UNIVERSIDAD AUTONOMA DE QUERETARO

FACULTAD DE INFORMATICA

PATRONES DE DISEÑO

PROYECTO FINAL TALLER PATRONES DE DISEÑO

ENRIQUE AGUILAR

ELIAS ALCANTARA HERNANDEZ

325859

INGENIERIA DE SOFTWARE

GRUPO 37

Descripción

El proyecto final consiste en el entendimiento y asociamiento de los conceptos vistos en clase (Principios de POO, Principios SOLID y algunos patrones de diseño) en una serie de archivos asignadas por el profesor.

1.-Carpeta .vscode

*(settings.json):

Este fragmento de código esta configurando como un corrector ortográfico para reconocer la palabra "inversify", puedes configurar cualquier palabra para que sea correcta.

2.-Carpeta db/migrations

*(20221212191745_wordle_sqs.sql):

Este fragmento de código es parte de un proceso de actualización de una base de datos. En la sección “migrate:up”, se crea una especie de contador que genera números automáticamente cada vez que se necesita un nuevo valor. En la sección “migrate:down”, se elimina este contador en caso de que se necesite deshacer la actualización.

*(20221212191802_user-table.sql):

Este código crea una nueva caja en una base de datos llamada “user_account”. Dentro de esta caja, puedes almacenar información sobre usuarios, como sus nombres, contraseñas y cuándo crearon o actualizaron sus cuentas. También se asegura de que cada usuario tenga un identificador único y de que la búsqueda de usuarios por nombre sea más rápida. Si necesitas deshacer estos cambios por alguna razón, el código también da las instrucciones necesarias para eliminar esta caja y todo el contenido.

3.-Carpeta src

*(common):

(auth):

-AuthService.ts:

Esta interfaz define métodos para manejar la autenticación de usuarios:

“generateToken(data: any)”: Genera un token de autenticación a partir de los datos proporcionados.

“verifyToken(token: string)”: Verifica si un token de autenticación dado es válido.

“matchPassword(password, hash)”: Compara una contraseña en texto plano con su hash correspondiente.

“hashPassword(plainPassword): string”: Convierte una contraseña en texto plano en un hash de contraseña y devuelve el hash como una cadena de texto.

- auth-service.ts:

Este código implementa una clase “AuthService” que maneja la autenticación de usuarios:

“generateToken(data: any)”: Genera un token de autenticación usando “jsonwebtoken”.

“verifyToken(token: string)”: Verifica la validez de un token usando “jsonwebtoken”.

“matchPassword(password, hash)”: Compara una contraseña con su hash correspondiente usando `bcryptjs`.

“hashPassword(plainPassword): string”: Convierte una contraseña en un hash usando `bcryptjs`.

(cache):

-CacheService.ts:

Esta interfaz define métodos para un servicio de caché:

“getByKey<T>(key: string): Promise<T | null>”: Obtiene un valor de la caché dado su clave.

“setByKey<T>(key: string, value: T, expireTimeInSeconds?: number): Promise<void>”: Establece un valor en la caché con una clave dada, con una opción para especificar el tiempo de expiración.

“generateFunctionKey<T>(functionName: string, args?: T): string”: Genera una clave única basada en el nombre de la función y sus argumentos.

“memoize<T>(method: (...args: unknown[]) => Promise<T>, ttl?: number): (...args: unknown[]) => Promise<T>”: Devuelve una versión de una función que almacena en caché los resultados para mejorar el rendimiento.

“deleteByKey(key: string): Promise<void>”: Elimina un valor de la caché utilizando su clave.

“getKeyTLL(key: string): Promise<number>”: Obtiene el tiempo restante antes de que un valor en la caché expire.

-index.ts:

El código exporta diferentes elementos desde varios archivos:

“CacheService” desde el archivo “CacheService” en el directorio “cache”.

“AuthService” desde el archivo “auth-service” en el directorio “auth”.

Todos los elementos exportados desde el archivo “útil”.

Todos los elementos exportados desde el archivo “types”.

“AuthService” desde el archivo “AuthService” en el directorio “auth”.

