UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

Facultad de Informática

Patrones de Diseño

Proyecto Final

Análisis de proyecto “template-nodejs”

Docente: Enrique Aguilar

Presenta: Gael de Jesús Posada Hernández

Ingeniería de Software

Introducción

En este proyecto final, se examinará la aplicación de los patrones de diseño estudiados en clase, las distintas capas del modelo de arquitectura limpia (Clean Architecture), la implementación de los principios SOLID, así como la manifestación de los cuatro pilares de la programación orientada a objetos, todo lo anterior presente en la carpeta de un proyecto de código brindado por el profesor. Se busca brindar una estructura clara y accesible para el análisis de los conceptos mencionados.

Common

La carpeta common almacena código compartido y utilizado en todo el proyecto. Contiene servicios comunes como gestión de caché, autenticación, herramientas útiles y clases estándar. Estos componentes brindan funcionalidades esenciales y facilitan la implementación consistente de características transversales en la aplicación. La carpeta common puede incluir otros elementos según las necesidades y convenciones del proyecto.

auth/AuthService.ts

En este archivo se exporta la interfaz que se utilizara para realizar la implementación

auth/auth-service.ts

Este archivo contiene la implementación de la clase "AuthService", la cual se encarga de automatizar la autenticación de usuarios y la generación de tokens. Esta clase incluye los siguientes métodos: Generación de un token JWT (JSON Web Token), Verificación de un token JWT, Verificación de la coincidencia de una contraseña, Generación de un hash de contraseña.

POO

La clase "AuthService" encapsula los métodos relacionados con la autenticación, generación de tokens, verificación de tokens y manejo de contraseñas. Los métodos están encapsulados y solo son accesibles a través de la instancia de la clase, lo que proporciona abstracción de estos servicios de autenticación. Esta clase no hereda de ninguna otra, pero implementa la interfaz "IAuthService". Al implementar dicha interfaz, "AuthService" puede ser utilizada en diferentes contextos sin afectar la funcionalidad de la aplicación, brindando flexibilidad y desacoplamiento.

SOLID

La clase "AuthService" tiene múltiples responsabilidades (generar tokens, verificar tokens, comparar contraseñas, hashear contraseñas), violando el Principio de Responsabilidad Única (SRP). Implementa la interfaz "IAuthService", dependiendo de una abstracción en lugar de una implementación concreta.

cache/CacheService.ts

El objetivo es definir la interfaz para un servicio de almacenamiento en caché. Este servicio se utilizará para almacenar datos de manera temporal con el fin de mejorar el rendimiento de la aplicación.

POO

La gestión de datos en caché se maneja a través de métodos encapsulados, definidos en una interfaz y sus implementaciones correspondientes. La interfaz especifica los métodos requeridos para interactuar con el servicio de caché, sin necesidad de conocer los detalles de la implementación concreta. Esto permite que el cliente pueda interactuar con diferentes implementaciones de servicios de caché, siempre y cuando cumplan con la misma interfaz definida.

CleanArchitecture

La interfaz especifica los métodos necesarios para interactuar con el servicio de caché, lo que permite encapsular la lógica de gestión de datos en caché y abstraer los detalles de implementación. Esta abstracción brinda flexibilidad, ya que el cliente puede interactuar con diferentes implementaciones de servicios de caché, siempre y cuando cumplan con la misma interfaz definida.

SOLID

La interfaz está diseñada para cumplir con el Principio de Responsabilidad Única, ya que únicamente define los métodos relacionados con la gestión de caché. Al establecer la estructura necesaria para interactuar con un servicio de caché, esta interfaz abstrae los detalles internos de su implementación, lo que permite a los clientes utilizar el servicio sin conocer cómo se implementa internamente. Además, al especificar solamente los métodos esenciales para interactuar con un servicio de caché, la interfaz garantiza que sus implementaciones sólo implementen los métodos estrictamente necesarios, evitando así funcionalidades adicionales no requeridas.

util.ts

Este archivo contiene herramientas útiles para ser empleadas a lo largo del proyecto.

POO

El encapsulamiento garantiza el aislamiento adecuado de funciones y variables dentro de métodos correspondientes, evitando el acceso directo externo, mientras que la abstracción utiliza interfaces para definir conjuntos de métodos que se implementan en diferentes secciones del código.

SOLID

Cada método está creado de forma que este realiza solo una tarea específica al momento.

