UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

Facultad de Informática

Proyecto Final Patrones de diseño

Angel Ryosuke Ishiwatari Domínguez

315443

sábado 18 de mayo del 2024

Pequeña presentación

Este documento examina la aplicación de patrones de diseño discutidos en clase, las diversas capas del modelo de arquitectura limpia, la implementación de los principios SOLID y la manifestación de los cuatro pilares de la programación orientada a objetos. Para facilitar la revisión, se ha optado por un formato tabular organizado por archivo. De este modo, se logra una presentación clara y accesible de la información relevante.

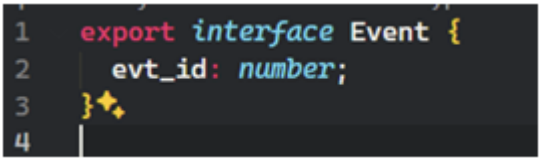
1. Carpeta “Common”.

Esta carpeta alberga servicios cruciales, incluyendo la gestión de caché y procesos de autenticación. Además, ofrece herramientas prácticas y clases estándar utilizadas a lo largo de todo el proyecto. a. Archivo "util.ts": Este documento contiene un conjunto de herramientas versátiles diseñadas para ser aprovechadas en diversas partes del proyecto.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Se asegura que todas las funciones y variables estén adecuadamente aisladas dentro de sus métodos correspondientes, evitando el acceso directo desde el exterior.
Abstracción.	Se emplea una interfaz para establecer un conjunto de métodos que posteriormente se implementan en distintas secciones del código.
Clean architecture	
N/A	No se encontró aplicaciones.
Principios SOLID	
Single Responsibility principle	Cada método está diseñado para llevar a cabo solo una tarea específica.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

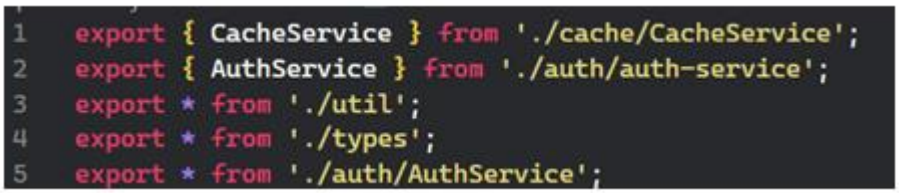
b. Archivo “types.ts”

En este documento únicamente se está definiendo una interfaz “Event” que contiene un atrivuto evt_id de tipo number.



c. Archivo “index.ts”

Este documento únicamente exporta las funciones además de los módulos que serán utilizados dentro de la carpeta common.



d. Subcarpeta “auth”

i. Archivo “auth-service.ts”

En este archivo se implementa una clase llamada “AuthService” Con la cual podemos automatizar usuarios y generar tokens. Metodos: Generacion de un token JWT, verificación de un token JWT, verificación de la coincidencia de contraseña así como de la generación de hash de contraseñas.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Los métodos de la clase "AuthService" están encapsulados y solo son accesibles a través de la instancia de la clase.
Abstracción.	La clase "AuthService" es una abstracción de los servicios de autenticación, generación de tokens, verificación de tokens y manejo de contraseñas.
Herencia	La clase "AuthService" no hereda de ninguna otra clase, pero implementa la interfaz "IAuthService".

Polimorfismo	La clase "AuthService" implementa la interfaz "IAuthService", lo que permite que se utilice en diferentes contextos sin afectar la funcionalidad de la aplicación.
Clean architecture	
N/A	No se encontró aplicaciones.
Principios SOLID	
Single Responsibility principle	La clase "AuthService" tiene múltiples responsabilidades, como generar tokens, verificar tokens, comparar contraseñas y hashear contraseñas. Esto viola el SRP.
Dependency Inversion Principle	La clase "AuthService" implementa una interfaz llamada "IAuthService", lo que significa que depende de una abstracción en lugar de una implementación concreta.
Patrón de diseño.	
N/A	No se encontró aplicaciones.
Áreas de oportunidad	
Podría ser útil separar las responsabilidades de la clase "AuthService" en clases más pequeñas y específicas, cada una encargada de una tarea específica (por ejemplo, una clase TokenService para generar y verificar tokens, y una clase PasswordService para hashear y comparar contraseñas).	

ii. AuthService.ts

Se exporta la interfaz con la cual se implementará para realizar el servicio de implementación.

```
1  export interface IAuthService {
2    generateToken(data: any): any;
3    verifyToken(token: string): any;
4    matchPassword(password, hash);
5    hashPassword(plainPassword): string;
6  }
```

e. Subcarpeta “cache”

i. Archivo “CacheService.ts”

Se busca establecer la interfaz para un servicio de almacenamiento en caché. Este servicio se usa para guardar datos temporales para optimizar el desempeño de la aplicación.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Los métodos definidos en la interfaz y sus implementaciones están encapsulados para manejar la gestión de datos en caché.
Abstracción.	Se logra mediante la definición de la interfaz, la cual especifica los métodos necesarios para interactuar con el servicio de caché, sin necesidad de conocer los detalles de su implementación concreta.
Polimorfismo.	Se aplica la capacidad de las implementaciones de la interfaz para permitir que el cliente interactúe con diferentes implementaciones de servicios de caché, siempre y cuando se ajusten a la misma interfaz.
Clean architecture	
Infraestructura	Se encarga de manejar la persistencia temporal de datos para mejorar la eficiencia de la aplicación.
Principios SOLID	
Single Responsibility principle	La interfaz solo define los métodos relacionados con la gestión de caché, lo que garantiza que la interfaz tenga una sola responsabilidad.
Dependency Inversion Principle	La interfaz define la estructura necesaria para interactuar con un servicio de caché, sin necesidad de conocer cómo se implementa internamente.
Interface Segregation Principle	La interfaz sólo define los métodos necesarios para interactuar con un servicio de caché, lo que garantiza que las implementaciones de la interfaz sólo implementen los métodos necesarios.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