-types.ts:

Esta interfaz define una estructura de datos para representar un evento. En este caso, un evento tiene un identificador único representado por “evt_id”, que es un número.

-util.ts:

“isValidVersion(version: string)”: Verifica si una cadena de versión sigue el formato semántico de versión (SEM_VERSION_REGEX).

“toCamelCase(word: string): string”: Convierte una cadena en formato kebab-case a UpperCamelCase.

“transformHeadersToCamelCase(headers: { [k: string]: string | string[] | undefined }): { [k: string]: string | string[] }”: Convierte todas las claves de un objeto de encabezados HTTP de formato kebab-case a UpperCamelCase.

“hashKeyFn<T>(data: T): string”: Genera un hash MD5 para un objeto JSON dado.

“urlParams(base: string, objectParams: { [k: string]: string }): string”: Convierte un objeto de parámetros de URL en una cadena de consulta y la concatena con una URL base dada.

*(config):

-(db):

-index.ts:

Este código exporta todo desde los archivos “redis” y “sql”, lo que significa que todas las funciones, clases o variables exportadas desde esos archivos estarán disponibles para ser importadas desde otros módulos.

-redis.ts:

“getRedisClient()”: Devuelve el cliente Redis actual.

“connectRedisClient(options?, requestId?): Promise<void>”: Establece una conexión con Redis utilizando la biblioteca “ioredis”. Las opciones de configuración pueden ser proporcionadas opcionalmente, así como un identificador de solicitud para registrar eventos de conexión y errores.

-sql.ts:

Este código maneja la conexión y las consultas a una base de datos PostgreSQL:

*Cliente PostgreSQL:

-Utiliza la biblioteca “pg” para interactuar con la base de datos PostgreSQL.

-Define tipos para los argumentos de las consultas SQL y las filas de respuesta de la base de datos.

*Función “connect(requestId?)”:

-Crea y establece una conexión con la base de datos utilizando la configuración proporcionada.

*Función “sql({ query, bind })”:

-Ejecuta una consulta SQL en la base de datos utilizando el cliente PostgreSQL creado.

-Devuelve una promesa que se resuelve con las filas de respuesta de la base de datos.

-(inversify):

-index.ts:

*Importaciones:

-Se importan las clases necesarias, como “Container” de InversifyJS y algunas constantes y clases de otros módulos.

*Configuración del contenedor:

-Se crea un nuevo contenedor de InversifyJS con el alcance predeterminado configurado como “Singleton”. Esto significa que cada vez que se solicite una dependencia, el contenedor devolverá la misma instancia única.

*Enlace de dependencias:

-Se vinculan las implementaciones con las interfaces o clases correspondientes utilizando los símbolos definidos en `SYMBOLS`. Esto se hace mediante el método “.bind()” del contenedor.

-Por ejemplo, “CacheService” se vincula con “RedisImpl” y “IAuthService” se vincula con “AuthService”.

*Controladores y casos de uso:

-Se vinculan los controladores y casos de uso necesarios para la lógica de la aplicación. Esto permite que el contenedor IoC gestione la creación y resolución de estas dependencias automáticamente.

*Repositorio:

-Se vincula la implementación del repositorio de usuario (UserRepositoryImpl) con la interfaz del repositorio (UserRepository). Esto facilita el cambio de implementaciones sin modificar el resto del código que depende de la interfaz del repositorio.

*Exportación del contenedor:

-Finalmente, el contenedor configurado se exporta para que pueda ser utilizado en otras partes del código de la aplicación.

-(symbols):

-index.ts:

Este código define un conjunto de símbolos únicos para identificar diferentes dependencias en el contenedor IoC. Cada símbolo está asociado con un nombre descriptivo que representa el tipo de servicio o caso de uso que identifica. Estos símbolos se utilizan luego en el

contenedor para vincular las implementaciones con las interfaces o clases correspondientes.

*(constants):

-code-errors.ts:

Este código define un objeto llamado “CODE_ERRORS” que contiene códigos de error comunes. En este caso, solo hay un código de error definido: “UNAUTHORIZED”, que probablemente se utiliza para representar un error de falta de autorización en el sistema. Este enfoque proporciona una forma centralizada de gestionar y referenciar códigos de error en la aplicación.

