



UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
FACULTAD DE INFORMÁTICA

Taller: Patrones de diseño

Proyecto Final

Fecha de Entrega: 17/05/2024

Integrante:

Alan Olvera Cristino - 307050

Contenido :

Tipo de arquitectura usada en el proyecto

Por la estructura de las carpetas y componentes del proyecto puedo mencionar que se está usando una arquitectura jerárquica, comúnmente conocida por capas, principalmente por el como esta dividido el funcionamiento, teniendo nuestras 4 principales capas:

Capa de Presentación:

- **Responsabilidad:** Interacción con el usuario. Gestiona la interfaz de usuario y la lógica relacionada con la experiencia del usuario.

Capa de Aplicación o Lógica de Negocio:

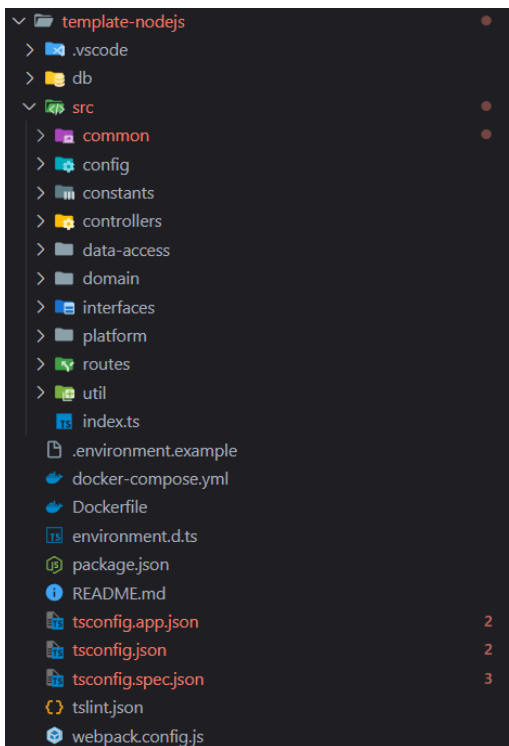
- **Responsabilidad:** Contiene la lógica del negocio y procesa las reglas de negocio. Actúa como intermediario entre la capa de presentación y la capa de datos.

Capa de Acceso a Datos:

- **Responsabilidad:** Manejamos las operaciones de lectura y escritura en la base de datos. Se implementan los patrones de acceso a datos.

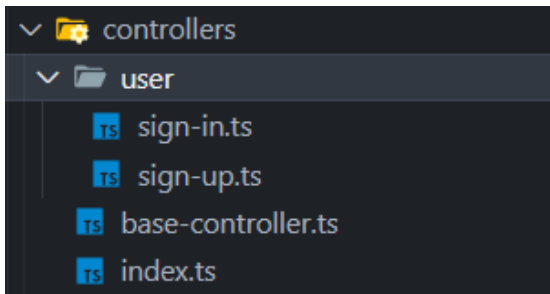
Capa de Persistencia (Base de Datos):

- **Responsabilidad:** Almacenamos y recuperamos datos desde una base de datos, en este caso se utiliza sql y redis

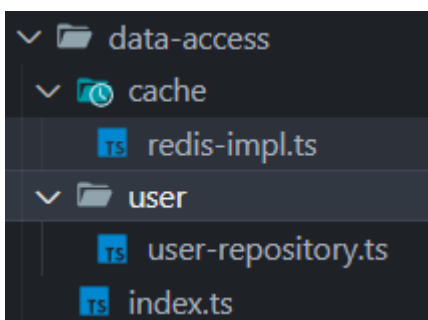


Podemos identificar las capas dentro de estos espacios principalmente

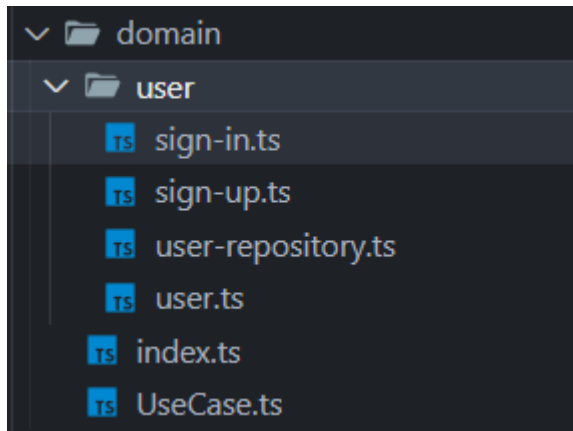
Controllers (Controladores): Esta capa se encarga de manejar las solicitudes del usuario y las respuestas del sistema.



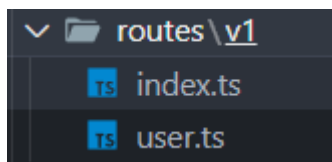
Data-Access (Acceso a Datos): Se ocupa de la comunicación con las bases de datos o cualquier otro tipo de almacenamiento.



Domain (Dominio): Contiene la lógica principal del negocio y las reglas asociadas.



Routes (Rutas): Administra las rutas y endpoints para el manejo adecuado de las solicitudes y respuestas.

