



UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

Facultad de Informática



Proyecto Final

Patrones de diseño

Carlos Noguez Juarez

315398

Sabado 18 de Mayo del 2024

Pequeña presentación.

En el presente archivo se analizará el uso e implementación de los patrones de diseño abordados en clase, las diferentes capas del modelo de clean architecture, la utilización de los principios SOLID así como la aparición de los 4 pilares de la programación orientada a objetos. Para lo cual se ha establecido un formato de tablas según cada archivo, pues se busca facilitar la revisión de este documento.

1. Carpeta Common.

En esta carpeta hay servicios como servicios para la gestión de caché y procesos de autenticación, además de ofrecer herramientas prácticas y clases estándar empleadas en todo el proyecto.

- a. Archivo “util.ts”.

Este documento incluye una serie de herramientas útiles para ser empleadas a lo largo 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 atributo evt_id de tipo number.

```
1  export interface Event {
2    evt_id: number;
3  }
```

- c. Archivo “index.ts”

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

```

1  export { CacheService } from './cache/CacheService';
2  export { AuthService } from './auth/auth-service';
3  export * from './util';
4  export * from './types';
5  export * from './auth/AuthService';

```

- 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, verificacion de un token JWT, verificacion de la cioncidencia de contraseña asi como de la generacion 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

Esta carpeta es responsable de administrar la configuración y el arranque de diferentes componentes de la aplicación. Se utiliza específicamente para configurar y operar la base de datos, implementar el inversor de control (IoC) para la inyección dentro de la aplicación, y establecer los símbolos que actúan como identificadores exclusivos para las dependencias dentro del contenedor de 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 lo que no se puede acceder a ellas desde fuera
Abstracción.	Se utiliza una abstracción (interfaz) con la cual se define el tipo de retorno de la función "getRedisClient"
Clean architecture	
N/A	No se encontró aplicación
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 implementación concreta en la función "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 encontró aplicación
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. Subcarpeta "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 aplicación
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. Subcarpeta "symbols"

i. Archivo "index.ts"

Este documento crea una serie de identificadores únicos, llamados symbols, para las dependencias en el contenedor IoC. Estos identificadores enlazan las interfaces o clases generales con sus respectivas versiones específicas dentro del contenedor.

```

1  export const SYMBOLS = {
2    CacheService: Symbol("CacheService"),
3    AuthService: Symbol("AuthService"),
4    ScoreIntentUseCase: Symbol("ScoreIntentUseCase"),
5    ScoreTopUseCase: Symbol("ScoreTopUseCase"),
6    InitGameUseCase: Symbol("InitGameUseCase"),
7    ScoreStatsUseCase: Symbol("ScoreStatsUseCase"),
8    SignupUseCase: Symbol("SignupUseCase"),
9    SigninUseCase: Symbol("SigninUseCase"),
10   CatalogueRepository: Symbol("CatalogueRepository"),
11   UserRepository: Symbol("UserRepository"),
12   ScoreRepository: Symbol("ScoreRepository"),
13 };
14

```

3. Carpeta "constants"

Dicha carpeta incluye archivos que establecen constantes o valores fijos usados en el proyecto. Estas constantes son valores que se aplican en diferentes secciones del código y permanecen iguales mientras el programa está en funcionamiento. No es necesario analizarlos de manera directa.

```

1  /**
2   * Redis uses ttl in seconds
3   * Reference: https://redis.io/commands/ttl
4  */
5  export const ONE_MINUTE = 60;
6  export const ONE_HOUR = ONE_MINUTE * 60;
7  export const ONE_DAY = ONE_HOUR * 24;
8  export const ONE_WEEK = ONE_DAY * 7;
9  export const ONE_MONTH = ONE_DAY * 30;
10
11
1  export const game = {
2    status: {
3      victory: "VICTORY",
4      gameOver: "LOOSER",
5    },
6    messages: {
7      wordInvalid: "Word Invalid",
8      initGameRequired: "Init Game Needed",
9      gameOver: "Game Over",
10    },
11  };
1
2
1  export * as redis from './redis';
2  export * as defaults from './default'
3  export * from './code-errors'
4  export * from './game';
5
1  const CODE_ERRORS = {
2    UNAUTHORIZED: 'UNAUTHORIZED',
3  };
4
5  export { CODE_ERRORS };
6