-default.ts:

Este código exporta una constante llamada “ERROR_MESSAGE” que contiene el mensaje de error "Internal Server Error". Esta constante probablemente se utiliza para proporcionar un mensaje predeterminado cuando se produce un error interno en el servidor. Es útil para mantener la coherencia en los mensajes de error en toda la aplicación.

-game.ts:

Este código exporta un objeto llamado “game” con dos propiedades:

*”status”:

-Contiene dos valores: "VICTORY" para indicar la victoria y "LOSER" para indicar la derrota del jugador.

*”messages”:

-Contiene tres mensajes relacionados con el juego: "Word Invalid", "Init Game Needed", y "Game Over". Estos mensajes se utilizan para comunicar diferentes estados o eventos dentro del juego.

-index.ts:

Este código exporta varios elementos desde diferentes archivos:

-Desde el archivo “redis”, exporta todo bajo el nombre de “redis”.

-Desde el archivo “default”, exporta todo bajo el nombre de “defaults”.

-Desde el archivo “code-errors”, exporta todo.

-Desde el archivo “game”, exporta todo.

-redis.ts:

Estas constantes representan diferentes duraciones de tiempo en segundos, utilizadas para configurar el tiempo de vida de los datos almacenados en Redis:

-ONE_MINUTE: 1 minuto.

- ONE_HOUR: 1 hora.

-ONE_DAY: 1 día.

-ONE_WEEK: 1 semana.

-ONE_MONTH: 1 mes.

*(controllers):

(user):

-sign-in.ts:

Este código define un controlador de Express para el proceso de inicio de sesión de usuario:

*Importaciones: Importa los módulos necesarios y algunas constantes.

*Clase “UserSigninController”: Es un controlador que extiende “BaseController” y está decorado para permitir la inyección de dependencias.

*Constructor: Recibe el caso de uso “SigninUseCase”.

*Método “execute(request, response)”:

-Maneja las solicitudes de inicio de sesión.

-Valida los datos de entrada.

-Ejecuta el caso de uso de inicio de sesión.

-Responde con el resultado o un error HTTP 500.

-sign-up.ts:

Este código define un controlador de Express para el proceso de registro de usuario:

*Importaciones: Importa los módulos necesarios, incluyendo aquellos para la inyección de dependencias y la manipulación de solicitudes y respuestas HTTP.

*Clase “UserSignupController”: Es un controlador de Express que extiende “BaseController” y está decorado para permitir la inyección de dependencias.

*Constructor:** Recibe el caso de uso “SignupUseCase”.

*Método “execute(request, response)”:

- Maneja las solicitudes de registro de usuario.
- Valida los datos de entrada.
- Ejecuta el caso de uso de registro de usuario.
- Responde con el resultado o un error HTTP 500.

-base-controllers:

Este código define una clase abstracta llamada “BaseController” que sirve como plantilla para otros controladores de Express en la aplicación:

-Tiene un método abstracto “execute(request, response)” que debe ser implementado por las clases que la extiendan.

-Proporciona métodos “ok()” y “fail()” para enviar respuestas exitosas y manejar errores respectivamente.

-Utiliza un middleware “errorHandler” para manejar errores de manera uniforme en toda la aplicación.

-Registra información de finalización de solicitud en la consola.

-index.ts:

Este código exporta todo desde dos archivos: “'./user/sign-up'” y “'./user/sign-in'”. Probablemente estos archivos contienen lógica relacionada con el proceso de registro y inicio de sesión de usuarios, respectivamente. Utilizar “export *” permite exportar todo lo que está disponible en esos archivos para que pueda ser importado desde otros lugares de la aplicación.

*(data-access):

(cache):

-redis-impl.ts:

Este código implementa una clase llamada “RedisImpl” que actúa como una implementación concreta del servicio de caché utilizando Redis. Proporciona métodos para interactuar con Redis, incluyendo la obtención y establecimiento de valores, memorización de funciones y eliminación de claves.

