



UNIVERSIDAD
AUTÓNOMA DE
QUERÉTARO

Nombre: Diego Mendoza Chávez

Expediente: 327967

Docente: Enrique Aguilar Orozco

Materia / Taller : Patrones de Diseño

Facultad de Informática

Universidad Autónoma de Querétaro

Trabajo a entregar: Proyecto Final de Patrones de Diseño y Pilares del POO

Fecha de Entrega: 18/05/2024

Identificar Patrones de la carpeta "template-nodejs" (carpeta src)

index.ts

-Primer carpeta(common)

```
export { CacheService } from './cache/CacheService';
export { AuthService } from './auth/auth-service';
export * from './util';
export * from './types';
export * from './auth/AuthService';
```

Pilares del POO	
Abstracción	<p>El código define interfaces para CacheService y AuthService, ocultando los detalles de implementación y exponiendo solo las funcionalidades necesarias para su uso.</p> <p>Se utilizan clases para encapsular la lógica y los datos de cada servicio, promoviendo la modularidad y la reutilización de código.</p> <p>Encapsulamiento:</p> <p>No se observa el uso explícito de modificadores de acceso (p.ej., private, public) para controlar la visibilidad de propiedades y métodos.</p>
Herencia	<p>No se observa la definición de clases jerárquicas ni herencia en el código proporcionado.</p>
Polimorfismo	<p>No se observa la implementación de polimorfismo, como la sobrecarga o sobrescritura de métodos.</p>

Clean Architecture:	
No hay	No hay evidencia de la implementación de una estructura de capas o módulos bien definida en el código proporcionado.

Principios SOLID	
Single Responsibility	No se puede evaluar completamente si cada clase cumple con SRP sin un análisis más profundo de su funcionalidad específica.
Principio de Dependencia	No se observan dependencias directas entre clases en el código proporcionado.
Segregación de Interfaz	No se identifican interfaces extensas o con muchos métodos en el código.

Principios SOLID


Principio de "Open/Close"	El código no presenta una estructura que facilite la extensión de la funcionalidad sin modificar el código existente.
Principio de Liskov	No se pueden evaluar las violaciones de LSP sin un análisis más profundo del comportamiento de las clases y sus métodos.

Patrón/es de Diseño

No hay	No se identifican patrones de diseño específicos en el código proporcionado.
--------	--

types.ts

```
1 export interface Event {
2   evt_id: number;
3 }
4
```



Pilares del POO	
Abstracción	La interfaz Event define una estructura abstracta para representar un evento con un identificador (evt_id). Esta interfaz establece una plantilla para los objetos que se ajustan a este tipo de evento, sin revelar detalles de implementación específicos.
Encapsulamiento	La interfaz Event no expone detalles de implementación interna, solo la propiedad evt_id. Esto promueve la ocultación de información y el control del acceso a los datos del evento.
Polimorfismo	En este fragmento de código no se observa polimorfismo explícito. Sin embargo, la interfaz Event podría servir como base para la creación de clases derivadas que implementen diferentes comportamientos o atributos relacionados con eventos específicos.
Herencia	No hay evidencia de herencia directa en este código. Sin embargo, la interfaz Event podría ser utilizada como supertipo para definir clases jerárquicas de eventos con relaciones de herencia.

Clean Architecture:	
No hay	En base en el código no es posible determinar si se aplica la arquitectura limpia (Clean Architecture) ya que no se observa una estructura completa de capas y módulos bien definida.

Principios SOLID	
Single Responsibility	No se puede determinar si cada clase o módulo cumple con una única responsabilidad bien definida.
Principio de Dependencia	No se aprecian abstracciones ni interfaces que permitan la inversión de dependencias.
Segregación de Interfaz	No se observan interfaces segmentadas para evitar clientes voluminosos.
Principio de "Open/Close"	No se observa la capacidad de modificar el código sin afectar el comportamiento existente.

Principios SOLID

Principio de
Liskov

No se observa la posibilidad de sustituir objetos sin violar las invariantes del programa.

