



# Universidad Autónoma de Querétaro
# Facultad de Informática

Proyecto final

**Alumno:** Duarte Fregoso Josafat Axel
**Profesor:** Enrique Aguilar

**Fecha: 17/05/2024**

**Análisis de los patrones de diseño del programa**

1. .vscode\settings.json

El archivo que se muestra, se incluye la palabra inversify, sirve para evitar que el corrector ortografico marque inversify como un error, esto esta relacionado con el patron de inyección de dependencias, este sirve para permitir a un objeto recibir otras instancias de objetos (dependencias) desde una entidad externa

2. db\migrations\20221212191745_wordle_sqs.sql

Este archive contiene commandos SQL, se utilizan para la migración de la base de datos, este tiene un enfoque de patron de diseño de gestión de base datos, utiliza el patron de migración, este se utiliza para gestionar cambios incrementales de manera controlada, también permite aplicar y revertir cambios

3. db\migrations\20221212191802_user-table.sql

Al igual que el archivo anterior, este también implementa el patron de migración, pero, también utiliza el patron de tabla de base de datos, este hace una definición estructurada de una tabla en una base de datos relacional

4. src\common\auth\auth-service.ts

Este archivo implementa varios patrones de diseño como vimos en el primero, utiliza una inyección de dependencias en la clase de AuthService, después el ptron de servicio, que este define una clase que proporciona funcionalidades reutilizables, en este caso encapsula todas las funcionalidades de autenticación.
Y por ultimo adapter (visto en clase) donde adapta las funcionales de bcrypt y jsonwebtoken

5. src\common\auth\AuthService.ts

Utiliza el patron de interfaz, que define un contrato que otras clases pueden implementra, esto ayuda a lograr la abstracción, este se relacionada con el anterior

6. `CacheService.ts`

Este archivo define la interfaz `CacheService`, que implementa el patrón de Interfaz para abstraer y flexibilizar las implementaciones del servicio de caché. Incluye métodos como `getByKey`, `setByKey`, y `deleteByKey`, encapsulando funcionalidades reutilizables típicas de un patrón de Servicio. Además, el método `memoize` utiliza el patrón de Estrategia, permitiendo aplicar dinámicamente distintas estrategias de memoización. Estos

patrones facilitan la inyección de dependencias y la sustitución de implementaciones específicas según las necesidades del proyecto.

7. `index.ts`

Este archivo implementa el patrón de Fachada al proporcionar un único punto de acceso a múltiples módulos y servicios, como `CacheService`, `AuthService`, utilidades y tipos. Al exportar estos componentes desde un único archivo, simplifica la importación en otras partes del proyecto, mejorando la organización y reduciendo la complejidad al encapsular las dependencias en un solo módulo centralizado.

8. `types.ts`

Este archivo define la interfaz `Event`, implementando el patrón de Tipo de Datos. Este patrón proporciona una estructura clara y consistente para los datos, asegurando la integridad y facilitando la manipulación de eventos en el sistema.

9. `util.ts`

Este archivo implementa el patrón de Utilidad (Utility Pattern), proporcionando una colección de funciones utilitarias reutilizables para operaciones comunes como validación de versiones, conversión de formato de cadenas, transformación de encabezados, hashing de datos y construcción de URLs con parámetros. Estas funciones están encapsuladas en un módulo utilitario, promoviendo la reutilización del código y la separación de preocupaciones.

10. `index.ts`

Este archivo implementa el patrón de Fachada, proporcionando un único punto de acceso a las configuraciones de base de datos para Redis y SQL. Al exportar estos módulos desde un solo archivo, se simplifica la importación y gestión de configuraciones de base de datos en otras partes del proyecto, mejorando la organización y reduciendo la complejidad al encapsular las dependencias relacionadas con la configuración de la base de datos.

11. `src\config\db\redis.ts`

Este archivo implementa varios patrones de diseño:

Singleton: La variable `redisClient` se utiliza como una instancia única de Redis, asegurando que solo haya una conexión activa en toda la aplicación.

Fachada: Las funciones `getRedisClient` y `connectRedisClient` proporcionan una interfaz simplificada para interactuar con Redis, ocultando la complejidad de la configuración y manejo de eventos de Redis.
Adaptador: La función `connectRedisClient` adapta la configuración de Redis (`Redis.RedisOptions`) para trabajar con la configuración específica del proyecto (`config.cache`).

