Servicios y Mock api

...

resumen mock api en base a https://mockapi.io
resumen de servicios con axios https://axios-http.com

Mockapi

Generar un nuevo proyecto

Generar un nuevo recurso

Guía: https://github.com/mockapi-io/docs/wiki/Quick-start-guide

Axios

Instalación de axios: npm install axios

Uso:

```
import axios from ‘axios”

async cargarLista() {
    try {
      const response = await axios.get("https://645aeb4e65bd868e9326bfdd.mockapi.io/api/v1/lista");
      this.lista = response.data
      console.log(this.lista);
    } catch (error) {
      console.error(error);
    }
 },
```

Axios, ejemplo post

```
async agregar() {
  try {
    const elem = { ...this.elemento }
    await axios.post("https://645aeb4e65bd868e9326bfdd.mockapi.io/api/v1/lista",elem)
  } catch (error) {
    console.log(error);
  }
 },
```

Axios, ejemplo delete

```
async eliminar(id) {
  try {
    await axios.delete("https://645aeb4e65bd868e9326bfdd.mockapi.io/api/v1/lista/" + id)
    await this.cargarLista()
  } catch ( error) {
    alert(' Se produjo un error')
  }
},
```

Axios, ejemplo Modificar

```
async modificar(id) {
  try {
    const elem = { ...this.elemento }
    await axios.put("https://645aeb4e65bd868e9326bfdd.mockapi.io/api/v1/lista/" + id, elem)
    await this.cargarLista()
  } catch ( error) {
    alert(' Se produjo un error')
  }
 },
```

Principio Solid

En programación, el principio SOLID es un conjunto de principios de diseño que se utilizan para desarrollar software de alta calidad y mantenible. Estos principios fueron acuñados por Robert C. Martin, también conocido como "Uncle Bob", un reconocido experto en desarrollo de software.
SOLID es un acrónimo que representa los cinco principios individuales:
Principio de Responsabilidad Única (Single Responsibility Principle, SRP): Este principio establece que una clase o módulo debe tener una sola razón para cambiar. En otras palabras, una clase debe tener una única responsabilidad o función en el sistema.
Principio de Abierto/Cerrado (Open/Closed Principle, OCP): Según este principio, una entidad de software debe estar abierta para su extensión pero cerrada para su modificación. Esto implica que se deben poder agregar nuevas funcionalidades sin necesidad de modificar el código existente.
Principio de Sustitución de Liskov (Liskov Substitution Principle, LSP): Este principio establece que los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin causar problemas en el programa. En resumen, las clases derivadas deben ser completamente sustituibles por sus clases base.
Principio de Segregación de la Interfaz (Interface Segregation Principle, ISP): Este principio establece que los clientes no deben depender de interfaces que no utilizan. En lugar de tener interfaces monolíticas, se deben crear interfaces específicas para cada cliente, evitando la dependencia de métodos no utilizados
Principio de Inversión de Dependencia (Dependency Inversion Principle, DIP): Según este principio, los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

Principio DRY

El principio DRY (Don't Repeat Yourself) es otro principio importante en programación. Fue acuñado por Andy Hunt y Dave Thomas en su libro "The Pragmatic Programmer" (El programador pragmático). El principio DRY se enfoca en la eliminación de la duplicación de código en un sistema de software.

El principio DRY establece que cada pieza de conocimiento o lógica en un programa debe tener una única representación autoritaria y clara en todo el sistema. En lugar de duplicar el mismo código en múltiples lugares, se debe crear una abstracción o función reutilizable que encapsule ese comportamiento repetido. De esta manera, si surge la necesidad de realizar cambios, solo se requiere modificar un único lugar en lugar de múltiples puntos dispersos en el código.

El objetivo del principio DRY es reducir la duplicación de código, lo que puede llevar a problemas de mantenimiento, inconsistencias y dificultades para hacer cambios en el sistema. Al seguir este principio, se busca mejorar la legibilidad, la reutilización y la modularidad del código, lo que a su vez puede facilitar su mantenimiento y evolución.

En resumen, el principio DRY promueve la escritura de código conciso y modular, evitando la duplicación innecesaria y fomentando la reutilización de código a través de abstracciones y funciones bien definidas.

Resumen de 10 puntos: libro Arquitectura Limpia de Robert Martin

1. La arquitectura limpia promueve la separación de preocupaciones, dividiendo el sistema en capas que representan diferentes responsabilidades y niveles de abstracción.
2. La independencia de las capas es esencial para permitir cambios en una capa sin afectar a las demás, lo que mejora la flexibilidad y la mantenibilidad del sistema.
3. El principio de inversión de dependencia es fundamental en la arquitectura limpia, donde las dependencias se definen en términos de abstracciones y no de implementaciones concretas.
4. Los casos de uso (use cases) son el punto de partida para diseñar la arquitectura. Se deben identificar y modelar los diferentes casos de uso del sistema antes de definir las capas y componentes.
5. La arquitectura limpia promueve la separación de la lógica de negocio de los detalles de implementación técnica, como el framework o la base de datos utilizados.
6. La modularidad es clave para una arquitectura limpia, donde los componentes deben ser independientes y tener responsabilidades únicas y bien definidas.
7. Los principios SOLID, incluido el principio de responsabilidad única (SRP), son fundamentales para la arquitectura limpia, ya que ayudan a crear componentes cohesivos y fáciles de mantener.
8. Las pruebas automatizadas, como las pruebas unitarias, son esenciales para garantizar la calidad y la integridad del sistema. La arquitectura limpia fomenta el diseño de componentes que sean fácilmente probables y aislables.
9. La comunicación entre capas debe hacerse a través de interfaces bien definidas y no mediante acoplamientos directos, lo que facilita el cambio de implementación sin afectar a otras partes del sistema.
10. La arquitectura limpia no es un objetivo en sí mismo, sino un enfoque que busca maximizar la calidad del software, la flexibilidad, la mantenibilidad y la escalabilidad a lo largo del tiempo.

Ejemplo servicios

Creamos una carpeta service y un archivo listaService.js , donde creamos un objeto para la conexión con la api. Configuración:

```
import axios from 'axios'
const apiClient = axios.create({
    baseURL: 'https://645aeb4e65bd868e9326bfdd.mockapi.io/api/v1',
    headers: {
        Accept: 'application/json',
        'Content-Type': 'application/json'
    }
})
```

Ejemplo servicios, métodos

```
export default {
    async cargar() {
        try {
            const response = await apiClient.get('/lista');
            return response.data
        } catch (error) {
            throw "Error de conexion"
        }
    },
    async agregar(elem) {
        try {
            await apiClient.post('/lista', elem);
        } catch (error) {
            throw "Error de conexion"
        }
    },
}
```

Ejemplo servicios, métodos

```
async eliminar(id) {
    try {
        await apiClient.delete("/lista/" + id)
    } catch (error) {
        throw "Error de conexion"
    }
 },
async modificar(id, elem) {
    try {
        await apiClient.put("/lista/" + id, elem)
    } catch (error) {
        throw "Error de conexion"
    }
 },
```