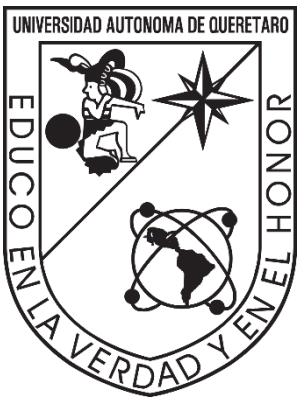
UNIVERSIDAD AUTÓNOMA DE QUERÉTARO



Ingeniería de Software

Patrones de Diseño

Proyecto Final

Alumno:

Emiliano Aquetzalli Obregón Reyes 307099

Sebastián Chavarría Villanueva 276804

Maestro:

Ing. Enrique Aguilar

Fecha:

18/05/2024

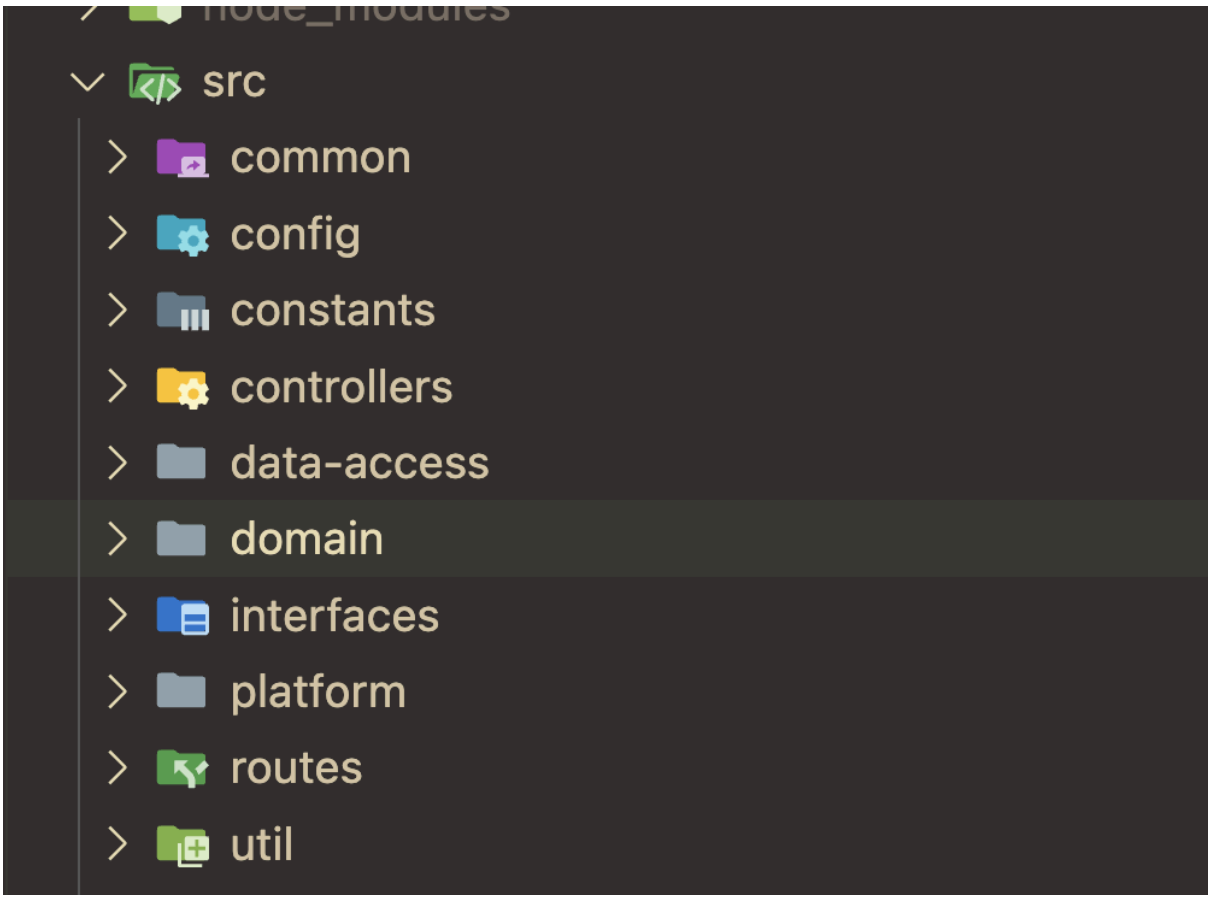
Implementación de Clean Architecture

La Clean Architecture busca crear sistemas robustos, escalables y mantenibles mediante la separación de preocupaciones y la modularidad en el diseño de software. Robert C. Martin, también conocido como "Uncle Bob", propuso que el diseño de un sistema debe ser independiente de su implementación técnica.