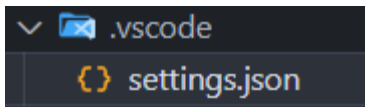


Descripción

A continuación navegaremos sobre todos los archivos que comprenden el proyecto, de esta manera podremos identificar qué patrones de diseño se implementaron y que elementos de POO se usaron

Carpeta .vscode

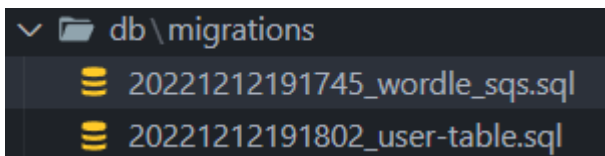
Este es uno de los archivos que podemos comprender como archivos de desarrollo o usados durante el desarrollo, el cual no comprende parte del proyecto, sino de su desarrollo, en este caso, especificando palabras especiales usadas en el proyecto que queremos que se agreguen al diccionario y tomen parte para verificar errores ortográficos



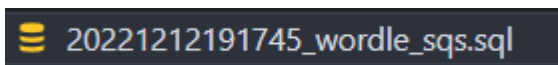
```
{
  "cSpell.words": [
    "inversify",
  ]
}
```

Carpeta db/migrations

En el caso de esta carpeta nos encontramos de cara a instrucciones de bd



En el caso del primer archivo lo que encontramos es una manera de crear una secuencia específica utilizada en el proyecto para poder asignar llaves primarias



```
-- migrate:up
CREATE SEQUENCE seq_wordle_user_wus_id
  INCREMENT BY 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1
  NO CYCLE;

-- migrate:down
DROP SEQUENCE seq_wordle_user_wus_id;
```

En cuanto al segundo archivo que encontramos tenemos que son instrucciones sql para crear y eliminar tablas de una base de datos, esto usado en caso de migraciones debido a las marcas de inicio en cada sección

20221212191802_user-table.sql

```
-- migrate:up
CREATE TABLE user_account (
  usr_act_id int8 NOT NULL DEFAULT nextval('seq_wordle_user_wus_id'::regclass),
  usr_act_name varchar(20) NOT NULL,
  usr_act_password varchar(150) NOT NULL,
  usr_act_creation_date timestamp NOT NULL DEFAULT now(),
  usr_act_update_date timestamp NOT NULL DEFAULT now(),
  usr_act_delete_date timestamp NULL,
  CONSTRAINT pk_usr_act_id PRIMARY KEY (usr_act_id),
  CONSTRAINT user_unique UNIQUE (usr_act_name)
);
COMMENT ON COLUMN "user_account"."usr_act_id" IS 'This column represent the id of the entity. It composes the primary key of';
CREATE INDEX usr_act_name_idx ON user_account USING btree (usr_act_name);
CREATE INDEX usr_act_delete_date_idx ON user_account USING btree (usr_act_delete_date);

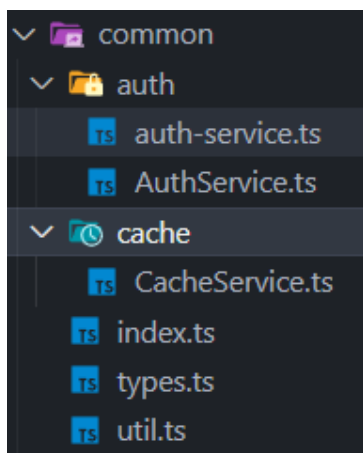
-- migrate:down
ALTER TABLE "user_account" DROP CONSTRAINT "pk_usr_act_id";
DROP INDEX "usr_act_name_idx";
DROP INDEX "usr_act_delete_date_idx";
DROP TABLE "user_account";
```

En el caso de esta carpeta podremos intuir que forma parte de cierta manera a la capa de persistencia

Seguimos con los archivos y carpetas dentro de la carpeta src

Carpeta common

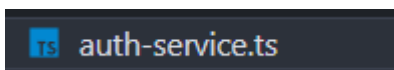
Dentro de esta carpeta nos encontramos con archivos que funcionan en la capa de la lógica de negocio



Carpeta common/auth

En el caso de este archivo se define un servicio de autenticación, la cual implementa una interfaz, el funcionamiento principal es generar y verificar JWT, tanto como hash para la comparación de contraseñas

Al observar el código nos damos cuenta de que usa un patrón de diseño muy común que es la inyección de dependencias, esto debido a la instrucción de `@injectable`, esto ya que hace que las instancias sean creadas por inversify, en cuanto a los pilares de poo se puede notar una encapsulación, esto debido a las funciones de verificación y hash están ocultas dentro del método `authservice`



```
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;
  }
}
```

En este archivo lo que se realiza es básicamente declarar la interfaz que se

utiliza en el código anterior, como podemos ver se realiza la definición de los métodos que serán implementados, en cuanto a los principios solid, este demuestra el principio de inversión de dependencias, ya que es una abstracción sobre la cual las clases de alto nivel pueden depender, en lugar de ser de una implementación directa

AuthService.ts

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

Carpeta cache


Este caso es similar al anterior, ya que estamos declarando una interfaz por lo que contiene el patrón de inversión de dependencias por la misma razón de la anterior, pero igual se usa el principio de segregación, esto al dotar de una definición específica al servicio de caché, también se utiliza lo que parece ser un tipo de patrón parecido a memento, ya que este guarda o almacena estados dentro de la interfaz

cache CacheService.ts

```
export interface CacheService {  
  getByKey<T>(key: string): Promise<T | null>;  
  setByKey<T>(  
    key: string,  
    value: T,  
    expiresInSeconds?: 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>;  
}
```



Archivos dentro de common

Lo que hace este archivo no es más que lógica dentro del desarrollo, ya que permite importar distintas dependencias desde un solo lugar, esto en lugar de importarlas directamente

 index.ts

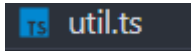
```
export { CacheService } from './cache/'  
export { AuthService } from './auth/au  
export * from './util';  
export * from './types';  
export * from './auth/AuthService';
```

En este archivo seguimos con archivos de configuración, ya que lo que único que hace es implementar una función en la cual se especifica un tipo de dato, esto al ser una interfaz usada en otros sitios cae en el principio de segregación de los principios SOLID

 types.ts

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

Este al ser un archivo util tiene como funcionamiento el proveer de distintas funciones al programa, en este caso se cuenta con varias funciones que principalmente cumplen con el principio de responsabilidad unica ya que cada función tiene una sola tarea