(user):

-user-repository:

Este código implementa un repositorio de usuarios con métodos para crear usuarios y obtener usuarios por su nombre de usuario. Utiliza consultas SQL para interactuar con la base de datos.

Index.ts:

Este código exporta dos elementos:

-RedisImpl` desde el archivo “./cache/redis-impl”.

-Todo desde el archivo “./user/user-repository”.

*(domain):

(user):

-sign-in.ts:

Este código implementa la clase “Signin”, que representa un caso de uso para el inicio de sesión de usuario. Utiliza un repositorio de usuarios para obtener información de usuario y un servicio de autenticación para generar tokens de acceso. El método “execute” maneja la lógica del inicio de sesión, verificando las credenciales del usuario y generando un token de acceso si son válidas.

-sign-up.ts:

Este código implementa la clase `Signup`, que representa un caso de uso para el registro de usuario. Utiliza un repositorio de usuarios para crear nuevos usuarios en la base de datos y un servicio de autenticación para hashear contraseñas. El método `execute` maneja la lógica del registro, hasheando la contraseña proporcionada y creando un nuevo usuario en la base de datos.

-user-repository.ts:

Esta interfaz proporciona una abstracción para interactuar con el almacenamiento de usuarios en la aplicación. Este código define una interfaz llamada “UserRepository” con dos métodos:

“create(data: User): Promise<void>”: Crea un nuevo usuario.

“getUserByUserName(userName: string): Promise<any>”: Obtiene un usuario por su nombre de usuario.

-user.ts:

Este código define un tipo “User” con tres propiedades: “id”, “userName” y “password”. Representa la estructura de datos de un usuario en la aplicación.

-UseCase:

Este código define un tipo llamado “CustomRequestId” con una propiedad “requestId” de tipo “string”. Luego, define una interfaz genérica llamada “UseCase”, que especifica la estructura básica de un caso de uso en la aplicación. El método “execute” toma un objeto de tipo “RequestType” que debe tener una propiedad “requestId” y devuelve una promesa o un resultado del tipo “ResponseType”.

-index.ts:

Este código exporta varios elementos:

-Exporta la interfaz “UseCase” desde el archivo “./UseCase”.

-Exporta todo desde el archivo “./user/user-repository”.

-Exporta todo desde el archivo “./user/sign-up”.

-Exporta todo desde el archivo “./user/sign-in”.

*(interfaces):

-custom-errors.ts:

Este código define dos interfaces:

*“IJsonObject”: Una interfaz que representa un objeto JavaScript donde todas las claves son cadenas y todos los valores son cadenas.

*“ICustomError”: Una interfaz que extiende la interfaz `Error`. Define las siguientes propiedades adicionales:

-code: Una cadena que representa el código de error.

-vars: Un objeto que contiene variables adicionales relacionadas con el error.

-httpCode: Un número que representa el código de estado HTTP asociado con el error.

-index.ts:

Este código exporta varias interfaces y tipos:

“ICustomError” y “IJsonObject” desde el archivo “./custom-error”.

“IGetEntityOlimpoRepository” desde el archivo “./olimpo-graphql/IGetEntityOlimpoRepository”.

“IRequestOlimpoRepository” desde el archivo “./olimpo-graphql/IRequestOlimpoRepository”.

*(platform):

(config):

-index.ts:

Este código exporta una variable “config” que contiene la configuración de la aplicación, incluyendo detalles como el entorno, el puerto, la dirección IP, la configuración de la caché y de la base de datos, así como la configuración de autenticación. Estos valores se obtienen de las variables de entorno correspondientes.

-types.ts:

Este código define un tipo llamado “Config”, que especifica la estructura de la configuración de la aplicación. Contiene propiedades para el entorno, el puerto, la dirección IP, la configuración de la caché, la configuración de la base de datos, la configuración del proyecto y la configuración de autenticación.

(lib/class):

-general-errors.ts;

Este código define una clase “CustomError” que se utiliza para crear errores personalizados. Tiene propiedades para el código de error, el mensaje, variables y el código HTTP asociado. El constructor inicializa estos valores.