La Clean Architecture tiene capas concéntricas, cada una con responsabilidades específicas y dependiendo solo de las capas internas. Estas capas suelen representarse en círculos concéntricos, donde la capa más interna representa las políticas de negocio y las externas representan los detalles de implementación técnica.

Empecemos analizando la capa de Entities (entidades) y en qué consisten : Las entidades del dominio, que representan los conceptos fundamentales y las reglas de negocio del sistema, están contenidas en esta capa. Las entidades suelen ser objetos sencillos y no dependen de ningún detalle técnico de implementación.

La implementación en el proyecto dado esta de la siguiente manera



Domain

Dentro de la carpeta principal del proyecto “src” se tiene otro directorio que contiene las entidades del dominio, es decir, las clases que representan los objetos fundamentales y las reglas de negocio de la aplicación como se muestra de la siguiente manera

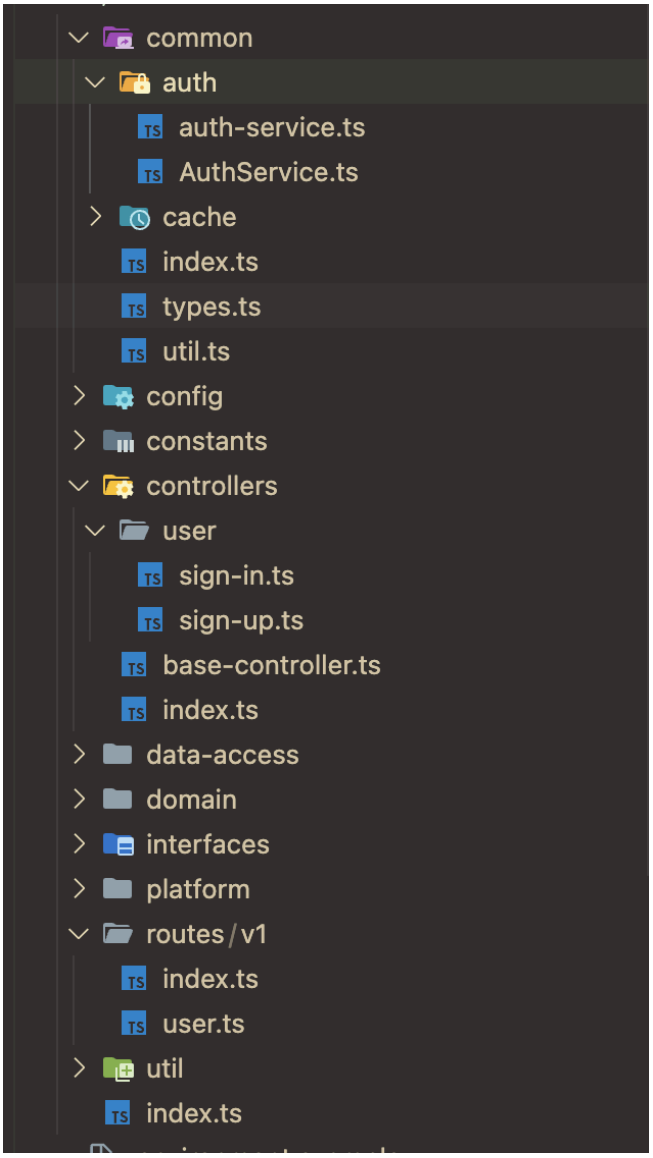
Las entidades del dominio como es el directorio de user y dentro de este el repositorio donde están alojados. Los objetos fundamentales y las reglas de negocio de la aplicación son representados por las entidades. Garantiza que estén desacopladas de cualquier detalle técnico al separarlas aquí.

