y heredar a otras clases de ser necesario.

Mejoras sugeridas:

Para mejorar la gestión de errores, se sugiere refinar su manejo y proporcionar mensajes más detallados para una mejor comprensión. Asimismo, se recomienda fortalecer la validación de datos de entrada utilizando bibliotecas como Joi.

Si bien el uso de medidas de seguridad como JWT y su configuración son positivas, se podría considerar la implementación de medidas adicionales como CORS y CSRF para fortalecer la protección contra posibles ataques y validar los orígenes de manera efectiva.

i. routes.....

j. util.....

Mejoras sugeridas:

Evitar dejar archivos vacíos por limpieza al proyecto (Index.ts).

La función sequentialPromises ejecuta promesas de manera recursiva secuencialmente.
Pero si una de las promesas falla, la función continúa ejecutando y no maneja los errores.

Sugerencias y Observaciones Generales

- Realizar separación de responsabilidades en la capa Platform. Actualmente, contiene demasiados componentes diferentes, como configuración, middlewares y servidor web.
- Revisar la consistencia en la nomenclatura de archivos y carpetas, ya que se observan diferentes convenciones (snake_case, kebab-case, PascalCase).
- Destacamos la necesidad de una capa de Aplicación o Servicio intermedia entre los Controladores y el Dominio para encapsular la lógica de coordinación de casos de uso.
- Implementación de los patrones de diseño facade y mediator, para simplificar la interacción entre los componentes