Universidad Autónoma de Querétaro

Facultad de Informática

Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
18/05/2024

Proyecto de Patrones de Diseño

1. Carpeta **.vscode (settings.json)**

- Este fragmento de código en settings.json es una configuración específica para la extensión **cSpell** que personaliza las palabras reconocidas como correctas en la verificación ortográfica.

2. Carpeta **db/migrations**

- **(1. 20221212191745_wordle_sqs.sql)**

-- **migrate:up** Esta línea indica que las instrucciones siguientes deben ejecutarse cuando se realice una migración "hacia arriba" **CREATE SEQUENCE** seq_wordle_user_wus_id Es un objeto de base de datos que genera valores numéricos secuenciales. En este caso, la secuencia se inicia en 1 y aumenta en 1 con cada llamada. **INCREMENT BY** 1 Indica que la secuencia se incrementará en 1 con cada llamada. **MINVALUE** 1 Especifica el valor mínimo que puede tener la secuencia. **MAXVALUE** 9223372036854775807 Especifica el valor máximo que puede tener la secuencia. **START** 1 Indica el valor inicial de la secuencia. **CACHE** 1 Especifica cuántos valores deben almacenarse en caché para mejorar el rendimiento. **NO CYCLE;** Indica que la secuencia no debe reciclar los valores una vez que se alcanza el valor máximo. **-- migrate:down** Esta línea indica que las instrucciones siguientes deben ejecutarse cuando se deshaga la migración, es decir, cuando se revierta la migración para retroceder los cambios en la base de datos. **DROP SEQUENCE** seq_wordle_user_wus_id Esta es una instrucción SQL que elimina la secuencia seq_wordle_user_wus_id creada anteriormente.	• Es un script de migración que crea una secuencia en la base de datos cuando se ejecuta hacia arriba y la elimina cuando se ejecuta hacia abajo. • DROP SEQUENCE **seq_wordle_user_wus_id:** Esta es una instrucción SQL que elimina la secuencia **seq_wordle_user_wus_id** creada anteriormente.

- **(2. 20221212191802_user-table.sql)**

-- migrate:up **CREATE TABLE** user_account (usr_act_id int8 **NOT NULL** DEFAULT nextval('seq_wordle_user_wus_id'::regclass), usr_act_name **varchar**(20) **NOT NULL**, usr_act_password **varchar**(150) **NOT NULL**,	**Abstracción:** El código define una tabla de base de datos llamada **"user_account"** con columnas que representan atributos de una entidad de usuario. Este enfoque abstracto refleja un sistema de base de datos relacional.

usr_act_creation_date **timestamp NOT NULL** DEFAULT now(), usr_act_update_date **timestamp NOT NULL** DEFAULT now(), usr_act_delete_date **timestamp NULL,** **CONSTRAINT** pk_usr_act_id **PRIMARY KEY** (usr_act_id), **CONSTRAINT** user_unique UNIQUE (usr_act_name)); **COMMENT ON COLUMN "user_account"."usr_act_id" IS** 'This column represent the id of the entity. It composes the primary key of the table.'; **CREATE INDEX usr_act_name_idx ON** user_account USING btree (usr_act_name); **CREATE INDEX usr_act_delete_date_idx ON** user_account USING btree (usr_act_delete_date); -- migrate:down **ALTER TABLE** "user_account" DROP **CONSTRAINT** "pk_usr_act_id"; **DROP INDEX** "usr_act_name_idx"; **DROP INDEX** "usr_act_delete_date_idx"; **DROP TABLE** "user_account";	**Encapsulamiento:** Cada columna de la tabla tiene un tipo de dato y restricciones definidas, lo que encapsula los datos relacionados y sus comportamientos. El fragmento de código define una tabla de base de datos relacional (CREATE TABLE user_account), lo que implica el uso de una **base de datos SQL (**Structured Query Language).