UseCases

Después pasamos a la capa de “UseCase” Aquí se implementarán los casos de uso de la aplicación. Cada caso de uso será una clase que contendrá la lógica de la aplicación y coordinará las operaciones en las entidades del dominio. dentro de la carpeta de cada entidad en este caso user se tienen las acciones que puede realizar como lo son sing-in y sing-up a un repositorio

Interfaces

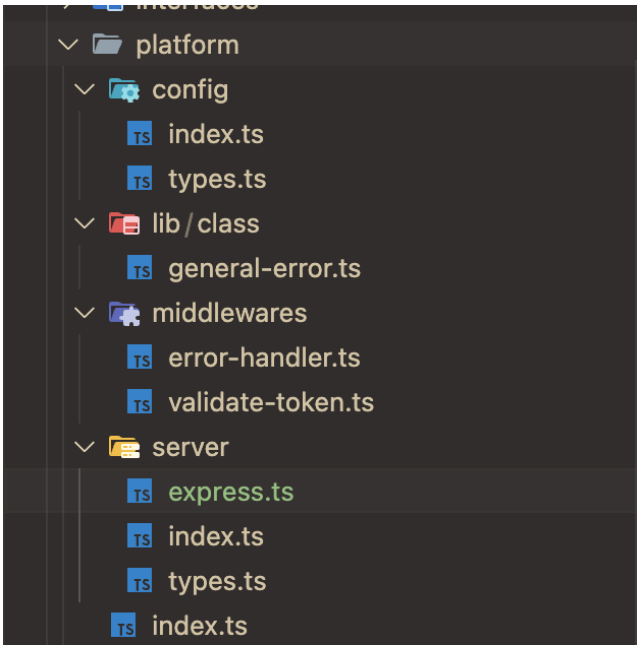
Antes de identificar los elementos presentes de esta capa en el proyecto hay que recordar que se encuentra aquí.
En la capa de Interfaces se encuentran todos los elementos que van a estar relacionados con la interacción aplicación y el mundo exterior, y una vez definido podemos señalar los elementos los cuales vienen siendo 3, las rutas, los middlewares y los controllers ya que estos se comunican con el cliente base a peticiones HTTP

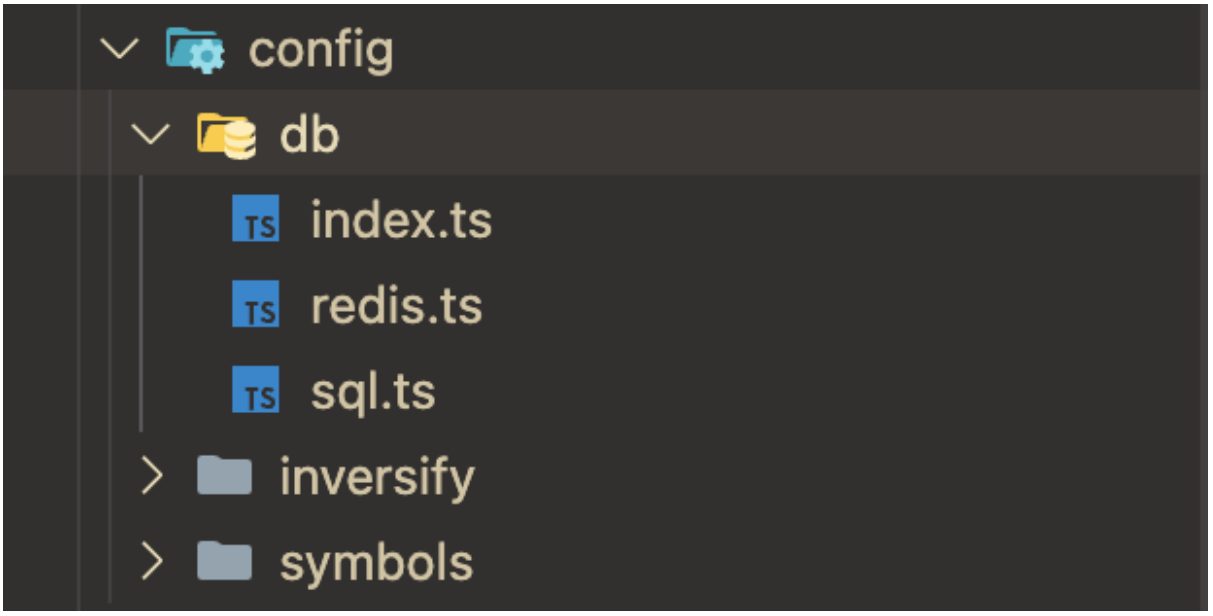


Se pueden identificar 3 directorios que cumplen con lo indicado los cuales son routes, controllers y la carpeta common ya que aquí se encuentran los middlewares de auth

