

NODE y TypeScript

Factory functions para inyección de dependencias en Vanilla JavaScript

Definición o concepto: Una "factory function" es una función que crea y devuelve objetos. Cuando hablamos de inyección de dependencias (DI) en JavaScript, normalmente se busca una forma de pasar las dependencias a un módulo o función sin que éste tenga que crearlas directamente. Esto facilita el testeo, la reutilización y la configuración flexible del código.

Explicación de uso: En un entorno sin frameworks, puedes crear una función que reciba las dependencias necesarias para crear un objeto o módulo, y luego devolver ese objeto totalmente configurado. Por ejemplo, en lugar de que un módulo cree su propia instancia de una librería, se la puedes inyectar desde afuera.

Ejemplo:

```
// servicioDeUsuarios.js
export function crearServicioDeUsuarios({ httpClient }) {
  return {
    async obtenerUsuarios() {
      const respuesta = await httpClient.get('/users');
      return respuesta.data;
    }
  };
}
```

En este ejemplo, httpClient se inyecta. Así, crearServicioDeUsuarios no se preocupa por crear su propio cliente HTTP. Podrás pasarle Axios u otro cliente de prueba en tests:

```
// ejemplo de uso
import axios from 'axios';
import { crearServicioDeUsuarios } from './servicioDeUsuarios.js';

const servicio = crearServicioDeUsuarios({ httpClient: axios });
servicio.obtenerUsuarios().then(usuarios => console.log(usuarios));
```

Esto hace que el servicio sea fácilmente testeable, ya que en tests podrías hacer:

```
const httpMock = {
  get: async () => ({ data: [{ id: 1, nombre: 'Fabricio' }]});
};

const servicioTest = crearServicioDeUsuarios({ httpClient: httpMock });
const usuarios = await servicioTest.obtenerUsuarios();
console.log(usuarios); // [{ id: 1, nombre: 'Fabricio' }]
```

Axios

Definición o concepto: Axios es una librería popular para realizar solicitudes HTTP desde Node.js o el navegador. Está construida sobre promesas y simplifica el envío de peticiones GET, POST, PUT, DELETE, así como el manejo de cabeceras, transformaciones de datos y manejo de errores.

Instalación:

```
User@DESKTOP-K0C1M9L MINGW64 ~/prueba-ts
$ npm install axios
```

Uso básico:

```
import axios from 'axios';

axios.get('https://api.ejemplo.com/data')
  .then(respuesta => {
    console.log(respuesta.data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

```
import {default as axios} from 'axios';

axios.get('https://pokeapi.co/api/v2/pokemon/pikachu')
  .then(respuesta => {
    console.log(respuesta.data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

O con async/await:

```
import {default as axios} from 'axios';

try {
  const respuesta = await axios.get('https://pokeapi.co/api/v2/pokemon/pikachu');
  console.log(respuesta.data);
} catch (err) {
  console.error(err);
}
```

Interceptores de Axios

Definición o concepto: Los interceptores de Axios permiten interceptar las solicitudes y respuestas antes de que lleguen a tu código principal. Esto es útil para:

- Agregar tokens de autenticación a todas las peticiones.
- Manejar errores globalmente.
- Transformar respuestas o solicitudes estandarizando su formato.

Uso de interceptores:

```
import {default as axios} from 'axios';

const instancia = axios.create({ baseURL: 'https://pokeapi.co/api/v2/pokemon/pikachu' });

// Interceptor de solicitud
instancia.interceptors.request.use(config => {
  // Añadir token de autenticación
  config.headers['Authorization'] = 'Bearer mi-token-secreto';
  return config;
}, error => Promise.reject(error));

// Interceptor de respuesta
instancia.interceptors.response.use(response => {
  // Transformar la respuesta
  return response.data;
}, error => {
  // Manejar errores globalmente
  console.error('Error en la respuesta:', error);
  return Promise.reject(error);
});

// Uso de la instancia con interceptores
const datos = await instancia.get('https://pokeapi.co/api/v2/pokemon/raichu');
console.log(datos);
```

Con estos interceptores, cada petición y respuesta pasan por este pipeline, facilitando la lógica transversal a todas las llamadas HTTP.