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

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

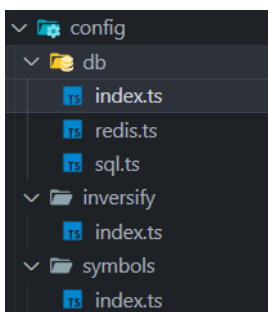
export const transformHeadersToCamelCase = (headers: {
  [k: string]: string | string[] | undefined;
}): { [k: string]: string | string[] } =>
  Object.entries(headers).reduce((r, [k, v]) => {
    return {
      ...r,
      [toCamelCase(k)]: v,
    };
  }, {});

export const hashKeyFn = <T>(data: T): string => {
  const objectToHash = JSON.stringify(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;
}
```

Carpeta config

Esta carpeta tiene varias reglas de negocio que pueden ser usadas como configuración a lo largo del proyecto, como lo son las configuraciones de las bd



Carpeta db

Nos encontramos con otro archivo de dependencias, el cual se usa de la misma manera que el anterior, más que nada sirve para importar bloques de dependencias desde un solo archivo

TS index.ts

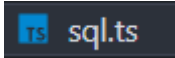
```
export * from './redis';  
export * from './sql';
```

En este caso se trata del manejo de conexión a una bd redis, debido a la naturaleza del archivo se aplica el patrón singleton para asegurar que solo haya una instancia de la conexión, de la misma se usa el principio de responsabilidad única, ya que cada función realiza únicamente una cosa

TS redis.ts

```
import Redis from "ioredis";    Cannot find module '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> {    An async function or method in ES5/ES3  
  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) => {    An async fu  
    console.error(  
      error,  
      "connect-redis-client",  
      requestId || "not requestId was provided",  
      "cache-sdk"  
    );  
  });  
}
```

En este caso al igual que el anterior tenemos una conexión a bd pero esta vez de sql, por lo que tenemos los mismos patrones y principios, tanto el singleton como de responsabilidad única



```
const { Client } = require("pg"); // Cannot find name 'require'. Do you need to install type definitions for this package?
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;
}

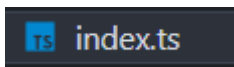
let client;

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[]> { // An as
  return await client.query(query, bind);
}
```

Carpeta inversify

En este caso nos encontramos con un archivo que contiene varias dependencias, por lo que se puede notar podemos concluir que se utiliza tanto un patrón de inyección de dependencias como de singleton, en cuanto a principios por su naturaleza se usa el de responsabilidad única ya que dentro del container cada declaración sólo tiene una responsabilidad así como una instancia



```
import { Container } from 'inversify';    Cannot find module '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';

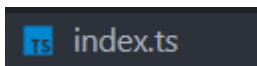
const container = new Container({ defaultScope: 'Singleton' });

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();
```

Carpeta symbols

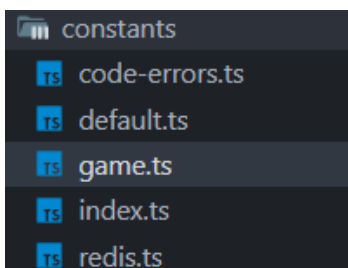
En el caso de este archivo se puede notar un patrón de inyección de dependencias, ya que se están declarando como si fueran tipos/símbolos cada una de ellas, para poder ser inyectadas, de la misma razón cumple con el principio de responsabilidad única ya que lo único que hace es definir los símbolos para las inyecciones

index.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")  
}
```

Carpeta constants

El caso de esta carpeta es muy peculiar ya que se trata de una carpeta con la única responsabilidad de declarar constantes que se usarán a lo largo del proyecto, por lo que el único principio que puede ser usado es el de responsabilidad única, debido a que solo se declaran variables constantes, esta carpeta podría ser situada en variables o procesos de desarrollo dentro del sistema

constants
code-errors.ts
default.ts
game.ts
index.ts
redis.ts

```
const CODE_ERRORS = {  
  ⚡UNAUTHORIZED: 'UNAUTHORIZED',  
};  
  
export { CODE_ERRORS };
```

```
export const ERROR_MESSAGE = 'Internal Server  
⚡
```

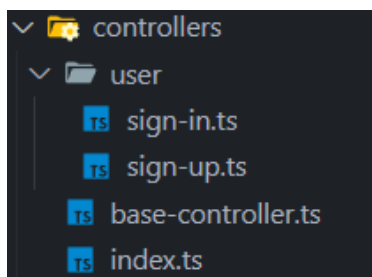
```
export const game = {  
  status: {  
    victory: "VICTORY",  
    gameOver: "LOOSER",  
  },  
  messages: {  
    wordInvalid: "Word Invalid",  
    initGameRequired: "Init Game Needed",  
    gameOver: "Game Over",  
  },  
};|
```

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

```
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;
```

Carpeta controllers

Esta carpeta al contener diversos controladores con los que el usuario va a interactuar se sitúa en la capa de presentación, ya que maneja directamente solicitudes provenientes del usuario



Carpeta controller/user

Este archivo define una clase que sirve para realizar el inicio de sesión dentro de la aplicación, por variables que ya vimos anteriormente como el @injectable podemos concluir que tiene inyección de dependencias, al igual que sigue principios de responsabilidad única dentro de su funcionamiento, al igual que cuenta con inversión de control, ya que la clase no es responsable como tal de crear las dependencias sino que serán creadas desde otro sitio

 sign-in.ts

```
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(

```

En el caso de este archivo realiza y tiene los mismos principios tanto de inversión, como de responsabilidad única e inyección de dependencias, ya que sigue el mismo flujo que el anterior solo que en este caso realiza una operación diferente como lo es registrar

 sign-up.ts


```

import { SignupUseCase } 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 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;    Property 'body' does not exist on type 'CustomRequest'


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