Framework y config

En esta capa debemos de identificar los elementos de configuración tanto de las bases de datos como de las tecnologías que se estén utilizando y para esto tenemos dos directorios principales que cumplen con este rol los cuales son los de platforms que llevan la configuración de express y el directorio de config que configura las dos bases de datos utilizadas SQL y Redis y otras tecnologías como GRAPHQL


platform
config
index.ts
types.ts
lib / class
general-error.ts
middlewares
error-handler.ts
validate-token.ts
server
express.ts
index.ts
types.ts
index.ts


config
db
index.ts
redis.ts
sql.ts
inversify
symbols

Patrones de diseño implementados

Analizando todos los códigos implementados en este proyecto uno por uno también identificamos algunos patrones de diseño vistos en clase a continuación mostraremos algunos ejemplos

“src/config/db/redis.ts”

```
import Redis from "ioredis";
import config from "../../platform/config/index";

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(
        "connect-redis-client",
        requestId || "not requestId was provided",
      );
    });
    redisClient.on("error", async (error) => {
      console.error(
        error,
        "connect-redis-client",
        requestId || "not requestId was provided",
        "cache-sdk"
      );
    });
}
```

El código implementa el patrón de diseño Singleton para garantizar que solo haya una instancia de cliente de Redis en toda la aplicación, lo que puede ser beneficioso para administrar la conexión y evitar múltiples conexiones innecesarias.

“src/config/db/sql.ts”

```
const { Client } = require("pg");
import config from '../../platform/config/index';

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("connection error", err.stack);
        } else {
            console.log("connected");
        }
    });
}

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

El código también implementa el patrón de diseño Singleton para garantizar que solo haya una instancia del cliente de PostgreSQL en toda la aplicación, lo que puede ser beneficioso para administrar la conexión y evitar múltiples conexiones innecesarias.

“src/config/inversify/indexl.ts”

```ts
import { Container } from 'inversify';
import { SYMBOLS } from '@example-api/config/symbols';
import {
  AuthService,
  CacheService,
  IAuthService,
} from '@example-api/common';
import { BaseController } from '../../controllers/base-controller';
import {
  UserSignupController,
  UserSigninController,
} from "@example-api/controllers";
import {
  RedisImpl,
  UserRepositoryImpl,
} from '@example-api/data-access';
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 };
```

Utilizando la biblioteca inversify, este código implementa claramente el patrón de Inyección de Dependencias. Varias dependencias (servicios, controladores, casos de uso, etc.) Se definen en un contenedor (Container) y luego se vinculan a sus implementaciones concretas. Posteriormente, estas clases pueden ser inyectadas en otras dependencias que las requieran, lo cual facilitará la modularidad y simplificará las pruebas unitarias.

“src/config/symbols/indexl.ts”

```
export const SYMBOLS = {
  CacheService: Symbol("CacheService"),
  AuthService: Symbol("AuthService"),
  ScoreIntentUseCase: Symbol("ScoreIntentUseCase"),
  ScoreTopUseCase: Symbol("ScoreTopUseCase"),
  InitGameUseCase: Symbol("InitGameUseCase"),
  ScoreStatsUseCase: Symbol("ScoreStatsUseCase"),
  SignupUseCase: Symbol("SignupUseCase"),
  SigninUseCase: Symbol("SigninUseCase"),
  CatalogueRepository: Symbol("CatalogueRepository"),
  UserRepository: Symbol("UserRepository"),
  ScoreRepository: Symbol("ScoreRepository"),
};
```

"Identificador de Símbolos" es el nombre del patrón utilizado en este código. En esencia, se asignan nombres especiales a diversas partes importantes de la aplicación, como servicios o funciones. Estos nombres son únicos y se utilizan para identificar y acceder a estas partes en el código. En lugar de usar directamente el nombre "CacheService" para referirse a un servicio, se prefiere emplear un nombre único asociado al mismo, como por ejemplo "SYMBOLS". CacheService".
Esta práctica mejora la claridad del código y reduce su propensión a errores, debido a que los nombres de los símbolos son descriptivos y fáciles de comprender. Además, ayuda a mantener la aplicación organizada y modular al facilitar la gestión de dependencias. En síntesis, el empleo de identificadores de símbolos contribuye a mejorar la legibilidad y mantenibilidad del código.

“src/controllers/user/sign_in.ts”