Patrón/es de Diseño

No hay

En este fragmento de código no se observa la implementación de un patrón de diseño específico relacionado con métodos.

```
import crypto from 'crypto';

const SEM_VERSION_REGEX = '^((\\d+\\.\\.?)?(\\d+\\.\\.?)?(\\d+\\.?)?)$';

export const isValidVersion = (version: string) => {
  return version.match(SEM_VERSION_REGEX);
};

/**
 * kebab-case to UpperCamelCase
 * @param {String} word
 * @return {String}
 */
export const toCamelCase = (word: string): string =>
  word.replace(/-./g, w => w[1].toUpperCase());

export const transformHeadersToCamelCase = (headers: {
  [k: string]: string | string[] | undefined;
}): { [k: string]: string | string[] } =>
  Object.entries(headers).reduce((r, [k, v]) => {
    return {
      ...r,
      [toCamelCase(k)]: v,
    };
  }, {});

export const hashKeyFn = <T>(data: T): string => {
  const objectToHash = JSON.stringify(data);
  return crypto.createHash('md5').update(objectToHash).digest('hex');
};

export const urlParams = (base: string, objectParams: {
  [k: string]: string;
}): string => {
  const params = new URLSearchParams(objectParams).toString();
  return base ? `${base}?${params}` : null;
}
```



Pilares del POO	
Abstracción	Las funciones isValidVersion, toCamelCase, transformHeadersToCamelCase, hashKeyFn y urlParams encapsulan lógica específica para realizar tareas bien definidas, ocultando la implementación interna a los usuarios. Esto promueve la abstracción al permitir que los usuarios interactúen con las funciones sin necesidad de conocer los detalles de su funcionamiento interno.
Encapsulamiento	Al no exponer propiedades o métodos privados, las funciones solo exponen interfaces públicas bien definidas, limitando el acceso a la implementación interna y promoviendo el encapsulamiento.
Polimorfismo	No se observa un uso explícito de polimorfismo en este código. Sin embargo, las funciones podrían ser utilizadas de forma polimórfica si se las pasa como argumentos a otras funciones o se las implementa en clases base y se las sobrescriben en clases derivadas.
Herencia	No se observa un uso explícito de herencia dentro del código.

Principios SOLID	
Single Responsibility	Cada función cumple con el SRP al tener una única responsabilidad bien definida.
Principio de Dependencia	Las funciones no dependen de implementaciones concretas, sino que reciben interfaces o tipos abstractos como parámetros. Esto facilita la prueba y la sustitución de dependencias.
Segregación de Interfaz	No se observa una violación evidente del ISP en este código. Las interfaces no parecen ser demasiado grandes o complejas.
Principio de "Open/Close"	Las funciones están abiertas a la extensión (por ejemplo, se pueden agregar nuevos parámetros) pero cerradas a la modificación (no se necesita cambiar la implementación interna).
Principio de Liskov	No se observa una violación evidente del LSP en este código. Las funciones parecen comportarse de manera consistente cuando se les pasan subtipos compatibles.

Patrón/es de Diseño	
No hay	<p>El código no implementa un patrón de diseño explícito con métodos. Sin embargo, se pueden observar algunas similitudes con patrones comunes:</p> <p>Función de utilidad: Las funciones individuales actúan como funciones de utilidad que realizan tareas comunes y reutilizables.</p> <p>Transformador: La función transformHeadersToCamelCase transforma un objeto de encabezados de kebab-case a UpperCamelCase, lo que se asemeja a un patrón de transformador.</p>

Clean Architecture

No hay exactamente


No se observa una implementación explícita de la arquitectura limpia en este código. Sin embargo, las funciones individuales exhiben algunas características consistentes con los principios de la arquitectura limpia:

-Responsabilidad única: Cada función tiene una responsabilidad bien definida y realiza una sola tarea específica.

-Dependencias abstractas: Las funciones no dependen de implementaciones concretas, sino que reciben interfaces o tipos abstractos como parámetros.

