



Universidad Autonoma de Queretaro

Facultad de Informatica

Patrones de Diseño

Alumno: Diego Octavio Nieves Terrazas

307047

Fecha de entrega: 18 de mayo de 2024

Análisis Patrones de Diseño

A lo largo de este documento analizaremos distintos códigos en busca de buenas practicas como lo son el uso de Clean architecture, pilares de la programación orientada a objetos, así como patrones de diseño vistos a lo largo del Taller y finalmente la identificación de principios SOLID. Todo esto como proyecto final del Taller llamado Patrones de Diseño. Para esto dividiremos el documento en dos secciones principales, las cuales estarán divididas en Lógica de aplicación y lógica de negocio.

Lógica de Aplicación

Análisis archivo “auth-service”

```
@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;
  }
}

export interface IAuthService {
  generateToken(data: any): any;
  verifyToken(token: string): any;
  matchPassword(password, hash);
  hashPassword(plainPassword): string;
}
```

En este apartado principalmente es donde pude identificar el uso de una configuración externa, accediendo a variables sensibles desde otro archivo sin exponer estos datos en texto plano de la aplicación.

Por otro lado en los pilares de la programación orientada a objetos, encontramos el encapsulamiento, ya que en AuthService encapsula todas las operaciones relacionadas con la autenticación como lo son la generación, verificación y comparación de Tokens, así como el hash de contraseñas .

También contamos con abstracción donde en AuthService se implementa la interfaz "IAuthService" abstrayendo detalles del manejo de tokens.

Principios SOLID:

Single Responsibility Principle (SRP): AuthService tiene una única responsabilidad: manejar la autenticación.

Open/Closed Principle (OCP): AuthService está cerrada para modificaciones pero abierta para extensión a través de IAuthService.

Patrones de diseño:

Adapter Pattern: AuthService actúa como un adaptador que oculta la implementación específica de JWT y bcrypt, proporcionando una interfaz uniforme para la autenticación.

```
import crypto from 'crypto';

const SEM_VERSION_REGEX = '^((\\d+\\.\\.?)?(\\d+\\.\\.?)?(\\d+)$)';

export const isValidVersion = (version: string) => {
  return version.match( regexp: SEM_VERSION_REGEX);
};

/**
 * kebab-case to UpperCamelCase
 * @param {String} word
 * @return {String}
 */
export const toCamelCase = (word: string): string =>
  word.replace( searchValue: /-./g, replacer: w => w[1].toUpperCase());

export const transformHeadersToCamelCase = (headers: {
  [k: string]: string | string[] | undefined;
}): { [k: string]: string | string[] } =>
  Object.entries(headers).reduce((r, [k, v]) => {
    return {
      ...r,
      [toCamelCase(k)]: v,
    };
  }, {});

export const hashKeyFn = <T>(data: T): string => {
  const objectToHash = JSON.stringify( value: data);
  return crypto.createHash('md5').update(objectToHash).digest('hex');
};

export const urlParams = (base: string, objectParams: {
  [k: string]: string;
}): string => {
  const params = new URLSearchParams(objectParams).toString();
  return base ? `${base}?${params}` : null;
}
```

```
export interface Event {
  evt_id: number;
}
```

```
export { CacheService } from './cache/CacheService';
export { AuthService } from './auth/auth-service';
export * from './util';
export * from './types';
export * from './auth/AuthService';
```

Ahora pasamos con “útil” en donde respecto a clean architecture encontramos que cada función tiene una responsabilidad perfectamente definida. En cuanto a pilares de POO encontramos la abstracción principalmente al generar hashes y transformar cadenas.

Los principios SOLID que pude detectar fueron Single Responsibility Principle donde toCamelCase transforma la cadena justamente a CamelCase y ‘hashKeyFn’ genera un hash, es decir ambas funciones contienen una única función, así mismo podemos ver el Open/Closed principle ya que las funciones están cerradas para la modificación pero pueden ser extendidas por nuevas funciones.

Desgraciadamente no pude encontrar algún patrón de diseño de los vistos en clase.

```

let redisClient;

export function getRedisClient(): Redis.Redis {
  return redisClient;
}

export async function connectRedisClient(
  options?: Redis.RedisOptions,
  requestId?: string | null
): Promise<void> {
  let opts = {
    port: +config.cache.port,
    host: config.cache.host,
    password: config.cache.password,
  };
  redisClient = new Redis(opts);

  redisClient.on("connect", () => {
    console.info(
      data[0]: "connect-redis-client",
      data[1]: requestId || "not requestId was provided",
    );
  });
  redisClient.on("error", async (error) => {
    console.error(
      data[0]: error,
      data[1]: "connect-redis-client",
      data[2]: requestId || "not requestId was provided",
      data[3]: "cache-sdk"
    );
  });
}

```

En cuanto a clean architecture encontramos una separación de preocupaciones, donde se hace pir seoarada la conexión y la configuración del cliente Redis.

En cuanto a pilares de la POO, encontramos abstracción ya que a través de las funciones permite interactuar con el cliente Redis sin exponer la configuración y la lógica interna de conexión.

Del mismo modo en los principios Solid tenemos el Single Responsibility donde cada función tiene una única responsabilidad Y también podemos ver el Dependency Inversion Principle ya que La función connectRedisClient depende de la abstracción Redis.RedisOptions para las opciones de configuración, y no de una implementación concreta.