2. Carpeta “Config”.

El propósito principal de esta carpeta es gestionar la configuración y el inicio de diversos componentes que conforman la aplicación. Su función específica radica en configurar y operar la base de datos, implementar el inversor de control (IoC) para facilitar la inyección de dependencias dentro de la aplicación, y establecer los símbolos que actúan como identificadores únicos para las dependencias dentro del contenedor IoC.

a. Subarpeta “db”

i. Archivo “redis.ts”

Se generan y exportan funciones que trabajan con redis (Sistema de almacenamiento en memoria basado en clave valor)

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Se maneja de una manera correcta las variables, por loc aul no se puede acceder a ellas desde fuera
Abstracción.	Se utiliza una abstaraccion (interfaz) con la cual se define el tipo de retorno de la funcion “getRedisClient”
Clean architecture	
N/A	No se encontro aplicacion
Principios SOLID	
Single Responsibility principle	Hay funciones como “getRedisClient” y ” connectRedisClient” las cuales realizan una única acción (tienen una sola responsabilidad)
Dependency Inversion Principle	Se utiliza una interfaz “REdis.Redis” en lugar de una implementacion concreta en la funcion “connectRedisClient”
Patrón de diseño.	
N/A	No se encontró aplicaciones.

ii. Archivo “sql.ts”

Se utiliza un cliente pg de Node.js con el cual se logra interactuar con la base de datos SQL (PostgreSQL)

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Las variables y funciones se encuentran adecuadamente encapsuladas dentro del archivo "sql.ts" y no se accede a ellas directamente desde fuera.
Abstracción.	El archivo "sql.ts" utiliza las interfaces DBSQLArguments y DBReplyDataRow para definir los tipos utilizados en las funciones.
Clean architecture	
N/A	No se encontro aplicacion
Principios SOLID	
Single Responsibility principle	Se encuentran las funciones “connect” y “sql” las cuales cuentan con una única responsabilidad cada una.
Dependency Inversion Principle	La función sql depende de abstracciones (DBSQLArguments y DBReplyDataRow) en lugar de implementaciones concretas.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

b. Subarpeta “inversify”

i. Archivo “index.ts”

Este archivo establece la configuración del contenedor IoC por medio de InversifyJS, asignando dependencias y vinculaciones entre implementaciones específicas y tipos abstractos.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Las dependencias y sus relaciones están confinadas dentro del contenedor, sin acceso directo externo.
Abstracción.	Interfaces y clases abstractas se usan para detallar dependencias y enlaces dentro del contenedor IoC.
Clean architecture	
N/A	No se encontró aplicaciones.
Principios SOLID	
Single Responsibility principle	Centraliza la definición y asociación de dependencias, aislando la configuración de inyección de dependencias del resto del código.
Dependency Inversion Principle	Se emplea el contenedor IoC para inyectar dependencias, evitando su creación directa en las clases, lo que mejora la flexibilidad, simplifica las pruebas y facilita el cambio de implementaciones.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

c. Subarpeta “symbols”

i. Archivo “index.ts”

Este archivo genera una serie de identificadores exclusivos, denominados símbolos, para las dependencias presentes en el contenedor IoC. Estos identificadores establecen el vínculo entre las interfaces o clases genéricas y sus respectivas implementaciones específicas dentro del contenedor.

```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"),
};

```

3. Carpeta “Constant”.

Esta carpeta contiene archivos que definen constantes o valores fijos utilizados en el proyecto. Estas constantes representan valores que se aplican en diversas secciones del código y se mantienen invariables mientras el programa se encuentra en ejecución. No se requiere un análisis exhaustivo de estos archivos.

```ts
/**
 * 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;

```

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

```ts
export const ERROR_MESSAGE = 'Internal Server Error';

```

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

```

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

export { CODE_ERRORS };

```

4. Carpeta “controllers”.

El propósito de esta carpeta es albergar los controladores de la aplicación. Estos controladores desempeñan la función crucial de gestionar las solicitudes HTTP entrantes, procesarlas y generar las respuestas adecuadas para enviar de vuelta al cliente.

a. Subcarpeta “users”

i. Archivo “sign-in.ts”