(middlewares):

-error-handler:

Este código define dos middlewares para manejar errores en Express. “errorHandler” imprime el error y construye la respuesta de error adecuada, mientras que “errorHandlerMiddleWare” sigue la firma estándar de los middlewares de manejo de errores en Express y llama a “errorHandler”.

-validate-token:

Este código define un middleware para Express que valida si hay un token de acceso en la solicitud. Si no hay ningún token, se lanza un error de autenticación no autorizada. Si se encuentra un token, se verifica utilizando la clave secreta de autenticación. Si la verificación es exitosa, el middleware permite que la solicitud continúe. Si hay algún error durante este proceso, se pasa al siguiente middleware de manejo de errores.

(server):

-express.ts:

Este código es una función que inicializa un servidor Express con middlewares y manejo de errores preconfigurados. Toma manejadores de solicitudes y opciones de configuración como entrada, y luego monta los manejadores en la ruta base especificada. Finalmente, inicia el servidor en el puerto y host proporcionados, y establece conexiones a la base de datos y Redis si están configurados.

-index.ts:

Entendido. Este código exporta la función “startExpressServer” desde el archivo “express.ts”, que se encarga de iniciar un servidor Express con configuraciones predefinidas. También exporta los tipos “CustomRequest” e “IStartOptions” desde el archivo “types.ts”, que proporcionan tipos específicos para solicitudes personalizadas y opciones de inicio del servidor.

-types.ts:

Este fragmento de código define dos tipos de TypeScript.

“CustomRequest`: Este tipo extiende el tipo “Request” de Express y agrega algunas propiedades personalizadas como “requestId”, “dataTokenUser”, “sourceApp”, “versionApp” y “accessToken”. Estos campos adicionales permiten adjuntar información adicional a las solicitudes HTTP recibidas por el servidor Express.

“IStartOptions”: Este tipo describe las opciones de configuración para iniciar el servidor Express. Incluye propiedades como “basePath” (la ruta base para el enrutamiento de la aplicación), “port” (el número de puerto en el que se escucharán las solicitudes), “host” (la dirección IP o el nombre de host en el que se ejecutará el servidor), “requestId” (el identificador único para la solicitud) y opcionalmente “corsOrigin” (la configuración de origen para la política de misma origen).

-index.ts:

El código exporta la configuración de la aplicación, una función para iniciar el servidor Express con opciones personalizadas y un tipo personalizado para las solicitudes de Express.

*(routes/v1):

(v1):

-index.ts:

Este código exporta un enrutador de Express para las rutas de la versión 1 de la API. Estas rutas incluyen rutas relacionadas con los usuarios, que se definen en el enrutador de usuario (userRouter). Las rutas de la versión 1 de la API están precedidas por el prefijo “/v1”.

-user.ts:

Este fragmento de código configura las rutas relacionadas con las operaciones de registro (signup) e inicio de sesión (signin) para usuarios en una API. Utiliza el enrutador de Express para definir las rutas “/signin” y “/signup”, que están asociadas a los controladores “UserSignupController” y “UserSigninController”, respectivamente. Estos controladores son obtenidos del contenedor de inversión de dependencias (`container`) configurado previamente. Cuando se recibe una solicitud POST en estas rutas, se llama al método “execute” de los controladores correspondientes para manejar la solicitud y enviar la respuesta.

*(util):

-index.ts:

Vacio.

-promises.ts:

Este bloque de código proporciona tres funciones útiles para trabajar con promesas en JavaScript:

“sequentialPromises”: Ejecuta todas las promesas secuencialmente, almacenando los resultados en un acumulador.

“memoizePromise”: Memoiza la respuesta de una función asíncrona para evitar ejecutarla varias veces con los mismos argumentos.

“executeArrayPromises”: Ejecuta un array de promesas y devuelve solo las que se ejecutaron con éxito, proporcionando opciones para personalizar su comportamiento.

-index.ts:

Este bloque de código inicializa y arranca un servidor Express que utiliza las rutas de la versión 1 (v1Routes) y la configuración proporcionada por “config”. Utiliza un identificador único generado por UUID (executorId) para identificar la instancia del servidor.