En cuanto a patrones de diseño nos encontramos con Observer en el manejo de eventos "connect" y "error" para responder a eventos emitidos por el cliente Redis.

```

export declare type DBSQLBind = string | number | (string | number)[];

export type DBSQLArguments = {
  query: string;
  bind?: DBSQLBind;
};

export type DBReplyDataRow = {
  [key: string]: any;
}

let client;

export function connect(requestId?: string | null): void {
  client = new Client({
    host: config.db.host,
    port: config.db.port,
    user: config.db.userName,
    password: config.db.password,
    database: config.db.dbName,
  });
  client.connect((err) => {
    if (err) {
      console.error( data[0]: "connection error", data[1]: err.stack);
    } else {
      console.log( data[0]: "connected");
    }
  });
}

export async function sql({ query, bind }: DBSQLArguments): Promise<DBReplyDataRow[]> {
  return await client.query(query, bind);
}

```

Por la parte de clean architecture tenemos la separación de preocupaciones en donde por un lado tenemos la lógica de la conexión a la base de datos y por otro la ejecución de las consultas.

Pilares de la POO Las funciones connect y sql encapsulan la lógica de conexión y consulta a la base de datos, respectivamente, sin exponer detalles innecesarios, también tenemos Abstracción donde se utiliza tipos (DBSQLArguments y DBReplyDataRow) y funciones para abstraer detalles de la implementación de la base de datos.

```

import {
  UserRepository,
  SignupUseCase,
  Signup,
  SigninUseCase,
  Signin
} from '@example-api/domain';

const container = new Container({ defaultScope: 'Singleton' });

container.bind<CacheService>(SYMBOLS.CacheService).to(RedisImpl);

container.bind<IAuthService>(SYMBOLS.AuthService).to(AuthService);

container.bind<BaseController>(BaseController).toSelf();
💡
container
  .bind<UserSignupController>(UserSignupController)
  .toSelf();
container
  .bind<UserSigninController>(UserSigninController)
  .toSelf();

container.bind<SignupUseCase>(SYMBOLS.SignupUseCase).to(Signup);
container.bind<SigninUseCase>(SYMBOLS.SigninUseCase).to(Signin);

container.bind<UserRepository>(SYMBOLS.UserRepository).to(UserRepositoryImpl);

export { container };

```

Clean architecture: Modularidad: Diferentes componentes (controladores, servicios, repositorios) están organizados en módulos, siguiendo un enfoque modular.

Pilares de la POO: Encapsulamiento: Vemos que cada clase está encapsulada y proporciona su funcionalidad a través de interfaces y clases específicas, también vemos abstracción.

Patrones de diseño usado: Singleton Pattern: vemos que se crea un contenedor de inversify con la configuración { defaultScope: 'Singleton' }, esto significa que cualquier dependencia registrada en este contenedor será una instancia única por defecto.

```

/**
 * Redis uses ttl in seconds
 * Reference: https://redis.io/commands/ttl
 */
export const ONE_MINUTE = 60;
export const ONE_HOUR = ONE_MINUTE * 60;
export const ONE_DAY = ONE_HOUR * 24;
export const ONE_WEEK = ONE_DAY * 7;
export const ONE_MONTH = ONE_DAY * 30;

export const game = {
  status: {
    victory: "VICTORY",
    gameOver: "LOOSER",
  },
  messages: {
    wordInvalid: "Word Invalid",
    initGameRequired: "Init Game Needed",
    gameOver: "Game Over",
  },
};

```

```

export * as redis from './redis';
export * as defaults from './default'
export * from './code-errors'
export * from './game';

```

```

const CODE_ERRORS = {
  UNAUTHORIZED: 'UNAUTHORIZED',
};

export { CODE_ERRORS };

```

Clean Architecture

Separación de Preocupaciones: Los archivos definen constantes y mensajes de error, es decir, se están separando las preocupaciones de configuración y mensajes estáticos del resto de la lógica de la aplicación. Las constantes relacionadas con los tiempos (en segundos) también están separadas y documentadas, lo que facilita su uso y mantenimiento en diferentes partes de la aplicación.

Pilares de la POO

Encapsulamiento: Aunque no se utilizan clases o instancias, las constantes están encapsuladas dentro de sus respectivos objetos y archivos, evitando su modificación accidental desde otras partes de la aplicación.

Abstracción: Las constantes y mensajes abstractos nos permiten manejar la abstracción que facilita el manejo de errores y mensajes en la aplicación sin depender de cadenas de texto específicas.

Principios SOLID

Single Responsibility Principle (SRP): Cada archivo tiene una responsabilidad única: CODE_ERRORS para códigos de error, ERROR_MESSAGE para un mensaje de error genérico, y game para constantes relacionadas con el juego y tiempos en segundos.

Open/Closed Principle (OCP):

Las constantes y mensajes están abiertos a la extensión (se pueden agregar nuevos códigos de error, mensajes de juego, etc.) sin modificar el código existente.


```

TS sign-ints 5
src > controllers > user > TS sign-ints > UserSignInController > constructor
1 import { inject, injectable } from "inversify";
2 import { SYMBOLS } from "@example-api/config/symbols";
3 import { Response } from "express";
4 import { constants } from "http2";
5
6 import { SignInUseCase } from "@example-api/domain";
7 import { CustomRequest } from "@example-api/platform/server/types";
8 import { CustomError } from "@example-api/platform/lib/class/general-error";
9 import { BaseController } from "../base-controller";
10
11 const {
12   HTTP_STATUS_CREATED,
13   HTTP_STATUS_BAD_REQUEST,
14   HTTP_STATUS_INTERNAL_SERVER_ERROR,
15 } = constants;
16
17 @injectable()
18 export class UserSignInController extends BaseController {
19   private signin: SignInUseCase;
20
21   public constructor(
22     @inject(SYMBOLS.SignInUseCase)
23     signin: SignInUseCase
24   ) {
25     super();
26     this.signin = signin;
27   }
28
29   async execute(request: CustomRequest, response: Response): Promise<Response> {
30     const { body } = request;
31
32     const inputDto = {
33       ...body,
34       requestId: request.requestId,
35     };
36     const { userName, password } = body;
37     if (!userName || !password) {
38       throw new Error("missing fields");
39     }
40   }

```

Clean Architecture

Separación de Preocupaciones: La clase `UserSignInController` es la responsable de manejar el flujo de la solicitud de inicio de sesión, delegando la lógica de negocio al `SignInUseCase`. Esto sigue el principio de separar la lógica de negocio (caso de uso) de la lógica de la interfaz (controlador).