3. Carpeta **src**

➔ **Carpeta common**

➢ Carpeta **auth**

○ **Archivo AuthService.ts**

Principios SOLID:

- **Interface Segregation Principle**: La interfaz IAuthService sigue el principio de segregación de interfaces al definir un contrato claro y específico para el servicio de autenticación. Cada método está relacionado con la autenticación y la gestión de contraseñas, evitando métodos no relacionados.
- **Dependency Inversion Principle**:La aplicación sigue el principio de inversión de dependencias, esto permite que el código dependa de abstracciones en lugar de implementaciones concretas.

Pilares de POO (Programación Orientada a Objetos):

- **Abstracción:** La interfaz IAuthService es una abstracción que define las operaciones que cualquier implementación de servicio.
- **Encapsulamiento:** Las implementaciones de IAuthService encapsulan de la generación, verificación de tokens, la gestión

de contraseñas, exponiendo solo los métodos definidos en la interfaz..

- **Polimorfismo:** Las implementaciones concretas de IAuthService pueden ser intercambiables, permitiendo que el código que dependa de IAuthService utilice diferentes implementaciones sin cambiar su propia lógica.

Patrones de Diseño:

- **Strategy**: IAuthService define la estrategia y diferentes implementaciones proporcionan diferentes mecanismos de autenticación.
- **Adapter**: IAuthService utiliza para adaptar una biblioteca de autenticación existente a una interfaz estándar que podría considerarse un patrón Adapter.
- **Proxy:** podría utilizarse un proxy para envolver una implementación de IAuthService.

Arquitectura Clean Architecture:

- **Capas:** IAuthService estaría en la capa de dominio, definiendo una abstracción que no depende de detalles de implementación, las implementaciones concretas estarían en la capa de infraestructura.
  - **Archivo auth-service.ts**

Principios SOLID:

- **Single Responsibility Principle:** La clase AuthService tiene una única responsabilidad relacionada con la autenticación: generación y verificación de tokens y hash de contraseñas.
- **Open/Closed Principle:** La clase AuthService está abierta para agregar nuevas funcionalidades, pero cerrada para modificación ya que las funcionalidades existentes no necesitan ser modificadas.
- **Liskov Substitution Principle**: Cualquier clase que implemente la interfaz IAuthService debe ser intercambiable con AuthService sin romper la funcionalidad.
- **Dependency Inversion Principle**: AuthService depende de la abstracción IAuthService y no de una implementación concreta. Además, la clase AuthService está marcada con @injectable(), lo que sugiere el uso de Inversify para la inyección de dependencias, promoviendo la inversión de dependencias.

Pilares de POO:

- **Abstracción**:La interfaz IAuthService es una abstracción que define las operaciones de autenticación que deben implementarse.
- **Encapsulamiento**:AuthService encapsula la lógica de autenticación y no expone su implementación interna a los de la clase.

- **Herencia:** Cualquier clase que implemente IAuthService puede heredar de otra clase base.
- **Polimorfismo:** Se permite que cualquier implementación de IAuthService sea utilizada de manera intercambiable con AuthService.

Patrones de Diseño:

- **Singleton**:La clase AuthService podría ser utilizada por estar gestionado por el contenedor de inyección de dependencias (Inversify).
- **Dependency Injection**:El uso de @injectable() indica que AuthService está diseñada para ser gestionada por un contenedor de dependencias.
- **Strategy**: AuthService actúa como una implementación de una estrategia de autenticación definida por IAuthService.
- **Adapter** :AuthService podría ser visto como un adaptador que para las funcionalidades de jsonwebtoken y bcryptjs a una interfaz de autenticación.
- **Arquitectura Clean Architecture:**El AuthService pertenecería a la capa de implementación de servicios (aplicación), y IAuthService a la capa de dominio. Esto asegura que la lógica de negocio no dependa de detalles de implementación.
- **Base de Datos:**Basándonos en la configuración proporcionada en los archivos anteriores (DB_HOST, DB_PORT, DB_USERNAME, DB_PASSWORD, DB_NAME) es una base de datos SQL.