```
import { inject, injectable } from "inversify";
import { SYMBOLS } from "@example-api/config/symbols";
import { Response } from "express";
import { constants } from "http2";

import { SigninUseCase } from "@example-api/domain";
import { CustomRequest } from "@example-api/platform/server/types";
import { CustomError } from "@example-api/platform/lib/class/general-error";
import { BaseController } from "../base-controller";

const {
  HTTP_STATUS_CREATED,
  HTTP_STATUS_BAD_REQUEST,
  HTTP_STATUS_INTERNAL_SERVER_ERROR,
} = constants;

@injectable()
export class UserSigninController extends BaseController {
  private signin: SigninUseCase;

  public constructor(
    @inject(SYMBOLS.SigninUseCase)
    signin: SigninUseCase
  ) {
    super();
    this.signin = signin;
  }

  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.signin.execute(inputDto);

      response.header('access-token', dataDto.token);

      return this.ok(request, response, HTTP_STATUS_CREATED, dataDto);
    } catch (error) {
      return this.fail(
```

```
        request,
        response,
        HTTP_STATUS_INTERNAL_SERVER_ERROR,
        error
      );
    }
  }
}
```

También emplea "Inyección de Dependencias" este código. En esencia, esto implica que la clase UserSigninController puede aceptar objetos externos (llamados dependencias) a través de su constructor. Se realiza esto para evitar que la clase tenga que crear estas dependencias por su cuenta, lo cual la hace más flexible y fácil de probar.

Para el correcto funcionamiento de la clase UserSigninController en este escenario, requiere un objeto SigninUseCase. En vez de crear este objeto dentro de la clase, se le pasa como un argumento al constructor utilizando la función inject y el decorador @injectable.

UserSigninController puede utilizar diferentes implementaciones de SigninUseCase sin necesidad de modificar su código gracias a este enfoque. Además, facilita probar UserSigninController con versiones simuladas o de prueba de SigninUseCase.
Aplica lo mismo para Sing-up.ts

“src/data-access/cache/redis-impl.ts”

```
import { injectable } from 'inversify';
import { getRedisClient } from '@example-api/config/db';
import { CacheService } from '@example-api/common';

@injectable()
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(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) => {
      const recordKey = this.generateFunctionKey(method.name, args);
      const record = await this.getByKey<T>(recordKey);
      if (record && typeof record === 'string') {
        try {
          return JSON.parse(record);
        } catch (err) {
          return record;
        }
      } else if (record) {
        return record;
      }
```

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

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

      return response;
    };
  }

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

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

    if (typeof args === 'object' && Object.keys(args).length) {
      return `${functionName}-${JSON.stringify(args)}`;
    }

    return functionName;
  }

  async deleteByKey(key: string): Promise<void> {
    const client = getRedisClient();
    await client.del(key);
  }
}
```

Aunque no está explícitamente implementado en el código, la función getRedisClient parece devolver una única instancia del cliente Redis. Esto sugiere un enfoque de diseño singleton para la gestión de la conexión a Redis, donde se garantiza que solo haya una instancia del cliente Redis en toda la aplicación.

“/src/platform/middlewares/error-handler.ts”

```
import { ICustomError } from '@example-api/interfaces';
import { defaults } from '@example-api/constants';
import { constants } from 'http2';
import { ErrorRequestHandler, Response, NextFunction } from 'express';
import { CustomRequest } from '../server';
import { CustomError } from '../lib/class/general-error';

const { HTTP_STATUS_INTERNAL_SERVER_ERROR } = constants;

const buildResponse = (error: ICustomError, translatedMessage: string)
=> {
  return {
    code: error.code,
    message: error.code ? translatedMessage : error.message,
  };
};

export const errorHandler = (
  error: CustomError,
  req: CustomRequest,
  res: Response
) => {
  try {
    console.error(error, req.requestId);
    const translatedMessage = error.message || defaults.ERROR_MESSAGE;
    const response = buildResponse(error, translatedMessage);
    return res
      .status(error.httpCode || HTTP_STATUS_INTERNAL_SERVER_ERROR)
      .json(response);
  } catch (err) {
    return res.status(HTTP_STATUS_INTERNAL_SERVER_ERROR).json(err);
  }
};

export const errorHandlerMiddleWare: ErrorRequestHandler = (
  error: CustomError,
  req: CustomRequest,
  res: Response,
  next: NextFunction
) => {
  errorHandler(error, req, res);
};
```