```

Carpeta controller


Este archivo lo que hace es encargarse del manejo de las peticiones y estatus de respuestas, al implementar funciones, se usó el principio de responsabilidad única, así como el de sustitución de liskov ya que tiene la posibilidad de sustituir sus clases sin alterar el funcionamiento

 base-controller.ts

```
export abstract class BaseController {
  protected ok<T>(
    res: Response,
    httpCode: number,
    dto?: T
  ): Response {
    console.info(
      `[END] - Path: ${req.originalUrl}`, Property 'originalUrl' does not exist on type 'Request',
      BaseController.name, Property 'name' does not exist on type 'BaseController',
      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;
    }
    console.info(
      `[END] - Path: ${req.originalUrl}`, Property 'originalUrl' does not exist on type 'Request',
      BaseController.name, Property 'name' does not exist on type 'BaseController',
      req.requestId
    );
    return errorHandler(error, req, res);
  }
}
```

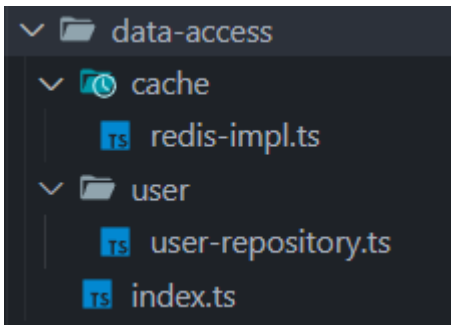
Es otro archivo conocido como barril de dependencias

 index.ts

```
export * from './user/sign-up';  
export * from './user/sign-in';
```


Carpeta data-access

Debido a que esta carpeta proporciona funciones para acceder a datos la podemos situar dentro de la capa de acceso de datos de nuestra arquitectura



carpeta data-access/cache

Esta clase implementa la interfaz de caché dando funcionalidades, por lo que tenemos inyección de dependencias, y principios de responsabilidad única ya que tenemos funciones encargadas de una sola cosa, y a su vez la inversión de control ya que a lo largo del proyecto podemos ver que las clases no crean sus propias dependencias sino que son creadas fuera de estas

 redis-impl.ts

```

export class RedisImpl implements CacheService {

  async getKeyTLL(key: string): Promise<number> {  An async function or method
    const client = getRedisClient();
    return await client.ttl(key);
  }

  async getByKey<T>(key: string): Promise<T | null> {  An async function or method
    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);  Properly
      const record = await this.getByKey<T>(recordKey);
      if (record && typeof record === 'string') {
        try {
          return JSON.parse(record);
        } catch (err) {
          return record;
        }
      }
    };
  }
}

```

carpeta data-access/user

Este archivo se encarga de usar la interfaz de repositorio para acceder con la base de datos de usuarios, creando y accediendo, lo que nos da un patrón de repositorio, que es básicamente abstraer la lógica de las operaciones para el almacenamiento y que la aplicación no las conozca, también tenemos principios de responsabilidad única ya que cada función hace una sola cosa

 user-repository.ts

```

@Inject()
export class UserRepositoryImpl implements UserRepository {

  async create(data: User): Promise<void> {    An async function
    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;

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

    return null;
  }
}

```

Tenemos un archivo barril para el data-acces, todos nuestros archivos de barril siguen un principio de ocultar información, esto proveniente de POO, en el que se sugiere que los detalles de implementación deben de estar ocultos

 index.ts

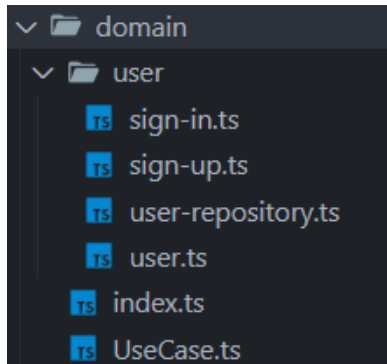
```

export { RedisImpl } from './cache/redis-imp';
export * from './user/user-repository';

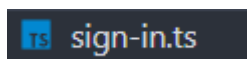
```

domain

Dentro de esta carpeta nos encontramos con la capa de dominio o también referenciada como capa de la logica de negocio



En el caso de este archivo sirve para manejar el proceso del inicio de sesión, al tener varias funciones y por cómo está hecha, maneja un principio de responsabilidad única, así como una inversión de control ya que no se crean la dependencias dentro de la clase, asimismo también se usa un principio de sustitución ya que sign puede ser sustituido por significase sin afectar el proceso



```
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, user.password);
      const generateToken = this.authService.generateToken(user);

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

Tenemos el mismo caso en cuanto a principios y patrón, al igual con inyección de dependencias ya que sigue la misma estructura pero ahora se encarga de manejar todo el proceso de registro

 sign-up.ts

```
interface responseDto {
  userName: string;
}

export type SignupUseCase = UseCase<requestDto, responseDto>;

@Injectable()
export class Signup implements SignupUseCase {
  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;
      console.log({ requestDto });
      const hashPassword = this.authService.hash(password);
      await this.userRepository.create({
        userName,
        password: hashPassword,
      });

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

En este archivo tenemos una creación de interfaz, por la manera en que está implementado se maneja un patrón de repositorio, que solamente abstrae las funciones de almacenamiento, esto permitiendo que la aplicación interactúe con la base de datos sin conocer como, así también se usa un principio de inversión de dependencias, ya que se usa para que las clases de alto nivel accedan, igual el principio de sustitución, ya que se puede cambiar la interfaz implementada sin complicación alguna

TS user-repository.ts

```
import { User } from "../user";

export interface UserRepository {
  create(data: User): Promise<void>;
  getUserByUserName(userName: string): Promise<User>;
}
```


Este es un archivo donde se exporta un tipo de variable/objeto y se declara su contenido, por lo que tiene principios de responsabilidad única y de inversión de dependencias

TS user.ts

```
export type User = {
  id?: string;
  userName: string;
  password: string;
}
```



domain

Este archivo es un archivo barril de dependencias

 index.ts

```
export { UseCase } from './UseCase';  
export * from './user/user-repository';  
export * from './user/sign-up';  
export * from './user/sign-in';
```




En el caso de este archivo tenemos una creación de interfaz que define un caso de uso, por cómo está construido cuenta con principios de sustitución al poder cambiar su use case, inversión de dependencias al ser creado para acceder por clases de alto nivel a él, y de segregación ya que es una interfaz pequeña y específica