-Aislamiento de alto nivel: Las funciones no dependen de otras funciones dentro del mismo módulo, lo que facilita su prueba y reutilización independiente.


```
export interface IAuthService {
  generateToken(data: any): any;
  verifyToken(token: string): any;
  matchPassword(password, hash);
  hashPassword(plainPassword): string;
}
```



Pilares del POO

Abstracción	La interfaz IAuthService define las funcionalidades del servicio de autenticación sin revelar la implementación. Esto permite a los clientes usar el servicio sin conocer los detalles internos. Las clases que implementan la interfaz IAuthService pueden encapsular la lógica de autenticación específica.
Polimorfismo	En este fragmento de código no se observa polimorfismo explícito. Sin embargo, la interfaz Event podría servir como base para la creación de clases derivadas que implementen diferentes comportamientos o atributos relacionados con eventos específicos.

Clean Architecture:

No hay	No hay evidencia de una arquitectura limpia en el código proporcionado. La arquitectura limpia separa las entidades de negocio y los casos de uso.
--------	--

Principios SOLID

Single Responsibility	La interfaz IAuthService parece cumplir con SRP al definir una única responsabilidad: la autenticación de usuarios.
Segregación de Interfaz	La interfaz IAuthService parece cumplir con ISP al definir métodos específicos para cada tarea de autenticación (generar token, verificar token, etc.).

Patrón/es de Diseño

No hay, ni los muestra	En si se podrían implementar patrones de diseño como el patrón Strategy para encapsular las diferentes estrategias de autenticación (p. ej., autenticación con tokens, autenticación básica) o el patrón Facade para simplificar el acceso a las funcionalidades del servicio de autenticación.
------------------------	---

auth-service.ts

```
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';

import { config } from '@example-api/platform/index';
import { IAuthService } from './AuthService';
import { injectable } from 'inversify';
@injectable()
export class AuthService implements IAuthService {
  // ...
}
```



Pilares del POO

Abstracción	<p>El código define una interfaz IAuthService que especifica las operaciones que un servicio de autenticación debe proporcionar. Esto permite abstraer la implementación de la autenticación y facilita el uso del servicio en diferentes partes de la aplicación.</p> <p>La clase AuthService implementa la interfaz IAuthService, ocultando los detalles de la implementación de la autenticación a los usuarios del servicio.</p>
Polimorfismo	<p>El método generateToken no está especificado en la interfaz IAuthService, lo que permite que la clase AuthService lo implemente de manera específica a sus necesidades.</p> <p>La posibilidad de inyectar dependencias en el constructor de AuthService permite utilizar diferentes implementaciones del servicio de autenticación en diferentes contextos.</p>
Encapsulamiento	<p>La clase AuthService utiliza modificadores de acceso (private) para restringir el acceso a propiedades y métodos internos, protegiendo el estado interno del servicio.</p> <p>El uso de métodos getter y setter para propiedades como con g.auth.secret permite controlar el acceso y la mutabilidad de datos sensibles.</p>

Clean Architecture:

Sí	<p>Separa las responsabilidades en clases distintas (AuthService) y define una interfaz (IAuthService) para abstraer la implementación.</p>
----	---

Patrón/es de Diseño

No hay, ni los muestra	<p>Sin embargo, la estructura del código y el uso de inyección de dependencias sugieren que podría seguir un patrón de diseño basado en inyección de dependencias, como el patrón de servicio.</p>
------------------------	--

Principios SOLID	
Single Responsibility	La clase AuthService parece cumplir con este principio al centrarse únicamente en las responsabilidades relacionadas con la autenticación (generación y verificación de tokens, gestión de contraseñas).
Segregación de Interfaz	La interfaz IAuthService es relativamente simple y define solo las operaciones esenciales para la autenticación. Esto ayuda a evitar interfaces voluminosas y promueve la separación de preocupaciones.
Principio de Dependencia	El uso de inyección de dependencias a través de inversify permite que AuthService no dependa de la creación de instancias de sus dependencias (como config), lo que facilita la prueba y la flexibilidad del código.
Principio de "Open/Close"	Las clases parecen estar diseñadas de manera que se puedan agregar nuevas funcionalidades sin necesidad de modificar las clases existentes. Por ejemplo, se podrían agregar nuevos métodos para la autenticación con redes sociales sin modificar la lógica de generación y verificación de tokens.