El código ejemplifica el patrón de diseño "Cadena de Responsabilidad". En este patrón, una secuencia de objetos (middlewares) procesan una solicitud uno por uno hasta que alguno la maneja correctamente. En el contexto de Express, cada middleware tiene la oportunidad de manejar el error capturado, pasando la responsabilidad al siguiente middleware de la cadena si no puede manejarlo. El middleware errorHandlerMiddleWare intenta manejar el error primero, delegando al manejador de errores predeterminado de Express si falla. Así, la responsabilidad de manejar el error se transfiere de un middleware a otro hasta que se maneja apropiadamente, demostrando el patrón "Cadena de Responsabilidad".

Pilares de la POO

La Programación Orientada a Objetos es uno de los paradigmas de programación más utilizados debido a sus grandes ventajas, como lo son la Modularidad, Reutilización de código, Mantenimiento y Escalabilidad, entre otras.

En el proyecto podemos encontrar cada uno de los pilares vistos en clase, como lo son: Herencia, Abstracción, Encapsulación y Polimorfismo, a continuación veamos algunos ejemplos de ello.

Abstracción

La abstracción consiste en ocultar los detalles complejos de la implementación y mostrar solo la funcionalidad esencial del objeto. Esto se logra mediante el uso de interfaces y clases abstractas.

Se puede observar la abstracción en la definición de las interfaces y clases en varios puntos del código. Por ejemplo, en las interfaces CacheService, UserRepository, UseCase, y en la clase abstracta BaseController.

Interfaces:

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

```ts
export interface CacheService {
  getByKey<T>(key: string): Promise<T | null>;
  setByKey<T>(
    key: string,
    value: T,
    expireTimeInSeconds?: number
  ): Promise<void>;
  generateFunctionKey<T>(functionName: string, args?: T): string;
  memoize<T>(
    method: (...args: unknown[]) => Promise<T>,
    ttl?: number
  ): (...args: unknown[]) => Promise<T>;
  deleteByKey(key: string): Promise<void>;
  getKeyTLL(key: string): Promise<number>;
}
```

Ejemplo de Implementación:

```ts
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);
```

Encapsulación

El encapsulamiento consiste en ocultar los detalles internos de un objeto y exponer **sólo lo necesario** a través de métodos públicos.

```
@injectable()
export class UserSigninController extends BaseController {
  private signin: SigninUseCase;

  public constructor(
    @inject(SYMBOLS.SigninUseCase)
    signin: SigninUseCase
  ) {
    super();
    this.signin = signin;
  }
```

El modificador de acceso private restringe el acceso a la variable signin de manera que sólo es accesible dentro de la clase UserSigninController. Esto es encapsulamiento porque protege la variable signin de ser accedida o modificada directamente desde fuera de la clase, esto ocurre también en sign-up.

```
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(
      `[END] - Path: ${req.originalUrl}`,
      BaseController.name,
      req.requestId
    );
    if (dto) {
      res.type('application/json');
      return res.status(httpCode).json(dto);
    }

    return res.sendStatus(httpCode);
  }

  protected fail(
```

En este otro ejemplo tenemos la clase de Base Controller, los métodos ok y fail están protegidos, lo que impide el acceso directo desde fuera de la clase y asegura que solo las clases que heredan de BaseController puedan utilizarlos.

Herencia

La herencia permite que una clase (subclase) herede atributos y métodos de otra clase (superclase), promoviendo la reutilización del código.

La herencia se utiliza para compartir comportamientos comunes entre diferentes clases. Por ejemplo, la clase BaseController actúa como una clase base para los controladores de registro y inicio de sesión (UserSignupController, UserSigninController),

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

```
@injectable()
export class UserSigninController extends BaseController {
  private signin: SigninUseCase;
```

Además, en TypeScript, las interfaces se pueden heredar de otras interfaces. Esto significa que una interfaz puede tomar propiedades y métodos de otra interfaz para reutilizar su estructura. Un ejemplo de esto se ve en la interfaz UseCase que hereda de CustomRequestId y ResponseType.

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

Polimorfismo

El Polimorfismo consiste en que puedes tener diferentes clases que comparten una misma interfaz o “superclase”. Aunque estas clases pueden tener implementaciones diferentes para sus métodos, pueden ser tratadas de la misma manera cuando se invocan a través de la interfaz común.