Este archivo describe cómo se maneja el proceso de inicio de sesión en la aplicación. El archivo contiene un controlador que recibe y procesa las peticiones de inicio de sesión, y luego responde adecuadamente.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Se protegen los datos internos mediante propiedades y métodos privados.
Abstracción.	El controlador simplifica la interacción con la lógica de inicio de sesión.
Herencia.	Hereda de BaseController para usar funcionalidades comunes.
Polimorfismo.	Implementa el método execute, permitiendo que diferentes controladores ofrezcan su propia versión del método, manteniendo una interfaz común.
Clean architecture	
El controlador está diseñado para mantener separadas sus responsabilidades y usa abstracciones en vez de implementaciones específicas. Obtiene las clases necesarias mediante inyección de dependencias.	
Principios SOLID	
Single Responsibility principle	El controlador se dedica exclusivamente a gestionar las solicitudes de inicio de sesión.
Dependency Inversion Principle	Se basa en abstracciones para la inyección de dependencias, permitiendo que el controlador trabaje con cualquier versión de la dependencia SigninUseCase.
Patrón de diseño.	
Inyeccion de dependencias.	La etiqueta @injectable() indica que la clase UserSigninController puede recibir

	inyecciones de dependencias. Con la etiqueta @inject(SYMBOLS.SigninUseCase) en el constructor, se introduce una versión de SigninUseCase al controlador, que luego se asigna a la propiedad signin. Así, UserSigninController obtiene SigninUseCase por inyección de dependencias, no por creación propia, lo que ayuda a separar las responsabilidades y hace más sencillo probar y mantener el código.

ii. Archivo “sign-up.ts”

Este archivo define una clase base llamada BaseController dentro del directorio de controladores. Dicha clase base proporciona métodos estándar y herramientas para facilitar el manejo de solicitudes HTTP

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Utiliza métodos protegidos para manejar el envío de respuestas y protege su implementación interna.
Abstracción.	Ofrece una interfaz común para que otros controladores hereden y apliquen su propia versión del método execute.
Herencia.	Como clase abstracta, espera ser extendida por otros controladores, permitiendo el uso compartido de funcionalidades.
Polimorfismo.	Define un método abstracto execute para ser implementado por clases derivadas, permitiendo un tratamiento uniforme a través de la interfaz común.
Clean architecture	
BaseController se adhiere a esta arquitectura al dividir responsabilidades y usar abstracciones en vez de soluciones específicas.	
Principios SOLID	
Single Responsibility principle	BaseController provee herramientas estándar para manejar peticiones HTTP, como enviar respuestas correctas o indicar errores.
Dependency Inversion Principle	Diseñada para ser extendida por otros controladores, no depende de soluciones específicas de otros componentes.
Patrón de diseño.	
Inyeccion de dependencias.	De manera similar al archivo “ign-in.ts”, la clase UserSignupController es anotada con @injectable(), y en su constructor se inyecta una instancia de SignupUseCase utilizando la anotación @inject(SYMBOLS.SignupUseCase), lo que permite desacoplar la creación de esta dependencia del controlador y promover la inversión de control.

b. Archivo “base-controller.ts”

Este documento establece una clase abstracta nombrada BaseController que sirve de fundamento para otros controladores en el directorio src/controllers/. Ofrece herramientas y métodos estándar para gestionar peticiones HTTP, incluyendo cómo enviar respuestas correctas o señalar errores.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Define métodos protegidos que manejan el envío de respuestas y protege su implementación interna.
Abstracción.	Provee una interfaz común para que otros controladores hereden y apliquen su propia versión del método execute.
Herencia.	Como clase abstracta, espera ser extendida por otros controladores, lo que permite compartir herramientas comunes.
Polimorfismo.	Establece un método abstracto execute para que las clases derivadas lo implementen, permitiendo un tratamiento uniforme a través de la interfaz común.
Clean architecture	
BaseController se adhiere a estos principios al dividir las tareas y confiar en conceptos generales en vez de soluciones específicas.	
Principios SOLID	
Single Responsibility principle	BaseController se encarga exclusivamente de proveer herramientas estándar para el manejo de peticiones HTTP.
Dependency Inversion Principle	Está pensada para que otros controladores la extiendan y no se basa en soluciones específicas de otros componentes, sino en conceptos generales como CustomRequest y CustomError.
Patrón de diseño.	
N/A	No se encontro ninguna posible aplicacion

c. Archivo “index.ts”

Se exporta el contenido de los archivos contenidos en la carpeta users.

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

5. Carpeta “Data-access”.

La carpeta "data-access" desempeña el rol fundamental de centralizar y administrar todas las operaciones relacionadas con el acceso a datos dentro de la aplicación. Actúa como una capa intermedia que facilita la conexión entre la lógica de negocio y las diversas fuentes de datos, ya sean bases de datos, servicios en línea o archivos del sistema.

a. Subcarpeta “cache”

i. Archivo “redis-impl.ts”

El archivo "redis-impl.ts" alberga una clase denominada RedisImpl. Esta clase representa una implementación específica de un servicio de caché que utiliza Redis. Cuenta con métodos para almacenar y recuperar datos de Redis, tales como obtener el valor asociado a una clave, asignar un valor a una clave, eliminar una clave, entre otros. Además, emplea una técnica conocida como memoización para recordar los resultados de ciertas funciones, lo que contribuye a mejorar el rendimiento general.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	La clase RedisImpl guarda dentro de sí todo lo que tiene que ver con la caché de Redis y ofrece métodos para usar y cambiar los datos.
Abstracción.	La clase RedisImpl usa la interfaz CacheService, lo que permite separar la parte concreta de la implementación y hace posible cambiarla por otra en el futuro.
Polimorfismo.	La clase RedisImpl usa métodos abstractos de su interfaz.
Clean architecture	
El código sigue una estructura organizada y divide bien las tareas usando una técnica donde las partes del código dependen de abstracciones, no de cosas concretas. Esto se hace con la ayuda de la anotación @injectable() y usando Inversify para conectar las partes del código. La clase RedisImpl, que está en la parte de "data-access", se encarga de trabajar con la base de datos Redis.	
Principios SOLID	
Single Responsibility principle	La clase RedisImpl solo se ocupa de las operaciones de caché en Redis.
Dependency Inversion Principle	La clase RedisImpl trabaja con la abstracción CacheService en vez de con Redis directamente, lo que hace que el código sea más adaptable y fácil de cambiar si es necesario.
Patrón de diseño.	
Inyeccion de dependencias	Se usa @injectable() y Inversify para conectar las partes del código de forma clara.
Memento (meorizacion)	Se almacena los resultados de una funcion para que la próxima vez que se llame a esa función con los mismos argumentos, se devuelva el resultado almacenado en caché en lugar de volver a ejecutar la función. El objetivo principal de la memoización es mejorar el rendimiento evitando cálculos redundantes.