 UseCase.ts


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

Carpeta interfaces

Esta carpeta por su contenido se encuentra situada en la capa de lógica de negocio, aunque pueda parecer contraintuitivo, ya que maneja errores


▼  interfaces
  custom-error.ts
  index.ts

Tenemos otro archivo donde se crea una interfaz, por lo que tenemos inversión de dependencias, por como funciona se tiene responsabilidad única y por su estructura puede funcionar si se cambia el objeto de error por cualquier otro sin afectar

 custom-error.ts

```
export interface IJsonObject {  
  ⚡[key: string]: string;  
}  
  
export interface ICustomError extends Error {  
  code: string;  
  vars?: IJsonObject;  
  httpCode: number  
}
```

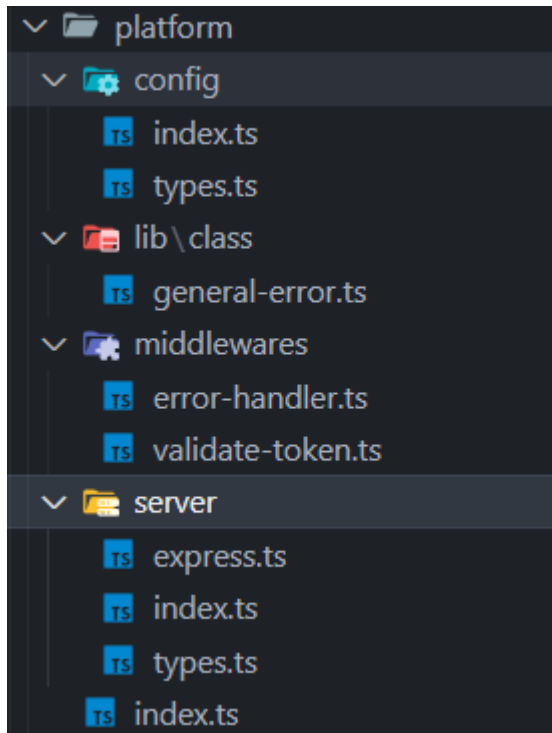
Este archivo es similar al barril, solamente que en este caso se realizan varias exportaciones para ser utilizadas en otros sitios, este utiliza un patrón de módulo, que se basa en encapsular código relacionado en un solo lugar, así como un principio de responsabilidad única al solo exportar y un principio de ocultación proveniente de POO esto para mantener la información de implementación oculta

 index.ts

```
import { ICustomError, IJsonObject } from '  
⚡import { IGetEntityOlimpoRepository } from '  
import { IRequestOlimpoRepository } from '  
  
export {  
  ICustomError,  
  IJsonObject,  
  IGetEntityOlimpoRepository,  
  IRequestOlimpoRepository,  
};
```

Carpeta platform

Esta carpeta por lo que contiene, en una clean architecture pertenecería a la capa de infraestructura que es la encargada de implementar interfaces definidas en capas superiores, es básicamente el cómo interactúan con los detalles técnicos



Carpeta platform/config


En cuanto a este archivo nos encontramos un módulo de configuración, ya que se define y exporta un módulo de configuración con variables, por lo mismo se usa un patrón de modulo que encapsula todo el código relacionado, al solo ser un objeto de conf se tiene un responsabilidad única

 index.ts

```
import type { Config } from './types';
const config: Config = {
  environment: process.env.ENVIRONMENT || "local",
  port: +process.env.PORT || 8080,
  host: process.env.HOST || "0.0.0.0",
  cache: {
    host: process.env.REDIS_HOST,
    port: process.env.REDIS_PORT,
    password: process.env.REDIS_PASSWORD,
  },
  db: {
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    userName: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    dbName: process.env.DB_NAME,
  },
  project: {
    appSecret: '0I851HFX0000112'
  },
  auth: {
    secret: 'wordle',
    salts: 10,
  }
};

export default config;
```

Al igual que el módulo anterior nos encontramos con un modulo de configuracion pero en este caso es de variables definidas, por lo mismo seguimos teniendo un patrón de módulo y un principio de responsabilidad única, podríamos mencionar un principio de inversión de dependencias ya que se busca que se acceda desde una clase de alto nivel a este archivo

 types.ts


```

export type Config = {
  readonly environment: string;
  readonly port: number;
  readonly host: string;
  cache: {
    (property) host: string;
    readonly host: string;
    readonly port: string;
    readonly password: string;
  };
  db: {
    readonly host: string;
    readonly port: string;
    readonly dbName: string;
    readonly userName: string;
    readonly password: string;
  };
  project: {
    readonly appSecret: string;
  };
  auth: {
    readonly secret: string;
    readonly salts: number;
  };
};

```

Carpeta platform/libs

En este archivo lo se define una clase de customerror, por lo que nos percatamos de que maneja una principio de inversión de dependencias ya que de la misma manera se está creando una interfaz dentro del archivo, a su vez cada función del archivo tiene un principio de responsabilidad única y por su estructura podemos usar el principio de sustitucion sin ningun problema mediante la implementación de custom error

 general-error.ts

```

import { IJsonObject } from '@example-api/interfaces';

interface ICustomErrorParams {
  code: string;
  message?: string;
  vars?: IJsonObject;
  httpCode?: number;
}

export class CustomError extends Error {
  /**
   * Create custom errors
   *
   * @param code (string) error code registered in the cms
   * @param message (string) error custom message
   * @param vars (object) variables that can be interpolat
   */

  public code: string;
  public message: string;
  public vars: IJsonObject;
  public httpCode: number;

  constructor(public params: ICustomErrorParams) {
    super(params.code);

    this.code = params.code;
    this.message = params.message;
    this.vars = params.vars;
    this.httpCode = params.httpCode;
    this.name = CustomError.name; Property 'name' does
    Object.setPrototypeOf(this, CustomError.prototype);
  }
}