A diferencia de la abstracción que se refiere a representar características esenciales de un objeto, el polimorfismo se refiere a tratar objetos de la misma forma mediante una interfaz común.

Por ejemplo, la clase BaseController donde podemos encontrar métodos comunes como ok y fail, que son utilizados por los controladores para manejar las respuestas HTTP. Aunque estos métodos están definidos en la clase base, pueden adaptarse a las necesidades específicas de cada controlador.

```
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(
      `[END] - Path: ${req.originalUrl}`,
      BaseController.name,
      req.requestId
    );
    if (dto) {
      res.type('application/json');
      return res.status(httpCode).json(dto);
    }

    return res.sendStatus(httpCode);
  }

  protected fail(
```

Principios SOLID

Single Responsibility:

Este principio establece que una clase debe tener una **única razón para cambiar**. Podemos ver este principio aplicado en varias partes. En las clases como AuthService, RedisImpl, UserRepositoryImpl, Signup, Signin, y BaseController, cada una de ellas tiene una responsabilidad claramente definida.

AuthService se encarga de la autenticación de usuarios

```
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;
  }
```

UserRepositoryImpl maneja las operaciones de base de datos relacionadas con los usuarios

```
export class UserRepositoryImpl implements UserRepository {

  async create(data: User): Promise<void> {
    const { userName, password } = data;
    await sql({
      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: `
        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;
```

Signup con el registro y Signin con el inicio de sesión.

Principio de Abierto/Cerrado:

Este principio establece que las clases deben estar abiertas para la extensión pero cerradas para la modificación. El código debe poder adaptarse a nuevos requisitos o cambios de comportamiento sin necesidad de modificar el código existente.

La arquitectura permite la extensión de funcionalidades sin necesidad de modificar el código existente. Por ejemplo, si se desea agregar una nueva funcionalidad, como la autenticación de dos factores, se puede hacer fácilmente extendiendo el código existente sin alterar su estructura interna.

Las interfaces y la inyección de dependencias permiten que las clases dependan de abstracciones en lugar de implementaciones concretas.

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

interface responseDto {
  userName: string;
}
```

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

Principio de Sustitución de Liskov:

El principio de Liskov se centra en la relación entre una clase base y sus subclases, estableciendo que las subclases deben poder sustituir a la clase base sin afectar el comportamiento esperado del programa.

En el proyecto, cada caso de uso que implementa la interfaz UseCase puede ser considerado una subclase que cumple con el principio de Liskov. Ya que estas subclases pueden ser utilizadas en lugar de su clase base “UseCase” en cualquier parte del código que espere un objeto que cumpla con el contrato de UseCase, sin cambiar el comportamiento esperado del programa.

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

Principio de Inversión de Dependencias:

Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. Además, las abstracciones no deben depender de los detalles. Los detalles deben depender de abstracciones.

La **inyección de dependencias** es un patrón de diseño que implementa el principio de inversión de dependencias. Dos ejemplos donde podemos ver la inversión de dependencias en el proyecto son:

```
1   import { injectable } from "inversify";
2   import { sql } from "@example-api/config/db";
3   import { UserRepository } from "@example-api/domain";
4   import { User } from "src/domain/user/user";
5
6   @injectable()
7   export class UserRepositoryImpl implements UserRepository {
8
9     async create(data: User): Promise<void> {
10      const { userName, password } = data;
11      await sql({
12        query: `
13          INSERT INTO
14            user_account(usr_act_name, usr_act_password)
15          VALUES($1,$2)
16        `,
17        bind: [userName, password],
18      });
19    }
```

El código ejemplifica el principio de Inversión de Dependencias al hacer que UserRepositoryImpl implemente la interfaz UserRepository y utilizar la anotación @injectable().

El uso de @injectable() de Inversify en el código marca UserRepositoryImpl como una clase que puede ser inyectada en otros componentes. Esto habilita la inyección de dependencias.

```
@injectable()
export abstract class BaseController {
  public abstract execute(
```

```
@injectable()
export class RedisImpl implements CacheService {
```

CONCLUSIÓN

El proyecto implementa la Arquitectura Limpia de manera efectiva, lo que promueve la modularidad, escalabilidad y mantenibilidad del sistema. Además, se han aplicado varios patrones de diseño vistos a lo largo del taller, como Singleton, Inyección de Dependencias, y Cadena de Responsabilidad. Se han respetado los principios SOLID, lo que garantiza un código limpio, extensible y fácil de mantener.