b. Subcarpeta "user"

i. Archivo "user-repository.ts"

El archivo "user-repository.ts" contiene una clase llamada UserRepositoryImpl. Esta clase representa una implementación específica del repositorio de usuarios. Proporciona métodos para gestionar los datos de usuarios, tales como agregar un nuevo usuario o buscar un usuario por su nombre. Para interactuar con la base de datos y realizar consultas SQL, utiliza la librería @example-api/config/db.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	UserRepositoryImpl guarda todo lo relacionado con el manejo de datos de

	usuarios y ofrece métodos específicos para esas tareas.
Abstracción.	UserRepositoryImpl trabaja con una interfaz que permite que la implementación específica pueda ser reemplazada en el futuro.
Polimorfismo.	UserRepositoryImpl puede trabajar con diferentes implementaciones de repositorios que siguen la misma interfaz.
Clean architecture	
Se está aplicando la capa repository. La clase UserRepositoryImpl implementa la interfaz UserRepository y se encarga de la interacción con la base de datos, específicamente con la tabla user_account.	
Principios SOLID	
Single Responsibility principle	UserRepositoryImpl se dedica solo a las tareas de manejo de datos de usuarios.
Dependency Inversion Principle	UserRepositoryImpl usa una interfaz llamada UserRepository, lo que hace que el código sea más flexible y fácil de modificar sin afectar otras partes.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

c. Archivo “index.ts”

Este archivo exporta el contenido de la subcarpeta cache así como user.

```
1  export { RedisImpl } from './cache/redis-impl';
2  export * from './user/user-repository';
```

6. Carpeta “Data domain”.

El directorio "domain" constituye el núcleo central del proyecto, albergando la lógica principal y las reglas que rigen el funcionamiento de la aplicación. En este lugar se definen los conceptos y entidades fundamentales relacionados con el área del problema que la aplicación pretende resolver. Además, su función radica en mantener la lógica de negocio separada de los aspectos técnicos de la implementación.

a. Subcarpeta “user”

i. Archivo “Sing-in.ts”

El objetivo principal de este archivo es implementar la funcionalidad de inicio de sesión para un usuario dentro del proyecto. Comprende la lógica necesaria para verificar las credenciales del usuario, generar un token de autenticación y devolverlo como resultado.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Los detalles internos de la clase “Signin” están protegidos y solo se pueden manejar a través de sus métodos públicos.
Abstracción.	Se crean interfaces como “requestDto”, “responseDto” y “SigninUseCase” para definir los tipos de datos y procesos usados.
Polimorfismo.	El método “execute” de la clase “Signin” puede usarse de diferentes maneras gracias a que sobrescribe un método de la interfaz “SigninUseCase”.
Clean architecture	
Se esta utilizando la capa de caso de uso. La clase Signin implementa la interfaz SigninUseCase y encapsula la lógica de negocio para autenticar a un usuario y generar un token de autenticación.	
Principios SOLID	
Single Responsibility principle	La clase “Signin” se encarga exclusivamente del proceso de inicio de sesión, sin mezclar otras funciones.
Dependency Inversion Principle	Se emplean técnicas como la inversión de control y la inyección de dependencias, utilizando decoradores “@inject” e interfaces para evitar la dependencia de componentes específicos.
Patrón de diseño.	
N/A	No se encontró la utilizacion de ningun patron

ii. Archivo “sign-up.ts”

El propósito de este código es llevar a cabo la funcionalidad de registro de usuario dentro del proyecto. Abarca todo lo necesario para agregar un nuevo usuario a la base de datos, haciendo uso del repositorio de usuarios y el servicio de autenticación.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Los detalles internos de la clase “Signin” están protegidos y solo se pueden manejar a través de sus métodos públicos.
Abstracción.	Se crean interfaces como “requestDto”, “responseDto” y “SigninUseCase” para definir los tipos de datos y procesos usados.
Polimorfismo.	El método “execute” de la clase “Signin” puede usarse de diferentes maneras gracias a que sobrescribe un método de la interfaz “SigninUseCase”.
Clean architecture	
Se esta utilizando la capa de caso de uso. La clase Signup implementa la interfaz SignupUseCase y encapsula la lógica de negocio para registrar un nuevo usuario en el sistema.	
Principios SOLID	
Single Responsibility principle	La clase “Signup” se dedica solamente a la tarea de registrar usuarios.
Dependency Inversion Principle	Se emplean técnicas como “@inject” y el uso de interfaces para que el código no dependa de componentes específicos.
Patrón de diseño.	
Patron de servicios.	La clase Signup depende de la interfaz AuthService para realizar operaciones relacionadas con la autenticación, como hashear la contraseña del nuevo usuario.

b. Archivo “UseCase.ts”

El objetivo de este archivo es definir una interfaz denominada "UseCase". Esta interfaz actúa como un modelo para los casos de uso dentro de la estructura del proyecto, estableciendo las pautas básicas que deben seguir las implementaciones de casos de uso en el ámbito específico del proyecto.