Independencia de la Interfaz de Usuario: El controlador se encarga de manejar la interfaz HTTP, pero la lógica de negocio reside en el caso de uso, que podría reutilizarse en otras interfaces de usuario.

Pilares de la POO

Encapsulamiento: Los detalles de cómo se realiza el inicio de sesión están encapsulados dentro del `SignInUseCase`.

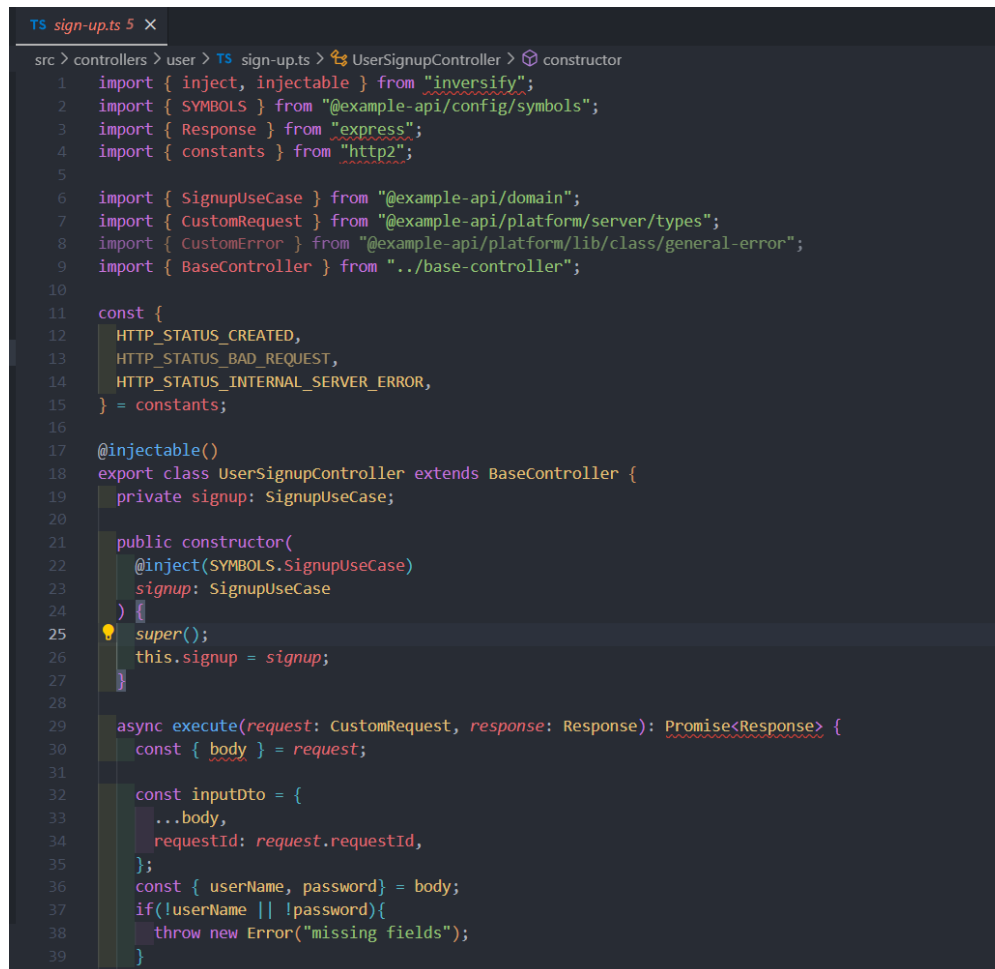
Herencia: `UserSignInController` hereda de `BaseController`, lo que sugiere reutilización de funcionalidad común a todos los controladores.

Principios SOLID

Single Responsibility Principle (SRP): `UserSignInController` tiene la única responsabilidad de manejar solicitudes de inicio de sesión. La lógica de negocio está desarrollada en `SignInUseCase`.

Open/Closed Principle (OCP): El controlador está abierto a la extensión (puede manejar nuevas reglas de negocio en SignupUseCase) pero cerrado a la modificación.