➢ **Carpeta cache**

- **Archivo CacheService.ts**

Principios SOLID:

- **Single Responsibility**: La interfaz CacheService tiene una única responsabilidad de definir las operaciones de un servicio de caché.
- **Interface Segregation:** La interfaz CacheService está bien definida y específica para las necesidades de un servicio de caché. No obliga a implementar métodos no relacionados, cumpliendo así con el principio de segregación de interfaces.

Pilares de POO:

- **Abstracción:** La interfaz CacheService es una abstracción que define las operaciones de caché sin exponer detalles de implementación.
- **Encapsulamiento:** Las implementaciones de CacheService encapsulan exponiendo solo los métodos definidos en la interfaz.
- **Polimorfismo:** Las implementaciones concretas de CacheService pueden ser intercambiables, permitiendo que el

código que dependa de CacheService utilice diferentes implementaciones sin cambiar su propia lógica.

Patrones de Diseño:

- **Strategy:** Cada implementación de CacheService puede representar una estrategia de caché distinta.
- **Adapter**: Si CacheService se utiliza para adaptar una biblioteca de caché existente a una interfaz estándar de la aplicación, podríamos considerarlo un patrón Adapter.
- **Decorator**:El método memoize podría utilizarse como parte de un patrón Decorator, añadiendo comportamiento de caché a otros métodos de manera transparente.
- **Proxy**: CacheService puede ser utilizado como un proxy para añadir caché a las operaciones, controlando el acceso a los datos en caché.
- **Arquitectura Clean Architecture**: Capas: CacheService pertenecería a la capa de dominio, definiendo una abstracción que no depende de detalles de implementación, mientras tanto las implementaciones concretas estarían en la capa de infraestructura.
- **Base de Datos**:Tiene una base de datos NoSQL.

- **Archivo index.ts**

Principios SOLID:

- **Single Responsibility**: El CacheService se encarga de la gestión de la caché, mientras que AuthService maneja la autenticación.
- **Open/Closed:** Los servicios están abiertos para extensión pero cerrados para modificación, también se puede extender los añadiendo nuevas funcionalidades sin modificar el código existente.
- **Liskov Substitution**: Cualquier clase que implemente CacheService o AuthService puede ser usada de manera intercambiable sin alterar el comportamiento del programa.
- **Interface Segregation**: Los códigos están segregados en interfaces de CacheService, IAuthService, evitando la implementación de métodos innecesarios.
- **Dependency Inversion**: Los módulos de alto nivel de AuthService y CacheService no dependen de módulos de bajo nivel, sino de abstracciones, esto por la dependencia @injectable() en AuthService.

Pilares de POO:

- **Abstracción:**Las interfaces CacheService y IAuthService proporcionan una capa de abstracción sobre las implementaciones.

- **Polimorfismo:** Permite que diferentes implementaciones de CacheService y IAuthService sean intercambiables, proporcionando flexibilidad en el uso de estas clases.

Patrones de Diseño:

- **Dependency Injection**: El AuthService utiliza @injectable(), lo que sugiere el uso de un contenedor DI (como Inversify).
- **Strategy**: Podrían implementarse estrategias distintas para autenticación y gestión de la caché.
- **Adapter**: El AuthService y CacheService podrían adaptar la funcionalidad de librerías externas.
- **Proxy:** El CacheService podría implementarse como un proxy para gestionar el acceso a la caché.
- **Clean Architecture:** Capas: En una arquitectura limpia, AuthService y CacheService estarían en la capa de aplicación o de servicios, mientras que las interfaces IAuthService, CacheService estarían en la capa de dominio.
- **Base de Datos: NoSQL o SQL:** La información anterior sugiere el uso de Redis para la caché, que es una base de datos NoSQL, mientras para la base de datos principal, que es una base de datos SQL.