Pilares de la Programación orientada a objetos.	
Encapsulamiento.	Los detalles de implementación de los casos de uso se encapsulan en clases concretas que implementan la interfaz "UseCase".
Abstracción.	La interfaz "UseCase" define una abstracción genérica para representar casos de uso en el dominio.
Polimorfismo	La interfaz "UseCase" permite que las clases concretas implementen el método "execute" de diferentes maneras, lo que permite la sustitución de casos de uso en tiempo de ejecución.

Clean architecture	
Se encuentra en la capa de dominio. Define una abstracción genérica que se puede implementar en la capa de aplicación para ejecutar casos de uso específicos.	
Principios SOLID	
Single Responsibility principle	La interfaz "UseCase" tiene la responsabilidad única de definir la firma del método "execute" para la ejecución de casos de uso.
Dependency Inversion Principle	El código cumple con el principio de DIP al utilizar la interfaz "UseCase" como una abstracción que puede ser implementada por casos de uso concretos. Esto permite desacoplar el código que ejecuta casos de uso de los detalles específicos de implementación.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

7. Carpeta “Interfaces”.

El propósito de la carpeta "interfaces" en este proyecto es definir interfaces y contratos que establecen la estructura y el comportamiento esperado de los diversos componentes que conforman el sistema.

a. Archivo “custom-error.ts”

Este archivo define dos interfaces (IJsonObject, ICustomError) que son exportadas. Estas interfaces proporcionan una estructura para representar errores con propiedades adicionales, como un código de error, variables asociadas y un código HTTP correspondiente.

Pilares de la Programación orientada a objetos.	
Abstracción.	Define la forma que deve de tener un objeto, especificando los nomnbre y tipos de las propiedades que debe tener, pero estas no son implementadas en este codigo.
Herencia	La interfaz ICustomError se extuende de la interfaz Erro.
Clean architecture	
Como este código es únicamente la declaración de interfaces no se puede identificar como una capa específica del modelo de clean architecture, sin embargo, las interfaces son utilizadas en varias capas como Entities, Use case o e Interface adapter.	
Principios SOLID	
Single Responsibility principle	El archivo se centra en definir interfaces para errores personalizados, lo que representa una única responsabilidad relacionada con la estructura y representación de errores.
Open/Close	El archivo no muestra una extensión o modificación directa, sino que define interfaces que pueden ser implementadas por diferentes tipos de errores personalizados.
Patrón de diseño.	
N/A	No se encontro ninguno

8. Carpeta “Platorm”.

a. Subcarpeta “Config”

Esta carpeta alberga la configuración esencial para el funcionamiento de la aplicación, incluyendo detalles de la base de datos, autenticación, configuración del servidor, entre otros aspectos. Contiene archivos que definen las variables de entorno, opciones de configuración y cualquier otro dato necesario para garantizar la operación correcta de la aplicación.

i. Archivo “types.ts”

En este archivo se define la estructura de la configuración de la aplicación. Esta estructura puede ser utilizada para garantizar que la configuración de la aplicación tenga el formato correcto y contenga todas las propiedades requeridas.

Pilares de la Programación orientada a objetos.	
Abstracción.	El tipo Config proporciona una abstracción de la configuración de la aplicación. Define la 'forma' que debe tener un objeto de configuración, especificando los nombres y tipos de las propiedades que debe tener,

	pero no cómo se implementan esas propiedades.
Clean architecture	
Este archivo maneja las configuraciones de la aplicación, general mente esto se encuentra en la capa de infraestructura.	
Principios SOLID	
Interface Segregation	El código actúa de manera similar a interfaces a manera de contrato en donde cualquier objeto que se ajuste a este tipo no estará cargado con propiedades innecesarias que no necesita.
Patrón de diseño.	
N/A	No se encontró aplicaciones.

ii. Archivo “index.ts”

Este archivo actúa como la base para la configuración del sistema. Crea un objeto de configuración que cumple con la estructura definida por el tipo "Config" y lo exporta como su valor predeterminado. Utiliza variables de entorno para establecer las diferentes propiedades del objeto de configuración, incluyendo el entorno, los puertos, los hosts, los detalles de la caché, la base de datos, el proyecto y la autenticación.

b. Subcarpeta “lib”

Esta carpeta contiene código y funcionalidades que pueden ser reutilizados en toda la aplicación. Alberga clases, funciones y utilidades que ofrecen servicios comunes, tales como la gestión de errores, operaciones con la base de datos, funciones auxiliares, entre otros. Esta carpeta representa un lugar idóneo para aplicar principios SOLID, como el Principio de Responsabilidad Única (SRP) y el Principio de Inversión de Dependencias (DIP), con el objetivo de desarrollar componentes más cohesivos y adaptables.

i. Archivo “general-error.ts”

Esta implementación de TypeScript define una clase CustomError que extiende la clase Error integrada de JavaScript. La clase CustomError tiene propiedades adicionales: code, message, vars y httpCode, que se inicializan a través de un objeto params de tipo ICustomErrorParams pasado al constructor. El tipo ICustomErrorParams es una interfaz que especifica la estructura requerida para el objeto params. Esta clase permite la creación de errores personalizados con un código, un mensaje, variables adicionales y un código de estado HTTP.