Dependency Inversion Principle (DIP): UserSignupController depende de la abstracción SignupUseCase y no de una implementación concreta, cumpliendo con este principio.



```
TS sign-up.ts 5 X
src > controllers > user > TS sign-up.ts > UserSignupController > constructor
1 import { inject, injectable } from "inversify";
2 import { SYMBOLS } from "@example-api/config/symbols";
3 import { Response } from "express";
4 import { constants } from "http2";
5
6 import { SignupUseCase } from "@example-api/domain";
7 import { CustomRequest } from "@example-api/platform/server/types";
8 import { CustomError } from "@example-api/platform/lib/class/general-error";
9 import { BaseController } from "../base-controller";
10
11 const {
12   HTTP_STATUS_CREATED,
13   HTTP_STATUS_BAD_REQUEST,
14   HTTP_STATUS_INTERNAL_SERVER_ERROR,
15 } = constants;
16
17 @injectable()
18 export class UserSignupController extends BaseController {
19   private signup: SignupUseCase;
20
21   public constructor(
22     @inject(SYMBOLS.SignupUseCase)
23     signup: SignupUseCase
24   ) {
25     super();
26     this.signup = signup;
27   }
28
29   async execute(request: CustomRequest, response: Response): Promise<Response> {
30     const { body } = request;
31
32     const inputDto = {
33       ...body,
34       requestId: request.requestId,
35     };
36     const { userName, password } = body;
37     if(!userName || !password){
38       throw new Error("missing fields");
39     }
40   }
41 }
```

Clean Architecture

Separación de Preocupaciones: UserSignupController se encarga únicamente de manejar las solicitudes de registro de usuario y delega la lógica de negocio al SignupUseCase, además la lógica de negocio no está mezclada con la lógica de la interfaz de usuario, lo que facilita el mantenimiento y la escalabilidad.

Pilares de la POO

Encapsulamiento: La lógica de registro de usuario está encapsulada dentro del SignupUseCase. El controlador simplemente llama al caso de uso y maneja las respuestas y errores.

Abstracción: UserSignupController depende de la abstracción SignupUseCase, lo que facilita cambiar la implementación del caso de uso sin modificar el controlador.

Herencia: UserSignupController hereda de BaseController, sugiriendo que hay métodos y funcionalidades comunes en BaseController que se reutilizan en los controladores derivados.

Polimorfismo: UserSignupController se puede tratar como un BaseController y utilizar los métodos comunes definidos en la clase base.

Principios SOLID

Single Responsibility Principle (SRP): UserSignupController tiene la única responsabilidad de manejar las solicitudes de registro de usuario. La lógica de negocio está delegada al SignupUseCase.

Open/Closed Principle (OCP):

El controlador está abierto a la extensión (puede manejar nuevas reglas de negocio en SignupUseCase) pero cerrado a la modificación.

Dependency Inversion Principle (DIP): UserSignupController depende de la abstracción SignupUseCase y no de una implementación concreta, cumpliendo con este principio.

```

import { injectable } from 'inversify';
import { Request, Response } from 'express';
import { errorHandler } from '@example-api/platform/middlewares/error-handler';
import { CustomError } from '@example-api/platform/lib/class/general-error';
import { CustomRequest } from '../platform';

@injectable()
export abstract class BaseController {
  public abstract execute(
    request: Request,
    response: Response
  ): Promise<Response>;

  protected ok<T>(
    req: CustomRequest,
    res: Response,
    httpCode: number,
    dto?: T
  ): Response {
    console.info(
      data[0]: `[END] - Path: ${req.originalUrl}`,
      data[1]: BaseController.name,
      data[2]: req.requestId
    );
    if (dto) {
      res.type('application/json');
      return res.status(httpCode).json(dto);
    }

    return res.sendStatus(httpCode);
  }

  protected fail(
    req: CustomRequest,
    res: Response,
    httpCode: number,
    error: CustomError
  ): Response {
    if (!error.httpCode) {
      // error.httpCode = httpCode;
    }
  }
}

```

Clean Architecture

Separación de Preocupaciones: BaseController se encarga de proporcionar métodos comunes para el manejo de respuestas HTTP (ok y fail). No contiene lógica de negocio, la cual se delega a las clases concretas que lo extienden.

Independencia de la Interfaz de Usuario: La lógica del controlador no está directamente acoplada a una interfaz de usuario específica. Se centra en la gestión de solicitudes HTTP, que es independiente de la interfaz concreta.

Pilares de la POO

Encapsulamiento: BaseController encapsula la lógica común de manejo de respuestas HTTP (ok y fail), evitando la repetición de este código en cada controlador concreto.

Abstracción: BaseController es una clase abstracta, proporcionando un nivel de abstracción que permite que diferentes controladores específicos implementen su método execute mientras reutilizan la lógica común.

Las clases concretas heredan de BaseController, aprovechando la funcionalidad común definida en la clase base.

Principios SOLID

Single Responsibility Principle (SRP): BaseController tiene la única responsabilidad de proporcionar métodos comunes para manejar respuestas HTTP, como ok y fail.

Open/Closed Principle (OCP): BaseController está abierta a la extensión (las clases concretas pueden extenderla y añadir nueva funcionalidad) pero cerrada a la modificación.

Liskov Substitution Principle (LSP): Las subclases de BaseController pueden sustituir a la clase base sin alterar el comportamiento esperado del programa.

Dependency Inversion Principle (DIP): BaseController depende de abstracciones (Request, Response, CustomRequest) en lugar de implementaciones concretas.

```
TS redis-impl.ts 6 X
src > data-access > cache > TS redis-impl.ts > RedisImpl > getByKey
1 import { injectable } from 'inversify';
2 import { getRedisClient } from '@example-api/config/db';
3 import { CacheService } from '@example-api/common';
4
5 @injectable()
6 export class RedisImpl implements CacheService {
7
8   async getKeyTTL(key: string): Promise<number> {
9     const client = getRedisClient();
10    return await client.ttl(key);
11  }
12
13   async getByKey<T>(key: string): Promise<T | null> {
14     const client = getRedisClient();
15     const response = await client.get(key);
16     if (!response) {
17       return null;
18     }
19     try {
20       return JSON.parse( text: response);
21     } catch (err) {
22       return response as unknown as T;
23     }
24   }
25
26   async setByKey<T>(key: string, value: T, seconds?: number): Promise<void> {
27     const expireTimeInSeconds = seconds ?? 20;
28     const client = getRedisClient();
29     await client.set(key, JSON.stringify(value), 'EX', expireTimeInSeconds);
30   }
31
32   memoize<T>(  
33     method: (...someArgs: unknown[]) => Promise<T>,  
34     ttl?: number  
35   ): (...someArgs: unknown[]) => Promise<T> {  
36     return async (...args) => {  
37       const recordKey = this.generateFunctionKey( functionName: method.name, args);  
38       const record = await this.getByKey<T>( key: recordKey);  
39       if (record && typeof record === 'string') {  
40         return f
```

Clean Architecture

Independencia de Frameworks y Herramientas:

RedisImpl utiliza Redis para la gestión de caché, pero esta dependencia está centralizada a través de un cliente específico (getRedisClient).