```

4. Carpeta “controllers”

En esta carpeta se busca guardar los controladores de la aplicación. Estos controladores son elementos encargados de manejar las peticiones HTTP que llegan, procesarlas y devolver una respuesta correcta 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.	
Inyección de dependencias.	La etiqueta <code>@injectable()</code> indica que la clase <code>UserSigninController</code> puede recibir inyecciones de dependencias. Con la etiqueta <code>@inject(SYMBOLS.SigninUseCase)</code> en el constructor, se introduce una versión de <code>SigninUseCase</code> al controlador, que luego se asigna a la propiedad <code>signin</code> . Así, <code>UserSigninController</code> obtiene <code>SigninUseCase</code> 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 documento describe una clase base denominada `BaseController` en el directorio de controladores. Esta clase base ofrece métodos estándar y herramientas para gestionar peticiones 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 <code>execute</code> .
Herencia.	Como clase abstracta, espera ser extendida por otros controladores, permitiendo el uso compartido de funcionalidades.
Polimorfismo.	Define un método abstracto <code>execute</code> 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.	
Inyección de dependencias.	De manera similar al archivo “ign-in.ts”, la clase <code>UserSignupController</code> es anotada con <code>@injectable()</code> , y en su constructor se inyecta una instancia de <code>SignupUseCase</code> utilizando la anotación <code>@inject(SYMBOLS.SignupUseCase)</code> , 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 encontró ninguna posible aplicación

- 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 función de la carpeta “data-access” es centralizar y controlar todas las acciones relacionadas con el acceso a datos en una aplicación. Actúa como un nivel de conexión entre la sección de lógica de negocio y las fuentes de datos, tales como bases de datos, servicios en línea o archivos del sistema.

- a. Subcarpeta “cache”
 - i. Archivo “redis-impl.ts”

El archivo “redis-impl.ts” contiene una clase llamada RedisImpl. Esta clase es una versión específica de un servicio de caché que usa Redis. Tiene métodos para guardar y sacar datos

de Redis, como por ejemplo, conseguir el valor de una llave, poner un valor a una llave, borrar una llave, entre otros. Además, usa una técnica llamada memoización para recordar los resultados de algunas funciones, lo que ayuda a que todo funcione más rápido.

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.	
Inyección de dependencias	Se usa @injectable() y Inversify para conectar las partes del código de forma clara.
Memento (memorización)	Se almacena los resultados de una función 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” tiene una clase llamada UserRepositoryImpl. Esta clase es una versión específica del repositorio de usuarios. Ofrece métodos para manejar datos de usuarios, como añadir un nuevo usuario o buscar un usuario por su nombre. Para trabajar con la base de datos y hacer consultas SQL, usa 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 la utilización de ningún patrón

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 “domain”

El propósito del directorio “domain” es ser el corazón del proyecto, manteniendo la lógica principal y las normas que rigen la aplicación. En este lugar se establecen los conceptos y entidades esenciales para el área del problema que la aplicación soluciona. Además, su función es mantener la lógica empresarial aparte de la parte técnica de la implementación

a. Subcarpeta “user”

i. Archivo “Sing-in.ts”

La finalidad de este archivo es desarrollar la funcionalidad de inicio de sesión para un usuario dentro del proyecto. Incluye la lógica requerida para verificar las credenciales del usuario, crear un token de autenticación y entregarlo 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 está 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 utilización de ningún patrón

ii. Archivo "sign-up.ts"

La meta de este código es llevar a cabo la función de registro de usuario dentro del proyecto. Incluye todo lo necesario para añadir un usuario nuevo 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 propósito de este archivo es establecer una interfaz llamada "UseCase". Esta interfaz actúa como un modelo para los casos de uso dentro de la estructura del proyecto, estableciendo las reglas básicas que deben seguir las implementaciones de los casos de uso en el área específica 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 encontro ninguno

7. Carpeta “interfaces”

La carpeta "interfaces" en este proyecto tiene como objetivo definir interfaces y contratos que establecen la estructura y comportamiento esperado de diferentes componentes del sistema.

a. Archivo “custom-error.ts”

Este archivo define 2 interfaces (IJsonObject, ICustomError) las cuales se exportan y se proporciona una estructura para representar errores con propiedades adicionales, como un código de error, variables y un código HTTP asociado.