```

Carpeta platform/middleware

Ya que nos encontramos dentro de la carpeta de middleware podemos intuir que nos encontraremos con archivos que intercepten peticiones, en este caso tenemos un patrón con el mismo nombre que define el comportamiento de un middleware, así mismo en cada etapa se implementó el principio de responsabilidad única, y por su estructura se puede modificar el objeto de error sin inconveniente por lo que se usó el principio de sustitución, y al ser un middleware se plantea su uso en una clase de nivel mayor por lo que también tenemos inversión de dependencias

```

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);
};

```

En este caso tenemos la misma implementación de patrones y principios solo que cambia el tipo de middleware, mientras en el otro se manejaban errores, este se encarga de validar tokens

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';

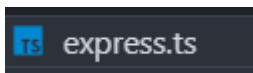
const { HTTP_STATUS_UNAUTHORIZED } = constants;

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

```

Carpeta platform/server

en este caso nos encontramos directamente con la parte encargada de levantar un servidor en express que se encargará de recibir peticiones, dentro de todas sus funciones de uso el principio de responsabilidad única, dentro de este mismo al hacer uso de otros archivos tenemos un patrón de middleware al implementar estos, así como una inversión de dependencias al hacer uso de abstracciones como el requesthandler




```
export const startExpressServer = (
  handlers: RequestHandler | RequestHandler[],
  options: IStartOptions
) => {
  const { basePath, port, host, requestId } = options;
  const app: Application = express();
  app.use((req: CustomRequest, res: Response, next: NextFunction) => {
    const { requestId, accessToken } = transformHeadersToCamelCase(req);
    req.requestId = (requestId as string) || uuidv4();
    req.accessToken = accessToken as string;
    console.info(
      `[START] - Path: ${req.originalUrl}`, Property 'originalUrl'
      req.requestId
    );
    next();
  });
  app.use(express.json({ limit: '10mb' }));
  app.use(express.urlencoded({ extended: false }));
  app.use(cors());
  app.use(helmet());
  app.use(compression());

  app.use(basePath, handlers);


  app.use((err: Error, req: CustomRequest, res: Response, next: NextFunction) => {
    errorHandlerMiddleware(err, req, res, next);
  });

  app.listen(port, host, async () => {
    // if postgres is available, init a connection pool for the server
    if (config.db.host) {
      connect(requestId);
    }
    // if redis is available, init the connection to Redis for the server
    // Use getRedisClient to get the client to perform operations on the
  });
}
```


Es un archivo de exportaciones, solamente para poder exportar más archivos desde uno solo y ocultarlos de los procesos, esto tanto en el que está dentro de la carpeta server como en el que está directamente en platform


 index.ts

```
export { startExpressServer } fr
export { CustomRequest, IStartOp
```

 index.ts

```
import config from './config';
import { startExpressServer, Customl
export { config, startExpressServer
```

Al igual que uno de los archivos anteriores tenemos un archivo de configuración de variables de un objeto, esto nos da un uso de principios como el de responsabilidad única como en de inversión de dependencias, así como un patrón de modulo, ya que tenemos un encapsulamiento de código

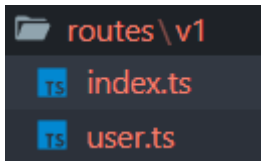
 types.ts

```
import { Request } from 'express'; Cannot f
export type DataToken = {
  data: {
    userId: number;
    userName: string;
  };
};
export interface CustomRequest extends Request
  (property) CustomRequest.sourceApp?: string
  sourceApp?: string | null;
  versionApp?: string | null;
  accessToken?: string | null;
}

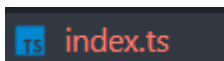
export interface IStartOptions {
  basePath: string;
  port: number;
  host: string;
  requestId: string;
  corsOrigin?: string | string[];
}
```

Carpeta routes

La carpeta de routes la encontramos dentro de la capa de lógica de negocio ya que interactúa con las reglas que se tienen en el programa como las rutas y sus funciones




En cuanto a patrones, en este archivo se utiliza el de módulo ya que encapsulamos el código relacionado en un solo lugar, al tener una función que hace una sola cosa tenemos el principio de responsabilidad única presente, así como una inversión de dependencias, ya que no se depende de un módulo de bajo nivel



```
import { Router } from 'express';  
import { userRouter } from './user';  
  
const v1Routes = Router();  
  
v1Routes.use("/v1/user", userRouter);  
  
export { v1Routes };  
|
```

En este caso tenemos algo similar al archivo anterior pero en este caso se manejan todas las rutas relacionadas al usuario, por lo que mantenemos el patrón de módulo y los principios de responsabilidad única y de inversión de dependencias, pero al hacer uso de otras dependencias como los controllers agregamos una inyección de dependencias

 user.ts

```
import { Router } from "express";    Cannot find module 'express' or its
import { container } from "@example-api/config/inversify";
import {
  UserSignupController,
  UserSigninController,
} from "@example-api/controllers";

const signup = container.get(UserSignupController);
const signin = container.get(UserSigninController);

const userRouter = Router();

userRouter.post("/signin", (req, res) => signin.execute(req, res));




userRouter.post("/signup", (req, res) => signup.execute(req, res));

export { userRouter };
```


Carpeta util

Las carpetas útiles dentro de la mayoría de proyectos se usan exclusivamente para destinar un lugar a funciones sueltas que se usan a lo largo del proyecto

util


-  index.ts
-  promises.ts
-  index.ts

En este caso tenemos un archivo de barril

 index.ts