Independencia de la Interfaz de Usuario:

La clase no está acoplada a ninguna interfaz de usuario, lo que es coherente con la Clean Architecture.

Pilares de la POO

Encapsulamiento: Los detalles de la interacción con Redis están encapsulados dentro de RedisImpl, exponiendo solo los métodos definidos en la interfaz CacheService.

Abstracción: RedisImpl implementa CacheService, proporcionando una abstracción sobre los detalles específicos de Redis.

Polimorfismo: RedisImpl puede ser tratada como una implementación de CacheService, permitiendo el polimorfismo.

Principios SOLID

Single Responsibility Principle (SRP):

RedisImpl tiene una única responsabilidad: gestionar la caché utilizando Redis.

Open/Closed Principle (OCP): La clase está abierta a la extensión (por ejemplo, añadiendo nuevos métodos de caché) pero cerrada a la modificación en cuanto a su funcionalidad básica.

Liskov Substitution Principle (LSP): RedisImpl puede ser sustituida por cualquier otra implementación de CacheService sin alterar el comportamiento del sistema.

Interface Segregation Principle (ISP): RedisImpl implementa una interfaz específica CacheService, lo que garantiza que solo implementa los métodos necesarios.

Dependency Inversion Principle (DIP): RedisImpl depende de abstracciones (CacheService) y no de implementaciones concretas.

```

import { inject, injectable } from "inversify";
import { SYMBOLS } from "@example-api/config/symbols";
import { UseCase, UserRepository } from "@example-api/domain";
import { AuthService } from "../../common/auth/auth-service";

interface requestDto {
  requestId: string;
  userName: string;
  password: string;
}

interface responseDto {
  token: string;
}

export type SigninUseCase = UseCase<requestDto, responseDto>;

@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) {
      // ...
    }
  }
}

```

Clean Architecture

Separación de Responsabilidades:

La clase Signin se centra en la lógica de negocio relacionada con la autenticación de usuarios. No contiene lógica relacionada con la infraestructura o la presentación, lo que muestra una separación de responsabilidades.

Pilares de la POO

Encapsulamiento: Los detalles de implementación de la autenticación (como la interacción con el repositorio de usuarios y el servicio de autenticación) están encapsulados dentro de la clase Signin.

Abstracción: Signin utiliza abstracciones como UserRepository y AuthService, lo que permite desacoplar la implementación concreta de estas dependencias.

Principios SOLID

Single Responsibility Principle (SRP): La clase Signin tiene una única responsabilidad: manejar la lógica de autenticación de usuarios.

Open/Closed Principle (OCP): La clase Signin está abierta a la extensión (puede agregarse nueva funcionalidad relacionada con la autenticación) pero cerrada a la modificación de su comportamiento existente.

Liskov Substitution Principle (LSP): Signin puede ser sustituida por cualquier otra implementación de SigninUseCase sin alterar el comportamiento esperado del sistema.

Interface Segregation Principle (ISP): La interfaz SigninUseCase define un contrato específico para la autenticación de usuarios, lo que permite a las clases consumidoras interactuar con Signin de manera coherente sin depender de detalles internos.

Dependency Inversion Principle (DIP): Signin depende de abstracciones (UserRepository y AuthService) en lugar de implementaciones concretas, lo que facilita la flexibilidad y la modularidad del código.

```
TS sign-up.ts 3 X
src > domain > user > TS sign-up.ts > Signup > constructor
1  import { inject, injectable } from "inversify";
2  import { SYMBOLS } from "@example-api/config/symbols";
3  import { UseCase, UserRepository } from "@example-api/domain";
4  import { AuthService } from "../../common/auth/auth-service";
5
6  interface requestDto {
7    requestId: string;
8    userName: string;
9    password: string;
10 }
11
12 interface responseDto {
13   userName: string;
14 }
15
16 export type SignupUseCase = UseCase<requestDto, responseDto>;
17
18 @injectable()
19 export class Signup implements SignupUseCase {
20   private readonly userRepository: UserRepository;
21   private readonly authService: AuthService;
22
23   public constructor(
24     @inject(SYMBOLS.UserRepository)
25     userRepository: UserRepository,
26     @inject(SYMBOLS.AuthService)
27     authService: AuthService
28   ) {
29     this.userRepository = userRepository;
30     this.authService = authService;
31   }
32
33   public async execute(requestDto: requestDto): Promise<responseDto> {
34     try {
35       const { userName, password } = requestDto;
36       console.log( data[0]: { requestDto });
37       const hashPassword = this.authService.hashPassword( plainPassword: password);
38       await this.userRepository.create( data: {
39         userName,
40
41 domain > user > TS user-repository.ts > ...
import { User } from "../user";
export interface UserRepository {
  create(data: User): Promise<void>;
  getUserByUserName(userName: string): Promise<any>;
}
```

Clean Architecture

Independencia de Frameworks y Herramientas: La clase Signup no depende directamente de ningún framework o herramienta externa. Utiliza las abstracciones proporcionadas por las interfaces y la inyección de dependencias para interactuar con las capas internas del sistema.