12. `src\config\db\sql.ts`

Este archivo implementa varios patrones de diseño:

Singleton: La variable `client` se utiliza para mantener una única instancia del cliente de PostgreSQL, asegurando que solo haya una conexión activa a la base de datos en toda la aplicación.
Fachada: Las funciones `connect` y `sql` proporcionan una interfaz simplificada para interactuar con PostgreSQL, ocultando la complejidad de la configuración y manejo de conexiones.
Adaptador: La función `connect` adapta la configuración de PostgreSQL (`Client`) para trabajar con la configuración específica del proyecto (`config.db`).

13. `src\config\inversify\index.ts`

Este archivo implementa varios patrones de diseño:

Inyección de Dependencias (Dependency Injection): Utiliza `Inversify` para gestionar las dependencias, permitiendo la inyección de servicios y controladores en el contenedor.
Contenedor de Servicios (Service Container): El contenedor `Container` de `Inversify` actúa como un contenedor de servicios, centralizando la configuración y gestión de las dependencias.
Singleton: Al configurar el contenedor con `{ defaultScope: 'Singleton' }`, asegura que las instancias de los servicios y repositorios sean únicas y compartidas en toda la aplicación.
Fachada: El contenedor proporciona una interfaz simplificada para acceder a las diferentes dependencias, ocultando la complejidad de su configuración y resolución.

14. `src\config\symbols\index.ts`

Este archivo implementa el patrón de Localizador de Servicios (Service Locator) al definir un objeto `SYMBOLS` que contiene símbolos únicos para cada servicio o caso de uso en la aplicación. Estos símbolos actúan como identificadores para las dependencias, permitiendo

que el contenedor de inyección de dependencias (`Inversify`) resuelva y gestione las instancias correspondientes de manera centralizada y desacoplada.

15. `src\constants\index.ts`

Este archivo implementa el patrón de Fachada (Facade Pattern), proporcionando un único punto de acceso para importar varias constantes desde diferentes módulos (`redis`, `default`, `code-errors`, `game`). Esto simplifica la gestión y el acceso a estas constantes en otras partes del proyecto, mejorando la organización del código y reduciendo la complejidad.

16. `src\controllers\user\sign-in.ts`

Este archivo implementa varios patrones de diseño:

- Inyección de Dependencias (Dependency Injection): Utiliza los decoradores `@inject` y `@injectable` de `Inversify` para inyectar dependencias en el controlador `UserSigninController`, promoviendo la modularidad y facilitando la prueba de unidades.
- Controlador (Controller Pattern): La clase `UserSigninController` sigue el patrón de Controlador, manejando las solicitudes de inicio de sesión y coordinando las acciones entre el caso de uso de inicio de sesión (`SigninUseCase`) y la respuesta HTTP
- Fachada (Facade Pattern): Extiende `BaseController`, simplificando las interacciones con las solicitudes y respuestas HTTP mediante métodos utilitarios como `ok` y `fail`, ocultando la complejidad de las respuestas HTTP.

17. `src\controllers\user\sign-up.ts`

Este archivo implementa varios patrones de diseño: el patrón de Inyección de Dependencias (Dependency Injection) utilizando `@inject` y `@injectable` de `Inversify` para inyectar dependencias en el controlador `UserSignupController`, promoviendo la modularidad y facilitando las pruebas unitarias; el patrón de Controlador (Controller Pattern), donde `UserSignupController` maneja las solicitudes de registro de usuario, coordinando entre el caso de uso `SignupUseCase` y la respuesta HTTP; y el patrón de Fachada (Facade Pattern), extendiendo `BaseController` para simplificar las interacciones con solicitudes y respuestas HTTP mediante métodos utilitarios como `ok` y `fail`, ocultando la complejidad de las respuestas HTTP.

18. `src\controllers\base-controller.ts`

Este archivo implementa varios patrones de diseño: el patrón de Clase Base (Template Method) al definir la clase abstracta `BaseController`, que establece una estructura común para los controladores y obliga a las clases derivadas a implementar el método `execute`; el patrón de Fachada (Facade Pattern), proporcionando métodos utilitarios `ok` y `fail` para simplificar las respuestas HTTP y ocultar la complejidad de la gestión de errores y respuestas; y el patrón de Inyección de Dependencias (Dependency Injection), utilizando el decorador `@injectable` de `Inversify` para permitir que `BaseController` y sus subclases sean gestionadas por un contenedor de dependencias.