```
import config from './config';    Cannot find module './config'
import { startExpressServer, CustomRequest } from './server';
import { config, startExpressServer, CustomRequest };
```

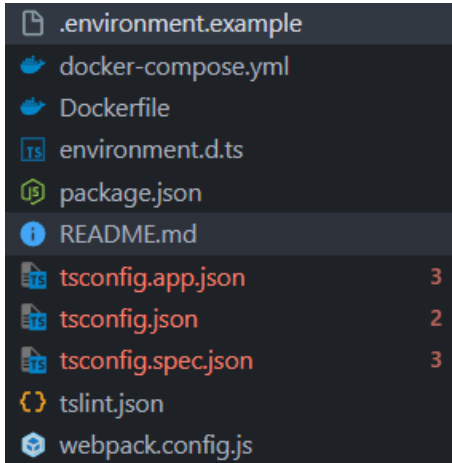
En este caso parece ser una función que maneja un arreglo de funciones, por lo que sigue un patrón de módulo al encapsular el código relacionado a esto, así como principio de sustitución al ser capaz de cambiar el objeto promise sin inconvenientes, y responsabilidad única, también tenemos inversión de dependencias con el promise al ser la abstracción de la que se depende

 promises.ts

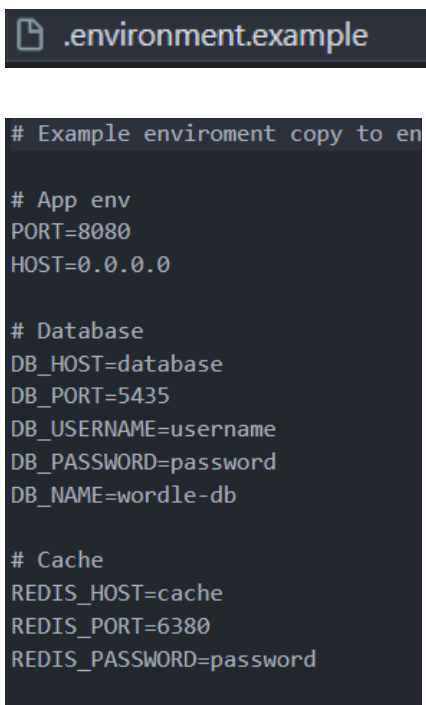
```
/**
 * Execute array of promises pending and return array of promises exec
 *
 * @param promisesArray Array of promises pending for execute
 * @param options Object with options for execute promises
 * @returns Array of promises execute successful
 */
export const executeArrayPromises = async <T>({
  promisesArray: Promise<T>[],
  options: executeArrayPromisesOptions = {
    errorDescriptionMessage: 'Result rejected',
    applyFlat: false,
  }
}) => {
  const promisesResult = await Promise.allSettled(promisesArray);
  const promisesExecutedSuccessful: Awaited<T>[] = [];
  promisesResult.forEach(result => {
    if (result.status === 'fulfilled' && result.value) {
      promisesExecutedSuccessful.push(result.value);
    } else {
      console.error(
        `${options.errorDescriptionMessage}`,
        inspect(result, { depth: 2 })
      );
    }
  });
  if (options.applyFlat) {
    return promisesExecutedSuccessful.flat();    Property 'flat' does
  }
  return promisesExecutedSuccessful;
}
```

Archivos dentro del root del template

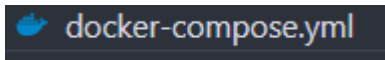
En este caso podemos ver que todos los archivos que se encuentran aquí son de configuración del proyecto



Cabe destacar que aun siendo archivos de configuración alteran el proceso del proyecto por lo que pueden estar ligados a capas como la capa de acceso en este caso, aunque no esté directamente en la capa, se hace uso dentro de ella, al ser un .env, tenemos un patrón de configuracion externa que especifica tener los datos de configuración sensibles de manera externa al código, así como un principio de responsabilidad única ya que solo define variables



Nos encontramos con un archivo de configuración de docker, por lo que seguimos con un patrón de configuración externa y un principio de responsabilidad única, aunque también podemos ver como una encapsulación ya que aísla el entorno de ejecución para el servicio



```
version: '3.9'

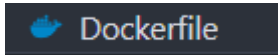
volumes:
  cache:
    driver: local
  database:
    driver: local

services:
  cache:
    image: redis
    command: redis-server --save 60 1 --loglevel warning --port ${REDIS_PORT}
    ports:
      - '${REDIS_PORT}:${REDIS_PORT}'
    volumes:
      - cache:/data

  database:
    image: postgres:13
    ports:
      - '${DB_PORT}:${DB_PORT}'
    volumes:
      - database:/data
    environment:
      POSTGRES_USER: '${DB_USERNAME}'
      POSTGRES_PASSWORD: '${DB_PASSWORD}'
      POSTGRES_DB: '${DB_NAME}'
      PGPORT: '${DB_PORT}'

  migrations:
    build:
      context: .
      dockerfile: Dockerfile
      target: base
    depends_on:
      - database
    environment:
      DATABASE_URL: 'postgres://${DB_USERNAME}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}'
    volumes:
```

Este al ser el archivo para la construcción del docker cuenta con un patrón de construcción multi etapa, el cual garantiza una construcción más eficiente al dividir la construcción en etapas, y ya que cada comando se usa para algo en específico tenemos una responsabilidad única



```
FROM node:16 as base

RUN curl -Lo /usr/bin/dbmate https://github.com/amacneil/dbmate/releases
    && chmod +x /usr/bin/dbmate

WORKDIR /app

CMD [ "npm", "run", "watch" ]

EXPOSE 8080

FROM node:16 as builder

WORKDIR /app

COPY --from=base /usr/bin/dbmate /usr/bin/dbmate

COPY package*.json ./
RUN npm ci

COPY . .