Separación de Responsabilidades: La clase Signup se centra en la lógica de negocio relacionada con el registro de nuevos usuarios. No contiene lógica relacionada con la infraestructura o la presentación.

Pilares de la POO

Encapsulamiento: Los detalles de implementación del registro de usuarios (como la interacción con el repositorio de usuarios y el servicio de autenticación) están encapsulados dentro de la clase Signup.

Abstracción: Signup utiliza abstracciones como UserRepository y AuthService, lo que permite desacoplar la implementación concreta de estas dependencias.

Principios SOLID

Single Responsibility Principle (SRP): La clase Signup tiene una única responsabilidad: manejar la lógica de registro de nuevos usuarios.

Open/Closed Principle (OCP): La clase Signup está abierta a la extensión (puede agregarse nueva funcionalidad relacionada con el registro de usuarios) pero cerrada a la modificación de su comportamiento existente.

Liskov Substitution Principle (LSP): Signup puede ser sustituida por cualquier otra implementación de SignupUseCase sin alterar el comportamiento esperado del sistema.

Dependency Inversion Principle (DIP): Signup depende de abstracciones (UserRepository y AuthService) en lugar de implementaciones concretas, lo que facilita la flexibilidad del código.

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

Clean Architecture

Separación de Responsabilidades: La interfaz UseCase define un contrato para los casos de uso en la aplicación, estableciendo así una clara separación de responsabilidades entre la lógica de aplicación y los detalles de implementación.

Pilares de la POO

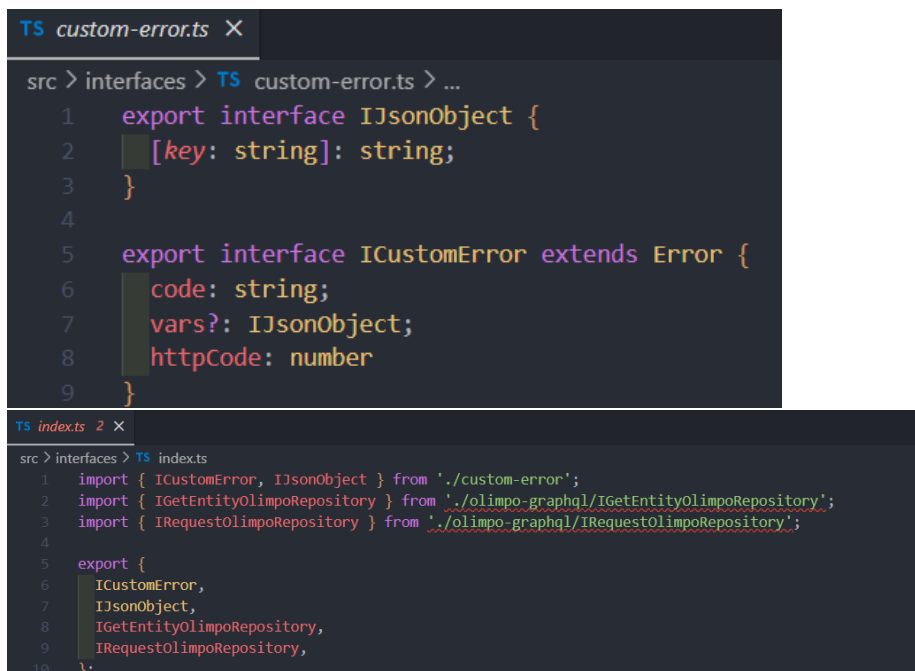
Abstracción: La interfaz UseCase proporciona una abstracción para representar cualquier caso de uso en la aplicación, independientemente de los detalles de implementación concretos.

Principios SOLID

Single Responsibility Principle (SRP):

UseCase tiene una sola responsabilidad: definir el contrato para los casos de uso en la aplicación.

Interface Segregation Principle (ISP): UseCase define un conjunto cohesivo de métodos (execute) que son relevantes para todos los casos de uso, sin obligar a las implementaciones a proporcionar funcionalidades innecesarias.



```
TS custom-error.ts X
src > interfaces > TS custom-error.ts > ...
1  export interface IJsonObject {
2    [key: string]: string;
3  }
4
5  export interface ICustomError extends Error {
6    code: string;
7    vars?: IJsonObject;
8    httpCode: number
9  }

TS index.ts 2 X
src > interfaces > TS index.ts
1  import { ICustomError, IJsonObject } from './custom-error';
2  import { IGetEntityOlimpoRepository } from '../olimpio-graphql/IGetEntityOlimpoRepository';
3  import { IRequestOlimpoRepository } from '../olimpio-graphql/IRequestOlimpoRepository';
4
5  export {
6    ICustomError,
7    IJsonObject,
8    IGetEntityOlimpoRepository,
9    IRequestOlimpoRepository,
10 };

```

Clean Architecture

Independencia de Frameworks y Herramientas: El código no depende de ningún framework específico y proporciona abstracciones genéricas que pueden ser utilizadas en cualquier contexto de aplicación.