Pilares de la Programación orientada a objetos.	
Polimorfismo	Aunque no es explícito, el hecho de que error sea de tipo CustomError (que implementa la interfaz ICustomError) sugiere que podría haber varias clases de errores personalizados que se manejan de manera uniforme por estas funciones. Un ejemplo de polimorfismo, donde el mismo código puede manejar diferentes errores.
Encapsulamiento.	Las funciones errorHandler y errorHandlerMiddleWare encapsulan la lógica de manejo de errores, ocultando los detalles de implementación y exponiendo sólo una interfaz para ser utilizada por otras partes del código. Esto permite un alto nivel de modularidad y separación de responsabilidades.
Clean architecture	
Se está aplicando la capa de manejo de errores (Error Handling) . El archivo contiene dos funciones principales: errorHandler y errorHandlerMiddleware, que se encargan de manejar los errores que ocurren en la aplicación y enviar una respuesta apropiada al cliente.	
Principios SOLID	
Interface Segregation	El código depende de abstracciones (interfaces y tipos) en lugar de implementaciones concretas, lo que promueve un bajo acoplamiento y facilita las pruebas unitarias y el mantenimiento.
Dependency Inversion Principle	Cada función tiene una responsabilidad única: errorHandler maneja los errores

	y envía la respuesta, mientras que errorHandlerMiddleware es un middleware de Express utilizado para encadenar el manejo de errores en la tubería de middleware.
Patrón de diseño.	
Middleware	errorHandler y errorHandlerMiddleWare son funciones de middleware que manejan los errores que ocurren durante el procesamiento de una solicitud HTTP. Estas funciones se pueden insertar en la cadena de middleware de una aplicación Express.js para manejar los errores de manera centralizada.

ii. Archivo "validate-token.ts"

Este código tiene como objetivo establecer un middleware que verifica un token de acceso en una solicitud HTTP. Primero, revisa si la solicitud contiene un token de acceso. Si el token está presente, procede a descodificar la información del token y la agrega al objeto de solicitud (req.dataTokenUser) para ser utilizada posteriormente en la aplicación. Sin embargo, si el token no está presente o es inválido, se genera un error personalizado.

Pilares de la Programación orientada a objetos.	
Polimorfismo	Aunque no es explícito en este fragmento de código, el hecho de que error sea manejado por la función next sugiere que podría haber varias clases de errores que se manejan de manera uniforme por esta función. Esto sería un ejemplo de polimorfismo, donde diferentes tipos de errores pueden ser manejados por el mismo código.
Encapsulamiento.	La función validateToken encapsula la lógica de validación del token, ocultando los detalles de implementación y exponiendo sólo una interfaz para ser

	utilizada por otras partes del código. Esto permite un alto nivel de modularidad y separación de responsabilidades.
Clean architecture	
Podemos identificar este archivo en la capa de adaptadores, ya que esta capa se encarga de comunicarse con las capas externas, como la red o la base de datos. En este caso, el middleware validateToken se encarga de validar el token de acceso en las solicitudes HTTP, que es una forma de comunicación con la capa externa de la red.	
Principios SOLID	
Interface Segregation	Aunque este código no utiliza interfaces en el sentido tradicional de POO, el tipo ICustomErrorParams actúa de manera similar, definiendo un "contrato" para la estructura de un objeto de parámetros de error. Este "contrato" es pequeño y específico, lo que significa que cualquier objeto que se ajuste a este tipo no estará cargado con propiedades innecesarias que no necesita.
Single Responsibility principle	La clase CustomError tiene una única responsabilidad, que es representar un error personalizado en la aplicación. No está mezclada con otras responsabilidades.
Patrón de diseño.	
Chain of Responsibility	validateToken es un manejador en la cadena de middleware de Express.js. Si puede manejar la solicitud (es decir, si puede validar el token), llama a next() para pasar la solicitud al siguiente manejador en la cadena. Si no puede manejar la solicitud (si el token no es válido), lanza un error, que puede manejarlo otro middleware de manejo de errores en la cadena.

c. Subcarpéta “server”

Esta subcarpeta contiene el código responsable de configurar e iniciar el servidor Express. Aquí se encuentran los archivos que definen las rutas, los controladores y los gestores de solicitudes del servidor. Además, en esta carpeta se ubican los archivos que establecen la configuración y el arranque del servidor, como el puerto, el host, la configuración de seguridad, entre otros. Esta carpeta es fundamental, ya que establece la infraestructura del servidor y conecta los diferentes componentes de la aplicación.

i. Archivo “express.ts”

Este archivo tiene el propósito de inicializar y configurar un servidor Express para la aplicación. Incluye una función llamada "startExpressServer" que se encarga de establecer varias características y middleware de Express, iniciar el servidor en un puerto y host específicos, y realizar la conexión con la base de datos y el servidor de Redis en caso de que estén disponibles.