CacheService.ts

```
export interface CacheService {
  getByKey<T>(key: string): Promise<T | null>;
  setByKey<T>(
    key: string,
    value: T,
    expireTimeInSeconds?: number
  ): Promise<void>;
  generateFunctionKey<T>(functionName: string, args?: T): string;
```

Pilares del POO

Abstracción	<p>La interfaz CacheService define un conjunto de métodos abstractos que representan las operaciones básicas de un servicio de caché, como getByKey, setByKey, deleteByKey, getKeyTLL y generateFunctionKey.</p> <p>Esta interfaz permite definir diferentes implementaciones del servicio de caché (por ejemplo, en memoria, en base de datos) sin exponer los detalles de implementación a los usuarios.</p>
Polimorfismo	<p>Las diferentes implementaciones del servicio de caché (en memoria, en base de datos) pueden implementar los métodos de la interfaz CacheService de manera diferente.</p> <p>Esto permite a los usuarios elegir la implementación de caché más adecuada para sus necesidades sin tener que modificar su código.</p>
Encapsulamiento	<p>Los métodos de la interfaz CacheService solo exponen la firma de los métodos (nombre, parámetros, tipo de retorno) sin revelar la implementación interna.</p> <p>Esto permite ocultar los detalles de cómo se almacena y recupera la información en caché, promoviendo la modularidad y la reutilización del código.</p>

Patrón/es de Diseño

Si	<p>Se podría implementar el patrón de diseño Memoization utilizando el método memoize de la interfaz CacheService.</p> <p>El patrón Memoization permite almacenar los resultados de funciones costosas para evitar recalcularlos en llamadas posteriores con los mismos parámetros.</p>
----	---

Principios SOLID

Single Responsibility	<p>La interfaz CacheService define un conjunto de métodos relacionados con la gestión de caché, lo que cumple con el principio SRP.</p> <p>Cada método tiene una única responsabilidad bien definida (obtener, almacenar, eliminar, etc.)</p>
Segregación de Interfaz	<p>Solo incluye los métodos esenciales para las operaciones básicas de caché, evitando la complejidad innecesaria.</p>
Principio de Dependencia	<p>Uso de inyección de dependencias y la adaptación a diferentes implementaciones de caché.</p>
Principio de "Open/Close"	<p>La interfaz CacheService se define de manera que se puedan agregar nuevos métodos sin modificar las implementaciones existentes.</p> <p>Esto permite extender la funcionalidad del servicio de caché sin afectar el código existente.</p>

CacheService.ts

```
export * from './redis';  
export * from './sql';
```

Redis.ts

```
import Redis from "ioredis";  
import config from "../../platform";  
  
let redisClient;
```

sql.ts

-Segunda carpeta(config)

```
const { Client } = require("pg");  
import config from "../../platform";  
  
export declare type DBSQLBind = string;
```

Pilares del POO

Encapsulamiento

Aplicación parcial.
Las variables redisClient y client en redis.ts y sql.ts son privadas dentro de sus respectivos módulos debido a la falta de declaración fuera de los archivos. Sin embargo, las funciones como getRedisClient y sql acceden directamente a ellas, lo que debilita la encapsulación.

Principios SOLID

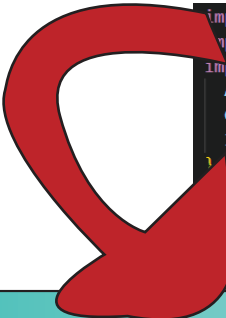
Single Responsibility

Posiblemente aplicado. Cada archivo (redis.ts y sql.ts) se centra en una única responsabilidad: conectarse a un tipo específico de base de datos (Redis o SQL).