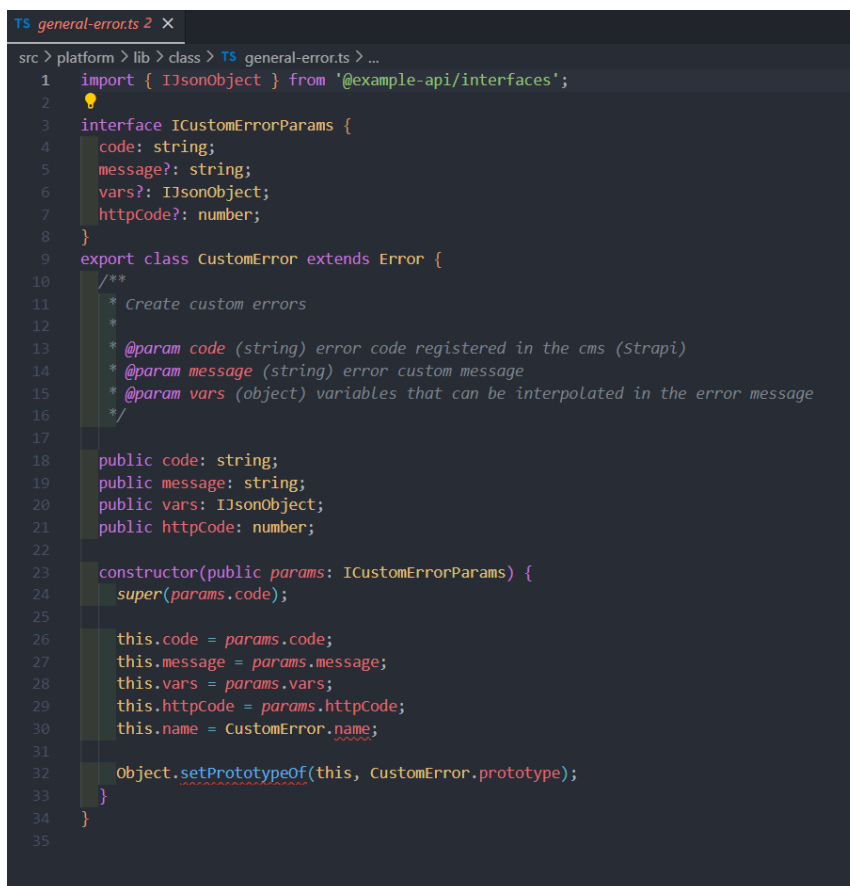
Pilares de la POO

Abstracción: Las interfaces `IJsonObject` e `ICustomError` proporcionan abstracciones para representar objetos JSON y errores personalizados, respectivamente. Esto permite desacoplar el código que utiliza estas estructuras de sus implementaciones concretas.

Principios SOLID

Interface Segregation Principle (ISP):

Las interfaces `IJsonObject` e `ICustomError` están bien segregadas, ya que representan un único concepto cohesivo y no obligan a las clases que las implementan a proporcionar funcionalidades irrelevantes.



```
TS general-errors.ts 2 X
src > platform > lib > class > TS general-errors > ...
1 import { IJsonObject } from '@example-api/interfaces';
2
3 interface ICustomErrorParams {
4   code: string;
5   message?: string;
6   vars?: IJsonObject;
7   httpCode?: number;
8 }
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   }
34 }
35
```

Pilares de la POO

Abstracción: La clase `CustomError` proporciona una abstracción para representar errores personalizados en la aplicación. Utiliza una interfaz `ICustomErrorParams` para definir los parámetros que pueden configurar el error.

Encapsulación: La clase CustomError encapsula la lógica relacionada con la creación de errores personalizados al manejar los parámetros de entrada y configurar las propiedades del error.

Principios SOLID

Single Responsibility Principle (SRP): La clase CustomError tiene una única responsabilidad: representar errores personalizados en la aplicación.

```
TS error-handlers.ts 2 X
src > platform > middlewares > TS error-handlers.ts > [⌘] errorHandler
1  import { ICustomError } from '@example-api/interfaces';
2  import { defaults } from '@example-api/constants';
3  import { constants } from 'http2';
4  import { ErrorRequestHandler, Response, NextFunction } from 'express';
5  import { CustomRequest } from '../server';
6  import { CustomError } from '../lib/class/general-error';
7
8  const { HTTP_STATUS_INTERNAL_SERVER_ERROR } = constants;
9
10 const buildResponse = (error: ICustomError, translatedMessage: string) => {
11   return {
12     code: error.code,
13     message: error.code ? translatedMessage : error.message,
14   };
15 };
16
17 export const errorHandler = (
18   error: CustomError,
19   req: CustomRequest,
20   res: Response
21 ) => {
22   try {
23     console.error( data[0]: error, data[1]: req.requestId);
24     const translatedMessage = error.message || defaults.ERROR_MESSAGE;
25     const response = buildResponse(error, translatedMessage);
26     return res
27       .status(error.httpCode || HTTP_STATUS_INTERNAL_SERVER_ERROR)
28       .json(response);
29   } catch (err) {
30     return res.status(HTTP_STATUS_INTERNAL_SERVER_ERROR).json(err);
31   }
32 };
33
34 export const errorHandlerMiddleWare: ErrorRequestHandler = (
35   error: CustomError,
36   req: CustomRequest,
37   res: Response,
38   next: NextFunction
39 ) => {
```

Pilares de la POO

Abstracción: La función errorHandler y el middleware errorHandlerMiddleWare encapsulan la lógica de manejo de errores en la aplicación, proporcionando una abstracción para este proceso.

Encapsulación: La función errorHandler y el middleware errorHandlerMiddleWare encapsulan la lógica de manejo de errores en funciones específicas, lo que facilita su reutilización y mantenimiento.