Pilares de la Programación orientada a objetos.	
Abstracción.	Define la forma que deve de tener un objeto, especificando los nombre 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 contiene la configuración necesaria para el funcionamiento de la aplicación, incluyendo detalles de la base de datos, autenticación y configuración del servidor, entre otros. Alberga archivos que definen variables de entorno, opciones de configuración y cualquier otro dato requerido para el correcto funcionamiento de la aplicación.

i. Archivo “types.ts”

Se define la estructura de la configuración de la aplicación, que puede ser utilizada para asegurar que la configuración de la aplicación tenga la forma correcta y contenga todas las propiedades necesarias.

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, generalmente 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 ninguno

ii. Archivo "index.ts"

Este archivo sirve como la base para la configuración del sistema. Crea un objeto de configuración que sigue la estructura del tipo "Config" y lo exporta como su valor por defecto. Utiliza variables de entorno para establecer las distintas 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"

Contiene código y funcionalidades que se pueden reutilizar en toda la aplicación. Contiene clases, funciones y utilidades que ofrecen servicios comunes, como la gestión de errores, operaciones con la base de datos, funciones auxiliares, entre otros. Esta carpeta es un lugar ideal para aplicar principios SOLID, como el Principio de Responsabilidad Única (SRP) y el Principio de Inversión de Dependencias (DIP), con el fin de desarrollar componentes más unificados y adaptables.

i. Archivo "general-error.ts"

Este archivo TypeScript define una clase CustomError que extiende la clase Error incorporada 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.

Herencia	La clase CustomError extiende la clase Error incorporada de JavaScript. Esto significa que CustomError hereda todas las propiedades y métodos de Error, pero también añade y sobrescribe algunas propiedades y métodos propios.
Encapsulación	La clase CustomError encapsula las propiedades code, message, vars, httpCode y params, y las hace accesibles solo a través de métodos definidos en la clase (en este caso, el constructor). Esto asegura que los datos estén bien protegidos y solo se puedan modificar de maneras controladas.

Clean architecture

Este código se encuentra en la capa de entidades, ya que la clase customError podría considerarse como una entidad del dominio, pues representa un concepto que es muy relevante para la lógica del negocio de la aplicación.

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.	
Factory Method	El constructor de la clase CustomError actúa como un método de fábrica. En lugar de requerir que los clientes instancien un objeto Error y luego lo modifiquen para satisfacer sus necesidades, los clientes pueden simplemente crear un CustomError directamente con todas las propiedades que necesitan (code, message, vars, httpCode).

c. Subcarpeta “middlewares”

Contiene funciones intermedias de Express que se ejecutan durante el procesamiento de solicitudes HTTP antes de enviar una respuesta. Estas funciones intermedias tienen varios propósitos en la aplicación, como el manejo de errores, la autenticación de usuarios, la validación de datos entrantes y la gestión de sesiones, entre otros. Los middlewares permiten encapsular y reutilizar lógica común a través de diferentes rutas y controladores, lo que mejora la estructura y facilita el mantenimiento del código.

A lo largo de esta carpeta podemos observar la aplicación del patrón middleware que es una variante del patrón Chain of responsibility

i. Archivo “error-handler.ts”

Este archivo pretende establecer un middleware que gestiona los errores en la aplicación. Tiene funciones diseñadas para tratar los errores y producir respuestas JSON apropiadas si se presenta un error al procesar una petición 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.
Encapsulacion	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.
-----------------------	---

Single Responsibility 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 pretende establecer un middleware que comprueba un token de acceso en una petición HTTP. Revisa si la petición contiene un token de acceso y, en caso de ser válido, decodifica la información del token y la añade al objeto de petición (req.dataTokenUser) para ser utilizada más adelante en la aplicación. 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.
Encapsulacion	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.
--	---

d. Subcarpeta "server"

Esta subcarpeta alberga el código encargado de configurar e iniciar el servidor Express. En esta carpeta se encuentran los archivos que definen las rutas, los controladores y los manejadores de solicitudes del servidor. Además, aquí 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 como propósito inicializar y configurar un servidor Express para la aplicación. Incluye una función llamada "startExpressServer" que se encarga de establecer diversas 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.
Encapsulacion	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.
Abstraccion	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 patrón 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 emplea para definir y organizar las rutas de la aplicación. Su principal propósito es brindar 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”

EsteScript configura las rutas vinculadas al recurso de usuario en la primera versión (v1) de la interfaz de programación de aplicaciones (API). Establece las rutas "/signin" y "/signup" con el propósito de gestionar las peticiones 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 implementan 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 e 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, donde cada ruta llama al método "execute" del controlador respectivo para manejar la solicitud entrante.

a. Archivo "promises.ts"

El archivo "promises.ts" contiene tres funciones relacionadas con promesas en JavaScript. La función sequentialPromises toma un array de funciones que devuelven promesas y un acumulador, ejecuta todas las promesas secuencialmente, almacenando los resultados en el acumulador. 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. 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.