Config

La carpeta "config" administra la configuración y arranque de componentes en la aplicación. Configura y opera la base de datos, implementa el IOC inyectando dependencias, y establece símbolos únicos como identificadores de dependencias. Almacena archivos de configuración con variables, bases de datos, seguridad, etc. Éstos centralizan y gestionan la configuración de forma accesible.

db/redis.ts

POO

Las variables se manejan correctamente al declararse como locales, impidiendo el acceso externo. Se implementa abstracción a través de una interfaz que define el tipo devuelto por la función "getRedisClient". Esto encapsula la lógica de obtención del cliente Redis mientras expone una firma estandarizada. La abstracción mediante interfaces oculta la implementación y enfoca sólo en la funcionalidad. En este caso aísla los detalles de conexión con Redis.

SOLID

Las funciones "getRedisClient" y "connectRedisClient" cumplen cada una una única responsabilidad promoviendo el principio SOLID, mientras que la función "connectRedisClient" utiliza la interfaz genérica "Redis.Redis" en lugar de una implementación concreta permitiendo abstraer la lógica de conexión a Redis e inyectar diferentes conectores a través de dicha interfaz, simplificando el entendimiento y mantenimiento al asignar una única acción por función y mejorando la modularidad e independencia de las partes a través del uso de interfaces sobre clases concretas.

db/sql.ts

POO

El archivo "sql.ts" encapsula correctamente las variables y funciones evitando el acceso directo desde otros archivos. Implementa abstracción mediante las interfaces DBSQLArguments y DBReplyDataRow que definen los tipos utilizados por las funciones, permitiendo ocultar la implementación concreta de datos y resultados de consultas SQL aprovechando las ventajas que brindan las interfaces en términos de desacoplamiento, estabilidad y sustituibilidad de módulos.

SOLID

Las funciones "connect" y "sql" cumplen el principio de responsabilidad única, donde cada una se encarga de una tarea específica. Esta última dependiente de abstracciones (DBSQLArguments y DBReplyDataRow) a través del uso de interfaces, siguiendo así el principio de inversión de dependencias en lugar de utilizar implementaciones directas, permitiendo desacoplar los módulos y que sean más flexibles al cambio e independientes entre sí gracias a esta técnica de diseño.

inversify/index.ts

POO

Dentro del contenedor, las dependencias y sus relaciones se encuentran confinadas sin posibilidad de acceso externo directo. Se implementa abstracción mediante el uso de interfaces y clases abstractas que permiten detallar de forma genérica las dependencias y enlaces dentro del mismo siguiendo los principios de inversión de dependencia e inyección de dependencias gracias a la capacidad que otorgan estas técnicas como el contenedor IoC de ocultar la implementación concreta de los servicios favoreciendo el desacoplamiento y la sustituibilidad de los mismos.

SOLID

El archivo de configuración sigue el principio de responsabilidad única centralizando la definición y asociación de dependencias, lo que aísla la configuración de inyección de dichas dependencias del resto del código haciéndolo más desacoplado y fácil de mantener. Asimismo, al emplear el contenedor IoC para inyectar las dependencias en lugar de crearlas directamente en las clases, se siguen los principios de inversión de dependencia e inyección de dependencias, mejorando la flexibilidad, simplificando las pruebas unitarias y facilitando el cambio de implementaciones, al permitir sustituir fácilmente ciertos servicios sin modificar el código existente.

Constants

La carpeta "constants" incluye archivos que establecen valores fijos usados en diferentes secciones del código y aplicaciones. Dichas constantes no requieren ser analizadas directamente y suelen contener definiciones como códigos de error, mensajes, claves, URLs que no cambian durante la ejecución. Centralizar estas constantes facilita su mantenimiento y reutilización en todo el proyecto, evitando la duplicación de valores y favoreciendo los cambios de una única vez en un solo lugar.

Controllers

La carpeta "controllers" almacena los controladores de la aplicación, los cuales se encargan de manejar las solicitudes HTTP entrantes, procesarlas e interactuar con el modelo de datos para devolver una respuesta al cliente. Siguen un patrón MVC u otros similares. Generalmente contienen métodos para acciones como crear, leer, actualizar y eliminar recursos, además de la lógica de negocio asociada. Organizarlos de esta forma mantiene la estructura ordenada y facilita la navegación y entendimiento del código, permitiendo cumplir su rol de gestionar cada petición, interactuar con el modelo y devolver una respuesta.

users/sing-in.ts

POO

El controlador implementa varios principios SOLID. Encapsula los datos mediante propiedades y métodos privados. Utiliza abstracción simplificando la interacción con inicio de sesión. Aprovecha la herencia de BaseController para reutilizar funcionalidades. Y gracias al polimorfismo, diferentes controladores pueden ofrecer su propia implementación del método execute manteniendo una interfaz común, favoreciendo la apertura al cambio y permitiendo la sustitución de la clase concreta por otra sin afectar el código cliente. De esta forma optimiza el diseño siguiendo los principales beneficios que aportan estos patrones.