Principios SOLID

Single Responsibility Principle (SRP): Las funciones errorHandler y errorHandlerMiddleWare tienen una única responsabilidad: manejar errores de manera adecuada en la aplicación.

```
form > middlewares > TS validate-token.ts > ...
import { NextFunction, Response } from 'express';
import jwt from "jsonwebtoken";
import { CODE_ERRORS } from '@example-api/constants';
import config from '../config';
import { constants } from 'http2';
import { CustomError } from '../lib/class/general-error';
import { CustomRequest } from '../server';

const { HTTP_STATUS_UNAUTHORIZED } = constants;

export const validateToken = async (
  req: CustomRequest,
  res: Response,
  next: NextFunction
) => {
  try {
    if (!req.accessToken) {
      throw new CustomError( params: {
        code: CODE_ERRORS.UNAUTHORIZED,
        httpCode: HTTP_STATUS_UNAUTHORIZED,
      });
    }
    req.dataTokenUser = jwt.verify(req.accessToken, config.auth.secret);
    next();
  } catch (error) {
    next(error);
  }
};
```

Pilares de la POO

Abstracción: La función validateToken encapsula la lógica de validación del token JWT, proporcionando una abstracción para este proceso específico.

Encapsulación: La función validateToken encapsula la lógica de manejo de errores al lanzar un error personalizado si el token no está presente o no es válido.

Principios SOLID

Single Responsibility Principle (SRP): La función `validateToken` tiene una única responsabilidad: validar el token JWT y manejar errores asociados con esta validación.

```
TS express.ts 8 X
src > platform > server > TS express.ts > ...
1  import express, {
2    Application,
3    NextFunction,
4    RequestHandler,
5    Response,
6  } from 'express';
7  import helmet from 'helmet';
8  import cors from 'cors';
9  import compression from 'compression';
10 import { v4 as uuidv4 } from 'uuid';
11 import { connect, connectRedisClient } from "@example-api/config/db";
12 import { transformHeadersToCamelCase } from '@example-api/common';
13 import config from '../config';
14 import { CustomRequest, IStartOptions } from './types';
15 import { errorHandlerMiddleware } from '../middlewares/error-handler';
16
17 /**
18  * Bootstraps and start express application
19  * @author Enrique Aguilar
20  * @param handlers (RequestHandler) express Request handler functions
21  * @param options (StartOptions) object to configure server behavior
22  * @returns express application
23  */
24 export const startExpressServer = (
25   handlers: RequestHandler | RequestHandler[],
26   options: IStartOptions
27 ) => {
28   const { basePath, port, host, requestId } = options;
29   const app: Application = express();
30   app.use((req: CustomRequest, res: Response, next: NextFunction) => {
31     const { requestId, accessToken } = transformHeadersToCamelCase(req.headers);
32     req.requestId = (requestId as string) || uuidv4();
33     req.accessToken = accessToken as string;
34     console.info(
35       data[0]: `[START] - Path: ${req.originalUrl}`,
36       data[1]: req.requestId
37     );
38     next();
39   });
40 }
```

Pilares de la POO

Abstracción: La función `startExpressServer` abstrae la lógica de configuración del servidor Express y el inicio del servidor, permitiendo que esta lógica sea reutilizable y extensible.

Encapsulación: La función `startExpressServer` encapsula la lógica de configuración del servidor y el manejo de solicitudes HTTP dentro de una función específica, lo que nos facilita su reutilización y mantenimiento.

Principios SOLID

Single Responsibility Principle (SRP): La función `startExpressServer` tiene una sola responsabilidad: configurar y arrancar el servidor Express.

```

TS user.ts 1 X
src > routes > v1 > TS user.ts > ...
1  import { Router } from "express";
2  import { container } from "@example-api/config/inversify";
3  import {
4      UserSignupController,
5      UserSigninController,
6  } from "@example-api/controllers";
7
8  const signup = container.get(UserSignupController);
9  const signin = container.get(UserSigninController);
10
11 const userRouter = Router();
12
13 userRouter.post("/signin", (req, res) => signin.execute(req, res));
14
15 userRouter.post("/signup", (req, res) => signup.execute(req, res));
16
17
18 export { userRouter };
19

```

Pilares de la POO

Abstracción: Se utiliza la abstracción de controladores para encapsular la lógica de manejo de solicitudes HTTP en los controladores `UserSignupController` y `UserSigninController`.

Encapsulación:

La lógica de manejo de solicitudes HTTP se encapsula dentro de las funciones `execute` de los controladores, lo que nos permite una fácil reutilización y mantenimiento.

Principios SOLID

Principio de Responsabilidad Única (SRP):

Cada controlador (`UserSignupController` y `UserSigninController`) tiene una única responsabilidad: manejar las solicitudes de inicio de sesión y registro de usuarios, respectivamente.


```

TS index.ts 1 X
src > TS index.ts > ...
1  import 'reflect-metadata';
2  import { v4 as uuidv4 } from "uuid";
3  import { v1Routes } from '@example-api/routes/v1';
4  import { config, startExpressServer } from '@example-api/platform/index';
5
6  const executorId = uuidv4();
7
8  export const app = startExpressServer( handlers: [v1Routes], options: {
9      requestId: executorId,
10     port: config.port,
11     host: config.host,
12     basePath: '',
13 });
14

```

Pilares de la POO

Abstracción: Se utiliza una abstracción para encapsular la lógica de enrutamiento de la API. La configuración del servidor y el inicio del mismo se manejan a través de una función `startExpressServer`, que abstrae la complejidad de la configuración del servidor y la gestión de rutas.

Principios SOLID

Principio de Responsabilidad Única (SRP): La función `startExpressServer` tiene una única responsabilidad: configurar y arrancar el servidor Express. Esta función es reutilizable y fácil de mantener debido a su enfoque en una única tarea.