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:

Lógica de negocio: La lógica de negocio reside en la carpeta src, esto pues la logica de negocios 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. Estas reglas están basadas en el conocimiento del área de negocio y dictan cómo se deben manejar los datos y las transacciones.

Lógica de Aplicación: Aquí podemos encontrar al resto de carpetas y archivos, pues esta lógica se ocupa 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.

- ❖ .vscode: Contiene configuraciones específicas del editor de código VS Code, lo cual es parte de la lógica de aplicación ya que ayuda en el desarrollo, pero no afecta la lógica de negocio.
- ❖ -db: Puede contener scripts de base de datos o migraciones, lo cual es una extensión de la lógica de aplicación ya que configura cómo la aplicación interactúa con la base de datos.
- ❖ .environment.example y -enviroment.d.ts: Estos archivos están relacionados con la configuración del entorno de la aplicación, lo cual es parte de la lógica de aplicación.
- ❖ docker-compose.yml: Es un archivo de configuración para Docker que ayuda a definir y ejecutar aplicaciones multi-contenedor, lo cual es parte de la lógica de aplicación.
- ❖ package.json: Define las dependencias y scripts de la aplicación, lo cual es parte de la lógica de aplicación.
- ❖ tsconfig.app.json, tsconfig.json, tsconfig.spec.json: Estos archivos de configuración de TypeScript son parte de la lógica de aplicación ya que definen cómo se compila el código TypeScript.

- ❖ tslint.json: Contiene las reglas para el linter de TypeScript, lo cual es parte de la lógica de aplicación ya que ayuda a mantener la calidad y consistencia del código.
- ❖ webpack.config.js: Es un archivo de configuración para Webpack, una herramienta de construcción que forma parte de la lógica de aplicación.

Conclusiones:

Estuvo difícil. 😊

Para este proyecto, se ha adoptado la Clean Architecture como enfoque para organizar el código y dividir las responsabilidades en capas bien delimitadas. Esta arquitectura fomenta una mayor modularidad, flexibilidad y facilidad de mantenimiento del sistema. Su implementación se ha basado en los principios SOLID, lo que ha posibilitado el desarrollo de componentes independientes, con alta cohesión interna y bajo acoplamiento entre ellos. Esto mejora la comprensibilidad, las pruebas y la capacidad de evolucionar el código, lo que a su vez eleva la calidad general del software.

Fuentes:

- Martin, R. C. (2018). Arquitectura limpia: Guía para especialistas en la estructura y el diseño de software.
- Coder, D. (2023). Introducción a las ‘Clean Architectures’. Medium. <https://medium.com/@diego.coder/introducci%C3%B3n-a-las-clean-architectures-723fe9fe17fa>
- Morales, I. (2015). Patrones de diseño: ¿qué son y para qué sirven?. Platzi. <https://platzi.com/blog/patrones-de-diseno/>
- Refactoring.Guru.(2024) Patrones de diseño. <https://refactoring.guru/es/design-patterns>
- freeCodeCamp. (2022, noviembre 28). Los principios SOLID de programación orientada a objetos explicados en Español sencillo. freeCodeCamp. <https://www.freecodecamp.org/espanol/news/los-principios-solid-explicados-en-espanol/>
- Profile. (2018, noviembre 22). SOLID: los 5 principios que te ayudarán a desarrollar software de calidad. Profile. [SOLID: los 5 principios que te ayudarán a desarrollar software de calidad \(profile.es\)](https://profile.es/SOLID-los-5-principios-que-te-ayudaran-a-desarrollar-software-de-calidad-profile.es)
- Capdevila, A. (s.f.). Qué es la lógica de negocio en programación y cómo distinguirla de la lógica de aplicación y de pantalla. <https://albertcapdevila.net/logica-negocio-programacion/>