- **Archivo types.ts**

Principios SOLID

- **Single Responsibility:** La interfaz Event tiene una responsabilidad que es representar un evento con un identificador.
- **Open/Closed:** La interfaz Event está abierta a extensión, donde podemos añadir más propiedades nuevas a través de la herencia.
- **Liskov Substitution**: Cualquier implementación que extienda Event debe poderse sustituir sin romper la funcionalidad que ya tiene.

Pilares de POO

- **Abstracción**: Event es una abstracción que define la estructura de un evento.

Patrones de Diseño

- **Dependency Inversion**: La interfaz en sí sigue este principio al definir una abstracción sin acoplarse a implementaciones concretas.
- **Clean Architecture: Capas:** En una arquitectura limpia, Event estaría en la capa de dominio como una entidad.
- **Base de Datos NoSQL o SQL:** Es probable que Event se almacenen en una base de datos como SQL.

- **Archivo util.ts**

Principios SOLID

- **Single Responsibility:** Cada función tiene una responsabilidad que es validar una versión, convertir una cadena, transformar encabezados, generar un hash y construir una URL con parámetros.
- **Open/Closed** : Las funciones están abiertas a extensión, ya que nuevas funcionalidades pueden ser añadidas sin modificar las existentes.

Pilares de POO

- **Abstracción:** Las funciones proveen abstracción sobre operaciones comunes como hashing y manipulación de cadenas.
- **Encapsulamiento:** Cada función encapsula una operación. específica.

Patrones de Diseño

- **Adapter (Estructural):** El transformHeadersToCamelCase puede ser visto como un adaptador que transforma la estructura de los encabezados.
- **Clean Architecture: Capas:** En una arquitectura limpia, estas utilidades estarían en la capa de utilidades o servicios generales, accesibles desde cualquier parte del sistema sin depender de detalles de implementación específicos.

4. Archivo **.environment.example**

- El archivo especifica la configuración para una base de datos con las variables DB_HOST, DB_PORT, DB_USERNAME, DB_PASSWORD y DB_NAME., sin embargo no especifica directamente si es una base de datos **NoSQL o SQL**., pero las variables de configuración son compatibles tanto con bases de datos SQL como con bases de datos NoSQL se podrían usar alguna de las dos..

5. Archivo **.Dockerfile**

- El **Dockerfile** es un proceso de construcción modularizado utilizando múltiples etapas (base, builder, production) sugiere una posible preocupación por la cohesión y la separación de responsabilidades.
- **Arquitectura Clean Architecture**: El uso de múltiples etapas en la construcción de la imagen Docker sugiere una preocupación por la separación de preocupaciones y la modularidad, lo que podría alinearse con esto.

6. Archivo **README.md**

- **Dependency Inversion Principle**: Se menciona el uso de Dependency Injection, lo que sugiere la aplicación de inversión de dependencias.
- Se está utilizando **Clean Architecture** tiene una estructura modular que separa claramente las capas de la aplicación, como la capa de dominio, la capa de aplicación y la capa de infraestructura.

Universidad Autónoma de Querétaro
Facultad de Informática
Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
18/05/2024

- El archivo utiliza **SQL como base de datos** ya que se especifica en el archivo .env.local con la configuración de host, puerto, nombre de usuario, contraseña y nombre de la base de datos.

7. Archivo **docker-compose.yml**

- **Singleton:** El "Redis" podría considerarse como un patrón singleton ya que generalmente se configura y se accede a través de un único punto de entrada..
- **Dependency Inversion Principle**: la configuración de Docker Compose permite que los servicios dependan de otros servicios, lo que podría reflejar una inversión de dependencias.
- **Arquitectura Clean Architecture:** La configuración de Docker Compose proporciona una forma de desplegar la aplicación y sus dependencias en contenedores.
- **Base de Datos:** El archivo de configuración Docker Compose incluye servicios para la utilización de una base de datos SQL.