TypeScript, Interfaces, Tipos y Clases

Definición o concepto: TypeScript es un superset tipado de JavaScript que aporta tipos estáticos opcionales al lenguaje. Esto permite encontrar errores en tiempo de compilación, auto-completado más inteligente, y una mayor robustez al desarrollar aplicaciones complejas.

- **Tipos (Types):** Definen la forma de datos primitivos o complejos, por ejemplo, string, number, boolean, o tipos personalizados con type.
- **Interfaces:** Describen la forma de un objeto, definiendo qué propiedades y métodos contiene.
- **Clases:** TypeScript añade tipado a las clases, permitiendo definir tipos para propiedades, métodos y constructores. Además, soporta modificadores de acceso (public, private, protected).

Ejemplo de tipos e interfaces:

```
type ID = number | string; // Un tipo ID que puede ser número o string

interface Usuario {
  id: ID;
  nombre: string;
  edad?: number; // propiedad opcional
}

Explain
function saludar(usuario: Usuario): void {
  console.log(`Hola, ${usuario.nombre}`);
}
```

En este ejemplo:

- ID es un tipo que puede ser number o string.
- Usuario es una interfaz que describe un objeto con id, nombre y una edad opcional.
- La función saludar recibe un Usuario.

Ejemplo de clases con TypeScript:

```
type ID = number
Explain
class Empleado {
  private id: ID;
  public nombre: string;
  protected sueldo: number;

  constructor(id: ID, nombre: string, sueldo: number) {
    this.id = id;
    this.nombre = nombre;
    this.sueldo = sueldo;
  }

  public subirSueldazo(aumento: number): void {
    this.sueldo += aumento;
  }

  public getSueldoAhora(): number {
    return this.sueldo;
  }
}

const emp = new Empleado(1, 'Fabricio', 3_500_000);
emp.subirSueldazo(600_000);
console.log(emp.getSueldoAhora()); // 4100000
```


Ejecutando:

```
User@DESKTOP-K0C1M9L MINGW64 ~/prueba-ts
$ npm install --save-dev ts-node typescript
```

```
$ cat package.json
{
  "name": "prueba-ts",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "subirsue1do": "ts-node empleado.ts",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "devDependencies": {
    "typescript": "^3.9.7",
    "ts-node": "^8.10.2"
  }
}
```

```
User@DESKTOP-K0C1M9L MINGW64 ~/prueba-ts
$ npm run subirsue1do

> prueba-ts@1.0.0 subirsue1do
> ts-node empleado.ts

4100000

User@DESKTOP-K0C1M9L MINGW64 ~/prueba-ts
```

Aquí:

- Se usan modificadores de acceso (private, public, protected).
- Se aplica un constructor tipado.
- Se verifica en tiempo de compilación que emp cumple con el tipo Empleado.

Integración con Node.js:

- En Node.js, TypeScript se compila a JavaScript usando tsc.
- Puedes usar TypeScript para tus módulos, servicios y controladores, garantizando mayor robustez.

- Si combinas TypeScript con factory functions, podrás describir las interfaces de las dependencias con mayor claridad.

Ejemplo integrando Axios, Factory functions y TypeScript:

```
// tipo para la dependencia http
interface HttpClient { get(url: string): Promise<any>; }

interface UsuarioService { obtenerUsuarios(): Promise<Empleado[]>; }

function crearUsuarioService(httpClient: HttpClient): UsuarioService {
  return {
    async obtenerUsuarios() {
      const respuesta = await httpClient.get('/users');
      return respuesta as Empleado[];
    }
  };
}

const consultaMock:HttpClient = {
  get: async (url: string) => [{id: 1, nombre: "jose", sueldo: 45000}, {id:2, nombre: "pedro", sueldo:4450}]
}

const test = crearUsuarioService(consultaMock)
const users = test.obtenerUsuarios().then(er => {console.log(er)})
console.log(users)
```

```
User@DESKTOP-K0C1M9L MINGW64 ~/prueba-ts
$ npx ts-node axiosTs.ts
4100000
Promise { <pending> }
[
  { id: 1, nombre: 'jose', sueldo: 45000 },
  { id: 2, nombre: 'pedro', sueldo: 4450 }
]
```

Acá, HttpClient es una interfaz que describe la dependencia que crearUsuarioService requiere. Puedes pasar Axios u otra cosa que cumpla con HttpClient.