RUN npm run build

FROM node:16

WORKDIR /app

COPY --from=base /usr/bin/dbmate /usr/bin/dbmate

COPY package*.json ./
RUN npm ci --only=production --prefer-online

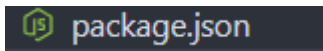
COPY --from=builder /app/dist .
COPY --from=builder /app/db ./db
```

En este archivo tenemos una declaración de variables por lo que contamos con un principio de responsabilidad única

`environment.d.ts`

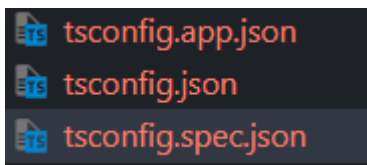
```
declare global {  
  namespace NodeJS {  
    interface ProcessEnv {  
      PORT: string;  
      HOST: string;  
  
      DB_HOST: string;  
      DB_PORT: string;  
      DB_USERNAME: string;  
      DB_PASSWORD: string;  
      DB_NAME: string;  
      DB_TYPE: string;  
      DB_POOL_SIZE: string;  
      DB_SCHEMA_LOCKED: string;  
      CONN_TIMEOUT: string;  
      IDLE_TIMEOUT: string;  
  
      REDIS_HOST: string;  
      REDIS_PORT: string;  
      REDIS_PASSWORD: string;  
      REDIS_TLS: string;  
    }  
  }  
}  
  
// If this file has no import/export  
// convert it into a module by a  
export {};
```


Llegamos al archivo de configuración del proyecto, este archivo tiene la responsabilidad de servir como guía de dependencias e información así como de scripts del proyecto, al verlo de esta forma le podemos otorgar principios como el de responsabilidad única, así como un patrón de módulo, ya que encapsula todo lo relacionado al proyecto




```
{
  "name": "example-api",
  "version": "1.0.0",
  "description": "",
  "main": "dist/index.js",
  "scripts": {
    "lint": "eslint --ignore-path .eslintignore --ext .ts .",
    "format": "prettier --ignore-path .gitignore --write \"**/*.+(js|t)",
    "cleanup": "npm run format; npm run lint",
    "watch": "webpack --watch --config webpack.config.js",
    "build": "webpack",
    "dev": "docker compose --env-file .environment.local up app",
    "start": "docker compose --env-file .environment.local up app-build",
    "dbmate": "docker compose --env-file .environment.local run --rm -",
    "dbmate:seed": "docker compose --env-file .environment.local run -",
    "migrations": "docker compose --env-file .environment.local run --",
  },
  "repository": {
    "type": "git",
    "url": "git@gitlab.com:"
  },
  "keywords": [],
  "dependencies": {
    "axios": "^0.27.2",
    "bcryptjs": "^2.4.3",
    "cerealizr": "^1.0.1-alpha",
    "chalk": "^4.0.0",
    "compression": "^1.7.4",
    "cors": "^2.8.5",
    "express": "^4.17.2",
    "helmet": "^5.0.1",
    "hot-shots": "^9.0.0",
    "inversify": "^6.0.1",
    "ioredis": "^4.28.3",
    "joi": "^17.6.1",
    "jsonwebtoken": "^8.5.1",
    "lodash": "^4.17.21",
    "pg": "^8.7.1",
    "reflect-metadata": "^0.1.13"
  }
}
```

En cuanto a los siguientes 3 archivos al ser archivos de configuración de typescript siguen el mismo principio de responsabilidad única como el patrón de configuración externa, especificando ciertas reglas que seguirá a la hora de verificar el lenguaje, lo que incluso podría llegar a afectar a la lógica de negocio

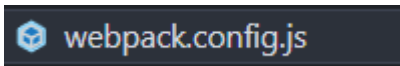


Seguimos manteniendo el principio de responsabilidad única como el patrón de configuración externa, solo que en este caso al ser un archivo de configuración para la codificación se sigue también un principio de mejores prácticas, garantizando que al usar estas reglas definidas el código siga las mejores prácticas

 tslint.json

```
{
  "rulesDirectory": [
    "node_modules/@nrwl/workspace/src/tslint",
    "node_modules/codelyzer"
  ],
  "linterOptions": {
    "exclude": [
      "**/*.ts",
      "tools/schematics/sls-schematic/files",
      "tools/schematics/crud-schematic/files"
    ]
  },
  "rules": {
    "arrow-return-shorthand": true,
    "callable-types": true,
    "class-name": true,
    "deprecation": {
      "severity": "warn"
    },
    "forin": true,
    "import-blacklist": [true, "rxjs/Rx"],
    "interface-over-type-literal": true,
    "member-access": false,
    "member-ordering": [
      true,
      {
        "order": [
          "static-field",
          "instance-field",
          "static-method",
          "instance-method"
        ]
      }
    ],
    "no-arg": true,
    "no-bitwise": true,
    "no-console": [true, "debug", "info", "time", "timeEnd", "trace"],
    "no-construct": true,
    "no-debugger": true,
  }
}
```

Al final tenemos el archivo de configuración para empaquetar la aplicación, lo que permite configurar el cómo se empaquetan los módulos, ya que sigue siendo una configuración, tenemos el patrón de configuración externa y el principio de responsabilidad única



```
module.exports = (async () => {
  return {
    entry: {
      index: [path.resolve(__dirname, '/src/index.ts')],
    },
    output: {
      libraryTarget: 'commonjs',
      path: path.join(__dirname, 'dist'),
      filename: 'index.js',
    },
    node: {
      __dirname: true,
    },
    stats: 'minimal',
    target: 'node',
    mode: 'development',
    externals: [nodeExternals(), 'dd-trace'],
    devtool: 'nosources-source-map',
    module: {
      rules: [
        {
          test: /\.ts$/,
          exclude: [
            path.resolve(__dirname, 'node_modules'),
            path.resolve(__dirname, '.webpack'),
            path.resolve(__dirname, 'dist'),
          ],
          loader: 'ts-loader',
          options: {
            transpileOnly: true,
          },
        },
      ],
    },
    plugins: [
      new NodeemonPlugin({
        args: process.env.ARGS ? process.env.ARGS.split(' ') : ''
      })
    ]
  }
})()
```

Posibles mejoras:

La únicas mejoras o mas bien implementaciones que podría sugerir seria en cuanto al contenido de routes, ya que se pueden implementar los middlewares en esta sección para realizar validaciones así como manejar errores

Referencias:

<https://github.com/jorggerojas/design-patterns-2024.git>