Clean Architecture:

Sí

No es directamente evidente. La arquitectura limpia aboga por separar la lógica de negocio de las preocupaciones de la infraestructura. Si bien el código interactúa con bases de datos, no muestra una separación clara de las posibles capas de lógica de negocio.



```
import { Container } from 'inversify';
import { SYMBOLS } from '@example-api/config/symbols';
import {
  AuthService,
  CacheService,
  IAuthService,
} from '@example-api/common';
import { BaseController } from '../controllers/base-controller';
import {
  UserSignupController,
  UserSignInController,
}
```

Pilares del POO	
Encapsulamiento	<p>El encapsulamiento se implementa utilizando la modificación de acceso private para las propiedades y métodos de las clases. Esto restringe el acceso a los miembros de la clase solo a la propia clase o a sus clases hijas, protegiendo los datos internos y promoviendo la modularidad.</p> <p>Se observa el uso de interfaces para definir los contratos de las clases, lo que permite desacoplar la implementación de las clases de su uso, promoviendo la modularidad y la testabilidad.</p>
Abstracción	<p>Se observa la abstracción en la definición de interfaces como IAuthService y UserRepository. Estas interfaces definen las firmas de los métodos que deben implementar las clases concretas, sin especificar cómo se implementan esos métodos.</p> <p>También se aplica la abstracción en el uso de clases abstractas como BaseController. Esta clase abstracta define una estructura básica para los controladores, permitiendo la reutilización de código y la definición de comportamientos comunes a todos los controladores.</p>
Polimorfismo	<p>Se aplica en el uso de inyección de dependencias para obtener instancias de las clases concretas a través de interfaces. Esto permite intercambiar implementaciones diferentes sin modificar el código que las utiliza, promoviendo la flexibilidad y la adaptabilidad.</p> <p>Se observa el uso de clases abstractas como BaseController para definir métodos que serán implementados por sus clases hijas de manera específica, permitiendo un comportamiento polimórfico.</p>
Herencia	<p>La herencia se utiliza en la definición de clases que heredan de clases abstractas como BaseController. Esto permite reutilizar código y compartir comportamientos comunes entre las clases relacionadas.</p> <p>Se observa la herencia en la definición de la clase UserRepositoryImpl que hereda de la interfaz UserRepository. Esto permite implementar la interfaz de manera específica para la clase concreta.</p>

Clean Architecture:	
Sí	<p>-Se observa la estructura de capas en el código, con una capa de dominio que define los casos de uso y entidades, una capa de datos que interactúa con el almacenamiento, y una capa de presentación que maneja la interacción con el usuario.</p> <p>-Se utiliza inyección de dependencias para desacoplar las capas, permitiendo que cada capa se pueda probar y desarrollar de manera independiente.</p> <p>-Se observan interfaces de nichos para cada capa, lo que permite la separación de preocupaciones y la sustitución de implementaciones concretas sin afectar el resto del código.</p>

Principios SOLID	
Single Responsibility	Cada clase parece tener una única responsabilidad bien definida. Por ejemplo, la clase UserRepository es responsable de interactuar con el almacenamiento de usuarios, y la clase SignupUseCase es responsable de la lógica de registro de usuarios.
Principio de Dependencia	Se utiliza inyección de dependencias para desacoplar las clases de sus dependencias. Esto permite que las clases se prueben con dependencias simuladas y facilita la sustitución de implementaciones concretas.
Segregación de Interfaz	Las interfaces parecen ser específicas para las necesidades de las clases que las implementan. Por ejemplo, la interfaz UserRepository solo define los métodos necesarios para interactuar con el almacenamiento de usuarios, y la interfaz IAuthService define los métodos necesarios para la autenticación de usuarios.
Principio de "Open/Close"	Las clases parecen estar diseñadas para ser abiertas para la extensión pero cerradas para la modificación. Por ejemplo, se podrían agregar nuevos métodos a las interfaces sin necesidad de modificar las clases que las implementan.
Principio de Liskov	Las clases hijas parecen ser sustituibles por sus clases padre sin causar efectos secundarios. Por ejemplo, se espera que una instancia de UserRepositoryImpl pueda ser utilizada en cualquier lugar donde se espera una instancia de UserRepository sin afectar el comportamiento del código.