Pilares de la Programación orientada a objetos.	
Polimorfismo	El polimorfismo se puede ver en cómo se pueden pasar diferentes tipos de manejadores de solicitudes a la función startExpressServer. Esto es posible gracias a la unión de tipos RequestHandler | RequestHandler[], que permite que handlers sea un solo manejador de solicitudes o un array de ellos.
Encapsulamiento.	El encapsulamiento se puede ver en cómo se agrupan los métodos y propiedades relacionados dentro de las clases y módulos. Por ejemplo, la función startExpressServer encapsula la lógica para iniciar un servidor Express.
Abstracción.	La abstracción se puede ver en cómo se utilizan interfaces y tipos para definir la estructura de los datos. Por ejemplo,

	IStartOptions es una interfaz que define la estructura de las opciones de inicio del servidor.
Clean architecture	
Este archivo se encuentra en la capa de infraestructura, pues se está configurando un servidor Express, que es una implementación específica de cómo se manejan las solicitudes HTTP. Esto incluye la configuración de middleware como Helmet, CORS y Compression, y la configuración de manejadores de solicitudes.	
Principios SOLID	
Interface Segregation	Las interfaces CustomRequest e IStartOptions son ejemplos de este principio. En lugar de tener una gran interfaz que haga muchas cosas, estas interfaces son pequeñas y específicas para sus propósitos.
Single Responsibility principle	La función startExpressServer tiene una única responsabilidad: configurar y devolver una aplicación Express.
Open/Close	La función startExpressServer es abierta para la extensión (puedes pasar diferentes manejadores de solicitudes y opciones de inicio) pero cerrada para la modificación (no necesitas cambiar la función en sí para cambiar su comportamiento).
Liskov sustitution	Este principio se aplica en la forma en que se manejan los manejadores de solicitudes. La función startExpressServer puede aceptar un RequestHandler o un array de RequestHandler[]. Cualquier objeto que cumpla con la interfaz RequestHandler puede ser utilizado aquí, lo que significa que cualquier subtipo puede ser sustituido por su supertipo.
Patrón de diseño.	
Strategy	No se muestra de manera directa este patron pero podemos observar una

	similitud en la forma en la que se manejan los manejadores de solicitudes. En el patrón de Estrategia, se define una familia de algoritmos y se hace que sean intercambiables. Aquí, RequestHandler | RequestHandler[] permite que se puedan usar diferentes manejadores de solicitudes, que podrían considerarse como diferentes "estrategias" para manejar las solicitudes HTTP.

ii. Archivo “types.ts”

Este archivo tiene como objetivo definir los tipos de datos personalizados que se utilizan en el entorno del servidor de la aplicación. Define la interfaz "CustomRequest" que extiende la interfaz "Request" de Express y agrega propiedades adicionales para manejar datos personalizados relacionados con la solicitud HTTP. Además, define la interfaz "IStartOptions" que describe las opciones de configuración necesarias para iniciar el servidor Express.

9. Carpeta “Rutes”.

La carpeta "routes" se utiliza para definir y organizar las rutas de la aplicación. Su principal propósito es proporcionar una estructura clara y modular para gestionar las diversas solicitudes HTTP que recibe el servidor.

a. Archivo “Index.ts”

Este archivo define las rutas principales de la versión 1 (v1) de la API. Se utiliza para agrupar y establecer las rutas relacionadas con los recursos de usuario.

b. Archivo “users.ts”

Este script configura las rutas relacionadas con el recurso de usuario en la primera versión (v1) de la interfaz de programación de aplicaciones (API). Establece las rutas "/signin" y "/signup" con el objetivo de gestionar las solicitudes entrantes relacionadas con el inicio de sesión y el registro de nuevos usuarios, respectivamente.

Pilares de la Programación orientada a objetos.	
Encapsulamiento	UserSignupController y UserSigninController probablemente encapsulan la lógica específica para el registro y el inicio de sesión de los usuarios, respectivamente. Estas clases exponen un método execute para interactuar con ellas, ocultando los detalles internos.
Polimorfismo	UserSignupController y UserSigninController implementen una interfaz común que incluye el método execute. Esto permitiría que las clases se utilicen de manera intercambiable en este contexto, lo cual es un ejemplo de polimorfismo.
Clean architecture	
Este archivo se encuentra en la capa de infraestructura, pues se está utilizando el módulo express para definir las rutas HTTP para las operaciones de inicio de sesión y registro de usuarios. Las instancias de UserSignupController y UserSigninController se obtienen del contenedor de inversify, y luego se utilizan para manejar las solicitudes HTTP correspondientes.	
Principios SOLID	
Principio de inversión de dependencias	Las dependencias hacia UserSignupController y UserSigninController no se inyectan directamente. En su lugar, se obtienen del contenedor de inversify. Esto significa que el módulo no depende de implementaciones concretas, sino de abstracciones, lo cual es una aplicación del DIP.
Principio de responsabilidad única	Cada controlador (UserSignupController y UserSigninController) parece tener una única responsabilidad: manejar el registro y el inicio de sesión de los usuarios, respectivamente. Esto es una aplicación del SRP.
Principio de sustitución de Liskov	Aunque no se muestra directamente en este fragmento de código, es probable que UserSignupController y UserSigninController sean sustituibles si implementan la misma interfaz o clase base (que incluiría el método execute). Esto sería una aplicación del LSP.
Patrón de diseño.	
Singleton	Se utiliza para garantizar que una clase tenga solo una instancia y proporcionar un punto de acceso global a ella. En este caso, el contenedor de inversify (container) es un ejemplo de un Singleton. Se utiliza para obtener instancias de UserSignupController y UserSigninController, y se supone que este contenedor es el mismo en toda la aplicación.