Cleanarchitecture

El controlador sigue el principio de responsabilidad única al mantener separadas sus distintas tareas. Implementa abstracción obteniendo las clases requeridas a través de inyección de dependencias en lugar de utilizar directamente las implementaciones concretas. Gracias a esta técnica optimiza el diseño al facilitar el desacoplamiento e introducir flexibilidad para cambiar o sustituir servicios sin modificar el propio controlador.

SOLID

Siguiendo el principio de responsabilidad única, el controlador de inicio de sesión se enfoca exclusivamente en gestionar dichas solicitudes.

Asimismo, al basarse en abstracciones para la inyección de dependencias a través de la interfaz SigninUseCase en lugar de una clase concreta, cumple el principio de inversión de dependencias permitiendo inyectar cualquier implementación de este caso de uso sin necesidad de modificar el controlador, lo que mejora su desacoplamiento, sustituibilidad y flexibilidad ante cambios futuros.

user/sing-up.ts

POO

El controlador base implementa varios principios SOLID. Utiliza encapsulamiento a través de métodos protegidos para gestionar las respuestas y ocultar su implementación. Define una interfaz común a través de abstracción mediante la clase abstracta que otros controladores pueden extender, aplicando así polimorfismo al permitir cada uno implementar su propia versión de execute manteniendo una interfaz unificada. Esto facilita la herencia y el uso compartido de funciones, así como introducir flexibilidad a través de la sustituibilidad de las clases derivadas, optimizando el diseño siguiendo los beneficios que aportan estos patrones.

CleanArchitecture

BaseController adhiere a esta arquitectura SOLID al dividir responsabilidades aplicando el principio de responsabilidad única para ofrecer funciones comunes a controladores derivados de forma genérica. Implementa también abstracciones definiéndose como clase abstracta con métodos por implementar, introduciendo flexibilidad para trabajar de forma independiente con cualquier controlador hijo sin conocer su implementación concreta. Al dividir tareas y usar abstracciones en lugar de soluciones específicas, maximiza la cohesión y minimiza el acoplamiento entre sus elementos, siguiendo así los principios básicos de la arquitectura SOLID para favorecer el mantenimiento y evolución a largo plazo.

SOLID

BaseController sigue los principios SOLID, especialmente la responsabilidad única al concentrarse en proveer herramientas genéricas para el manejo estándar de peticiones HTTP, como enviar respuestas o errores de forma uniforme. Asimismo, al estar diseñada para ser extendida de forma abstracta, cumple con la inversión de dependencias al no vincularse a implementaciones concretas de otros componentes, dependiendo de forma genérica de abstracciones que pueden ser implementadas de múltiples formas, maximizando la decoupling y flexibilidad del diseño.

Conclusiones

A nivel general, este código implementa los patrones de diseño más utilizados al trabajar bajo el paradigma de la orientación a objetos:

- Encapsulamiento: Se ocultan detalles de implementación a través de clases y métodos privados/protegidos, definiendo claras interfaces de acceso a los datos. Esto se aprecia en clases como RedisImpl y UserRepositoryImpl.
- Abstracción: Se define abstracción mediante interfaces que establecen contratos genéricos independientes de implementaciones concretas. Esto permite ocultar detalles y trabajar a nivel conceptual, como en RedisImpl y UserRepositoryImpl al usar interfaces CacheService y UserRepository.
- Polimorfismo: Se permite que clases derivadas implementen métodos abstractos de forma diferente, habilitando un tratamiento uniforme a través de interfaces comunes. Esto se logra con RedisImpl al usar métodos de CacheService.
- Inyección de dependencias: Se habilita inyectar dependencias en tiempo de ejecución en lugar de acoplar directamente la creación de objetos. Esto se hace mediante dependencias abstractas y el uso de @injectable junto a Inversify.
- Inversión de dependencias: Se aplica depender de abstracciones en vez de implementaciones concretas, como RedisImpl con CacheService. Esto desacopla el código y permite independencia e intercambiabilidad de módulos.
- Responsabilidad única: Cada clase/función se encarga de una única tarea, aplicando alto grado de cohesión y bajo acoplamiento. Ejemplo claro son funciones de DataAccess.
- Memento: Se memorizan resultados en RedisImpl aplicando patrones de diseño como Memento para mejorar el rendimiento mediante caché y evitar recalcular datos.

Referencias

NetMentor. (s. f.). *Clean Architecture | Explicación y opinión*. https://www.netmentor.es/entrada/clean-architecture. https://www.netmentor.es/entrada/clean-architecture

Baca, J. (2023, 30 octubre). *Clean Architecture: La Guía Definitiva para el Diseño de Software*. BacaSoftware. https://www.bacasoftware.com/clean-architecture-la-guia-definitiva-para-el-diseno-de-software/

Patrones de diseño / Design patterns. (s. f.). https://refactoring.guru/es/design-patterns