Patrón/es de Diseño	
No hay, ni los muestra	<p>Sin embargo, se pueden identificar algunos patrones generales que se utilizan:</p> <p>Patrón de inyección de dependencias: Se utiliza para desacoplar las clases de sus dependencias y permitir la sustitución de implementaciones concretas.</p> <p>Patrón de capa de cebolla: Se organiza el código en capas concéntricas con responsabilidades bien definidas.</p> <p>Patrón de interfaz: Se definen interfaces para separar las preocupaciones y permitir la sustitución de implementaciones concretas.</p>

index.ts

```
export const SYMBOLS = {  
  CacheService: Symbol("CacheService"),  
  AuthService: Symbol("AuthService"),  
  ScoreIntentUseCase: Symbol("ScoreIntentUseCase"),  
  ScoreTopUseCase: Symbol("ScoreTopUseCase"),  
  InitGameUseCase: Symbol("InitGameUseCase"),  
  ScoreStatsUseCase: Symbol("ScoreStatsUseCase"),  
  SignupUseCase: Symbol("SignupUseCase"),  
  SigninUseCase: Symbol("SigninUseCase")  
}
```

Pilares del POO

Abstracción

El código utiliza interfaces para definir los contratos de los servicios y repositorios, lo que permite abstraer la implementación y enfocarse en el uso. ([Ejemplo: SYMBOLS de ne interfaces para servicios y repositorios])

Principios SOLID

Principio de Dependencia

El uso de interfaces (simbolizadas en SYMBOLS) podría indicar una posible aplicación del DIP, ya que permite la inyección de dependencias y desacopla las clases concretas de sus abstracciones.



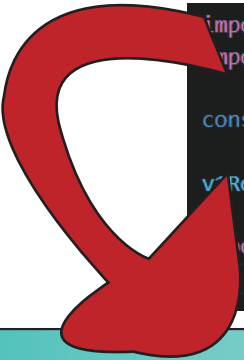
```
export interface IJsonObject {
  [key: string]: string;
}

export interface ICustomError extends Error {
  code: string;
  vars?: IJsonObject;
  httpCode: number
}
```

```
1 import { ICustomError, IJsonObject } from '
2 import { IGetEntityOlimpoRepository } from '
3 import { IRequestOlimpoRepository } from '
4
5 export {
6   ICustomError,
7   IJsonObject,
8   IGetEntityOlimpoRepository,
9   IRequestOlimpoRepository,
10 };
11 |
```

Pilares del POO	
Abstracción	Las interfaces ICustomError e IJsonObject definen la estructura de datos para los errores personalizados y objetos JSON, respectivamente. Estas interfaces establecen un contrato para las propiedades y tipos de datos esperados, permitiendo la abstracción de la implementación específica.
Encapsulamiento	Modificadores de acceso: No se observan modificadores de acceso (public, private, protected) en los archivos proporcionados. La encapsulación se podría implementar utilizando estos modificadores para controlar la visibilidad de propiedades y métodos dentro de las clases.

Principios SOLID	
Segregación de Interfaz	Incorpora la interfaz necesaria, que es breve y a su vez permite separar las tareas de la clase principal en este archivo.



```
import { Router } from 'express';
import { userRouter } from './user';

const v1Routes = Router();

v1Routes.use("/v1/user", userRouter);

export { v1Routes };
```

```
import { Router } from "express";
import { container } from "@example";
import {
  UserSignupController,
  UserSigninController,
} from "@example-api/controllers";

const signup = container.get(UserSi
const signin = container.get(UserSi
```

Pilares del POO

Abstracción

Se observa abstracción en la definición de las rutas de la API, utilizando el módulo Router de Express. Esto permite encapsular la lógica de las rutas y exponer una interfaz simplificada para su uso.