8. Archivo **environment.d.ts**

- **Base de Datos:** Las variables de entorno relacionadas con la base de datos (DB_HOST, DB_PORT, DB_USERNAME, DB_PASSWORD, DB_NAME, DB_TYPE, DB_POOL_SIZE, DB_SCHEMA_LOCKED, CONN_TIMEOUT, IDLE_TIMEOUT) se está configurando en una base de datos **SQL**.

9. Archivo **package.json**

- **Dependency Inversion Principle**: La dependencia en inversify sugiere que se están aplicando principios de inversión de dependencias.
- **Singleton:** El "express" podría estar implementando el patrón singleton.
- **Arquitectura Clean Architecture:** El código proporcionado parece estar orientado a una arquitectura de "backend" limpia.

10. Archivo **tsconfig.app.json**

- **Dependency Inversion Principle:** el uso de "types": ["node", "jest"] indica que el proyecto depende de las definiciones de tipos para Node.js y Jest. Esto puede considerarse una forma de dependencia inversa, el proyecto depende de abstracciones en lugar de implementaciones concretas.
- **Clean Architecture**: A, al excluir los archivos de prueba ("exclude": ["**/*.test.ts"]) y al incluir archivos TypeScript y JSON ("include": ["**/*.ts", "**/*.json"]), tiene una estructura organizativa de un enfoque de arquitectura limpia, con una separación clara de las diferentes capas y la exclusión de elementos no esenciales para la compilación.

11. Archivo **.tsconfig.json**

- **Dependency Inversion Principle**: Al usar ("paths") en la configuración, el proyecto puede depender de abstracciones en lugar de implementaciones concretas.
- **Clean Architecture**: La configuración incluye rutas para diferentes partes del proyecto, como controladores, constantes, interfaces, repositorios, servicios…

- Al establecer "experimentalDecorators": true y "emitDecoratorMetadata": true, el proyecto parece utilizar decoradores, lo que podría indicar la aplicación de estructurales (Decorator).

12. Archivo **.tsconfig.spec.json**

- **Dependency Inversion Principle** Aunque no se aplica directamente en este código, al extender otro archivo de configuración (tsconfig.json), se podría decir que sigue el principio de al permitir la configuración a través de abstracciones.
- La inclusión de "types": ["node", "jest"] especifica los tipos de datos que se utilizarán en el proyecto.
- **Composite Pattern:** El uso de "composite": true indica que se está habilitando el patrón Composite en el proyecto.

13. Archivo **tslint.json**

- Esta configuración se utiliza para definir reglas de estilo y convenciones de codificación que deben seguirse en el proyecto.
- Algunas reglas como "prefer-const", "arrow-return-shorthand", "no-debugger", "no-console" están relacionadas con la legibilidad y la seguridad del código, lo que podría considerarse como una aplicación indirecta de principios de programación como la legibilidad del código y para evitar prácticas peligrosas.
- Los nombres de las variables y las funciones ("variable-name", "member-ordering") podrían considerarse como una aplicación de estructuración de clases y funciones.
- Las reglas sobre directivas y selectores de componentes ("directive-selector", "component-selector") pueden estar relacionadas con la arquitectura de componentes.
- La exclusión de ciertos directorios y archivos en las opciones del linter ("exclude") puede ser para evitar que el linter analice automáticamente ciertos archivos o directorios generados o no relevantes.

14. Archivo **webpack.config.js**

Dependency Inversion Principle En el código, webpack-node-externals se utiliza para excluir los módulos de Node.js del paquete compilado.
Adapter: El uso de TsconfigPathsPlugin podría considerarse una forma de adaptar la configuración de TypeScript a la configuración de Webpack.
En cuanto a la arquitectura, el código parece seguir un enfoque de **"Clean Architecture"** ya que está estructurado en capas bien definidas y separadas. La configuración de Webpack en sí misma no define la arquitectura del proyecto, solo facilita su compilación.