19. `src\controllers\index.ts`

Este archivo implementa el patrón de Fachada (Facade Pattern) al proporcionar un único punto de acceso para exportar múltiples controladores (`sign-up` y `sign-in`). Esto simplifica la importación y gestión de los controladores en otras partes del proyecto, mejorando la organización del código y reduciendo la complejidad.

20. `src\data-access\cache\redis-impl.ts`

Este archivo implementa el patrón de Repositorio (Repository Pattern) al definir `UserRepositoryImpl`, que proporciona una abstracción sobre las operaciones de acceso a datos para la entidad `User`. Esto permite manejar la persistencia de datos sin exponer detalles de la infraestructura de la base de datos. También utiliza el patrón de Inyección de Dependencias (Dependency Injection), utilizando el decorador `@injectable` de `Inversify` para gestionar la instancia de `UserRepositoryImpl` a través de un contenedor de dependencias.

21. `src\data-access\index.ts`

Este archivo implementa el patrón de Fachada (Facade Pattern) al proporcionar un único punto de acceso para exportar múltiples módulos de acceso a datos (`RedisImpl` y `user-repository`). Esto simplifica la importación y gestión de estos módulos en otras partes del proyecto, mejorando la organización del código y reduciendo la complejidad.

22. `src\domain\user\sign-in.ts`

Este archivo implementa varios patrones de diseño: el patrón de Caso de Uso (Use Case Pattern) al definir `Signin`, que encapsula la lógica específica del caso de uso de inicio de sesión de usuario; el patrón de Inyección de Dependencias (Dependency Injection) al utilizar `@inject` y `@injectable` de `Inversify` para inyectar las dependencias `UserRepository` y `AuthService`, promoviendo la modularidad y facilitando las pruebas unitarias; y el patrón de Repositorio (Repository Pattern) al depender de `UserRepository` para acceder y gestionar los datos del usuario.

23. `src\domain\user\sign-up.ts`

Este archivo implementa varios patrones de diseño: el patrón de Caso de Uso (Use Case Pattern) al definir `Signup`, que encapsula la lógica específica del caso de uso de registro de usuario; el patrón de Inyección de Dependencias (Dependency Injection) al utilizar `@inject` y `@injectable` de `Inversify` para inyectar las dependencias `UserRepository` y `AuthService`, promoviendo la modularidad y facilitando las pruebas unitarias; y el patrón de Repositorio (Repository Pattern) al depender de `UserRepository` para gestionar la creación de nuevos usuarios.

24. `src\platform\config\index.ts`

Este archivo implementa el patrón de Singleton al definir y exportar una única instancia de configuración (`config`) para toda la aplicación. Esto asegura que todos los módulos de la aplicación compartan la misma configuración, centralizando y unificando los valores de configuración.

25. `src\platform\middlewares\error-handler.ts`

Este archivo implementa el patrón de Cadena de Responsabilidad (Chain of Responsibility Pattern) mediante el middleware `errorHandlerMiddleWare` para manejar errores en una aplicación Express. Además, utiliza el patrón de Adaptador (Adapter Pattern) al adaptar `CustomError` para que se ajuste a la estructura de respuesta esperada del cliente, encapsulando la lógica de construcción de la respuesta de error en `buildResponse`.

26. `src\platform\middlewares\validate-token.ts`

Este archivo implementa el patrón de Cadena de Responsabilidad (Chain of Responsibility Pattern) mediante el middleware `validateToken` para manejar la validación de tokens en una aplicación Express. Este middleware verifica la presencia y validez de un token de acceso y pasa la solicitud al siguiente middleware en la cadena si el token es válido, o lanza un error si no lo es.

27. `src\routes\v1\user.ts`

Este archivo implementa varios patrones de diseño: el patrón de Fachada (Facade Pattern) al proporcionar un único punto de acceso para definir las rutas de usuario (`userRouter`); el patrón de Inyección de Dependencias (Dependency Injection) al utilizar un contenedor de `Inversify` para obtener instancias de `UserSignupController` y `UserSigninController`, promoviendo la modularidad y la testabilidad del código; y el patrón de Controlador (Controller Pattern) al delegar las solicitudes de las rutas a los métodos `execute` de los controladores `UserSignupController` y `UserSigninController`.