Las clases UserSignupController y UserSigninController representan abstracciones de la lógica de negocio relacionada con el registro y el inicio de sesión de usuarios, respectivamente.

Encapsulamiento

La lógica de las rutas y la lógica de negocio se encuentran encapsuladas en sus respectivos módulos (index.ts y user.ts).

Los detalles de implementación de las clases UserSignupController y UserSigninController se mantienen ocultos al resto del código, promoviendo la modularidad y la reutilización.

Clean Architecture:

Sí

-Estructura de capas:

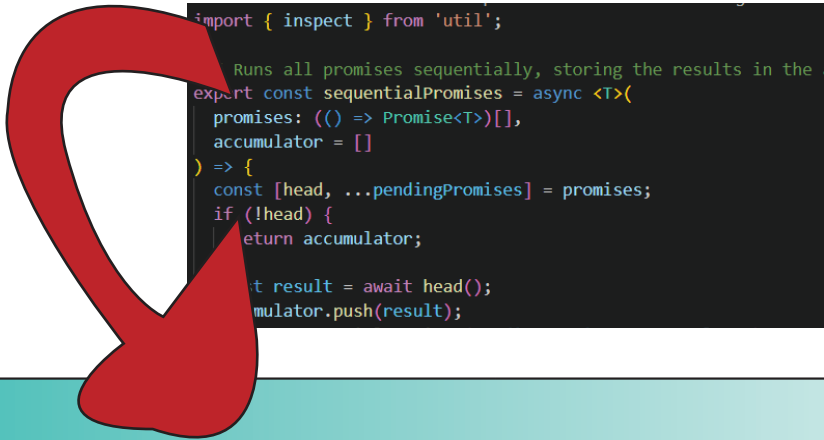
Se observa una separación básica entre las capas de red (rutas de API) y negocio (controladores).

-Dependencias:

Las dependencias de las clases controladoras se gestionan mediante inyección de dependencias a través de Inversify.

Principios SOLID	
Principio de Dependencia	La inyección de dependencias a través de Inversify implementa el principio DIP, permitiendo que las clases controladoras no dependan de las implementaciones concretas de los servicios, sino de abstracciones.
Principio de "Open/Close"	El código parece seguir el principio OCP al permitir la modificación de las clases controladoras sin afectar a las rutas de la API o a otras partes del sistema.
Single Responsibility	Las clases UserSignupController y UserSigninController parecen cumplir con SRP al tener una única responsabilidad: manejar el registro y el inicio de sesión de usuarios, respectivamente.

Patrón/es de Diseño	
Sí, Patrón de Comando	Podría ser aplicable al código proporcionado, ya que las clases controladoras encapsulan la lógica de negocio para el registro y el inicio de sesión de usuarios, y se invocan a través de las rutas de la API.



Pilares del POO	
Abstracción	Aunque no se implementa directamente en estas funciones, se promueve la abstracción al definir interfaces (executeArrayPromisesOptions) y usar genéricos (sequentialPromises, executeArrayPromises). Esto permite flexibilidad y reutilización del código sin detalles concretos de implementación.

Principios SOLID	
Principio de "Open/Close"	Se demuestran sus aspectos como lo es: -Agregar lógica de manejo de promesas más compleja, sin modificar el código existente Aún así es posible que el diseño actual no esté completamente cerrado a modificaciones si surge la necesidad de cambios significativos en el comportamiento.
Single Responsibility	Sus funciones se adhieren al principio: -sequentialPromises: Ejecuta promesas secuencialmente, acumulando resultados. -memoizePromise: Crea una versión memorizada de una función asíncrona. -executeArrayPromises: Ejecuta una matriz de promesas, maneja errores y devuelve resultados exitosos.