10. Carpeta “Util”.

La carpeta "util" contiene archivos que definen las rutas de la API, incluyendo las rutas para el registro y el inicio de sesión de usuarios. Estos archivos importan el enrutador de Express, crean instancias de controladores utilizando un contenedor de inyección de dependencias, y configuran las rutas correspondientes asociadas a las operaciones de registro e inicio de sesión. Cada ruta llama al método "execute" del controlador respectivo para manejar la solicitud entrante.

a. Archivo “primises.ts”

la funcionalidad del archivo "promises.ts":

1. La función sequentialPromises toma un array de funciones que devuelven promesas y un acumulador, y ejecuta todas las promesas secuencialmente, almacenando los resultados en el acumulador.
2. La función memoizePromise toma una función que devuelve una promesa y devuelve una versión memoizada de esa función, que almacena los resultados de las llamadas anteriores en un objeto de caché y devuelve el resultado almacenado cuando se llama con los mismos argumentos.
3. La interfaz executeArrayPromisesOptions define las opciones para una función no mostrada en el fragmento de código proporcionado, incluyendo un mensaje de descripción de error y un booleano que indica si se debe aplicar un aplanamiento.

Estas funciones y la interfaz parecen ser herramientas útiles para manejar promesas de manera secuencial y con caché, así como para configurar opciones para ejecutar arrays de promesas.

Pilares de la Programación orientada a objetos.	
Encapsulacion	La función memoizePromise y su variable cache están agrupadas en una sola unidad. Esto es una forma de encapsulamiento, ya que se agrupan datos relacionados y funciones que operan en esos datos.
Abstraccion	Las funciones sequentialPromises y memoizePromise ocultan los detalles de cómo se manejan las promesas y cómo se almacenan los resultados, respectivamente. Esto es una forma de abstracción, ya que se ocultan los detalles de implementación y se muestra solo la funcionalidad al usuario.

Clean architecture	
No se encontro relacion con ninguna capa.	
Principios SOLID	
Single Responsibility principle	Este archivo se encarga de manejar operaciones relacionadas con promesas, como la ejecución secuencial de promesas (sequentialPromises) y la memorización de promesas (memoizePromise). Ademas cada función en este archivo tiene una única responsabilidad. Por ejemplo, sequentialPromises se encarga de ejecutar promesas de forma secuencial y memoizePromise se encarga de memorizar el resultado de una promesa para evitar ejecuciones innecesarias.
Patrón de diseño.	
Memorizacion	La función memoizePromise guarda los resultados de las promesas en un objeto cache y los reutiliza si se llama a la función con los mismos argumentos. Esto puede ser útil para evitar llamadas a la red innecesarias o cálculos costosos.

*Nota: relacionaba algunos patrones que encontraba con otros vistos en la clase, pero tras consultarlo pude observar que la mayoría de los utilizados eran relacionados o derivados de estos. *

Lógica del proyecto:

La lógica de negocio y la lógica de aplicación son dos conceptos clave en el desarrollo de software. La lógica de negocio se refiere al conjunto de reglas y procedimientos que definen cómo se llevan a cabo las operaciones y procesos esenciales de una empresa o dominio específico. Esta lógica está basada en el conocimiento del área de negocio y dicta cómo se deben manejar los datos y las transacciones. La lógica de negocio reside en la carpeta "src" del proyecto.

Por otro lado, la lógica de aplicación se encarga de la funcionalidad técnica que permite que la aplicación funcione correctamente. Esto incluye la gestión de la interfaz de usuario, el procesamiento de datos y la comunicación con otras aplicaciones o servicios. La lógica de aplicación se encuentra en el resto de carpetas y archivos del proyecto.

Algunos ejemplos de elementos que pertenecen a la lógica de aplicación son:

- .vscode: Contiene configuraciones específicas del editor de código VS Code, lo cual ayuda en el desarrollo pero no afecta la lógica de negocio.

- -db: Puede contener scripts de base de datos o migraciones, lo cual configura cómo la aplicación interactúa con la base de datos.
- .environment.example y -enviroment.d.ts: Archivos relacionados con la configuración del entorno de la aplicación.
- docker-compose.yml: Archivo de configuración para Docker que ayuda a definir y ejecutar aplicaciones multi-contenedor.
- package.json: Define las dependencias y scripts de la aplicación.
- Archivos de configuración de TypeScript (tsconfig, tslint): Definen cómo se compila el código TypeScript y ayudan a mantener la calidad y consistencia del código.
- webpack.config.js: Archivo de configuración para Webpack, una herramienta de construcción.

conclusión:

Este proyecto, presentó algunos desafíos, pero logré superarlos satisfactoriamente.
Para organizar el código y dividir las responsabilidades, adopté la arquitectura Clean Architecture. Esta aproximación fomenta una mayor modularidad, flexibilidad y facilidad de mantenimiento del sistema.
La implementación de la Clean Architecture se basó en los principios SOLID. Esto me permitió desarrollar componentes independientes, con alta cohesión interna y bajo acoplamiento entre ellos. Como resultado, mejoré la comprensibilidad del código, la capacidad de realizar pruebas y la facilidad para evolucionar el software a lo largo del tiempo.
En consecuencia, la aplicación de esta arquitectura y sus principios asociados elevó de manera significativa la calidad general del proyecto.
Si bien hubo dificultades iniciales, logré superarlas gracias a una cuidadosa planificación y la adopción de mejores prácticas de diseño de software. El resultado final es un sistema más modular, testeable y sostenible a lo largo del tiempo.
Para este proyecto, se ha adoptado la Clean Ar.