Patrones de diseño: Singleton

Luis Arturo Gonzalez Valencia

Universidad de la Cuenca del Plata – Ingeniería en sistemas – Ingeniería de Software II

Resumen

El patrón de diseño a desarrollar y definir que se tocará en este informe es el patrón Singleton.

Previamente a esto necesitamos conocer y entender ¿Qué es un patrón de diseño?

En el proceso de desarrollo de software, una de las etapas más importantes es la codificación o programación de este. Dicho proceso requiere resolver problemas que en muchos de los casos ya han sido resueltos de manera similar en el pasado por alguien más, por lo cual podríamos modelar nuestro software o componente de la misma manera que lo hicieron estos. Por lo cual, si la forma de solucionar un problema se puede extraer, explicar y reutilizar en múltiples ámbitos (llámese software o componente), entonces nos encontramos con un patrón de diseño.

Un patrón de diseño es una forma reutilizable de resolver un problema común.

Los patrones de diseños son importantes y los beneficios que otorgan son:

1. Ahorran tiempo: buscar siempre una nueva solución a los mismos problemas reduce la eficacia del desarrollar y alarga los procesos de desarrollo en el tiempo.
2. Seguridad en la validez del código: Los patrones de diseño son estructuras probadas por millones de desarrolladores a lo largo de muchos años.
3. Establecen un lenguaje común: permiten explicar a otras personas como se han resuelto los problemas durante el desarrollo.

Los patrones de diseño se dividen en distintos grupos según el tipo de problema que resuelvan. (Patrones creacionales, estructurales y de comportamiento).

Para nuestro estudio nos centraremos en el grupo de patrones creacionales al que pertenece Singleton.

Funcionamiento de Singleton

Siempre es importante tener un buen control de las instancias de clases que estemos usando en nuestro proyecto, pero ¿qué pasaría si pudiéramos limitar a una única instancia en toda la aplicación a ciertas clases de nuestro proyecto? A esta práctica se le conoce como patrón de diseño Singleton.

Singleton limita a uno el número de instancias posibles de una clase en nuestro programa y proporciona un acceso global al mismo.

Es útil cuando se necesita solo una instancia de una clase para poder coordinar acciones entre uno o varios sistemas.

¿Como darnos cuenta de que podemos usarlo? Cuando nuestra clase o código tiene lógica que nunca va a cambiar, que siempre va a ser la misma, por lo cual pierde el sentido de estar instanciándola constantemente.

¿Qué tipo de problemas resuelve Singleton y cómo se construye?

Performance

Al no generar una instancia por cada vez que se necesita utilizar la lógica o la información de una clase, se ahorra espacio en memoria lo que permite que el software funcione más rápido.

Ej.: Si tenemos en el backend una clase Calendario, y tenemos miles de usuario que quieren acceder a la misma, al generar una sola instancia del calendario, este se construirá una sola vez para todos, y cada uno estará haciendo uso de la misma instancia. Si en cambio cada uno solicita una nueva instancia del calendario, se estará construyendo un nuevo objeto por cada usuario, por cada petición y con una lógica que no cambia. Lo que reduce mucho la performance.

Construcción

Para mostrar su construcción lo haremos en typescript, usando un ejemplo de calendario para manejar el formateo de fechas y horas dentro de un proyecto.

1. Generar la clase con la instancia declarada y el constructor privado:

```typescript
export class Calendario {
    //instancia privada para manejar el singleton internamente
    private static instance: Calendario;


    //el constructor debe estar privado para no permitir generar instancia de la clase
    private constructor() {

    }

}
```

2. Generar un método estático (por convención getInstance()) que devuelva la instancia de la clase y realice la lógica del Singleton:

```typescript
export class Calendario {
    //instancia privada para manejar el singleton internamente
    private static instance: Calendario;


    //el constructor debe estar privado para no permitir generar instancia de la clase
    private constructor() {

    }



    //el metodo estatico permite acceder a la instancia sin necesidad de instanciarla
    public static getInstance(): Calendario {
        //si la instancia no existe, la creamos
        if (!Calendario.instance) {
            Calendario.instance = new Calendario();
        }
        //devolvemos la instancia de la clase
        return Calendario.instance;
    }

}
```

3. Generamos los métodos con la lógica de la clase:

```
export class Calendario {
    //instancia privada para manejar el singleton internamente
    private static instance: Calendario;

    //el constructor debe estar privado para no permitir generar instancia de la clase
    private constructor() {

    }

    //el metodo estatico permite acceder a la instancia sin necesidad de instanciarla
    public static getInstance(): Calendario {
        //si la instancia no existe, la creamos
        if (!Calendario.instance) {
            Calendario.instance = new Calendario();
        }
        //devolvemos la instancia de la clase
        return Calendario.instance;
    }


    //generamos los getters publicos que puden devolver valores de atributos privados o lógica (como en este caso)

    public getFecha(): string {
        let fecha = new Date();
        let dia = fecha.getDate();
        let mes = fecha.getMonth() + 1;
        let anio = fecha.getFullYear();

        return dia + "/" + mes + "/" + anio;
    }

    public getHora(): string {
        let fecha = new Date();
        let hora = fecha.getHours();
        let minutos = fecha.getMinutes();
        let segundos = fecha.getSeconds();

        return hora + ":" + minutos + ":" + segundos;
    }

    public getFechaHora(): string {
        return this.getFecha() + " " + this.getHora();
    }

    public getFechaHoraFormato(formato: string): string {
        let fecha = new Date();
        let dia = fecha.getDate();
        let mes = fecha.getMonth() + 1;
        let anio = fecha.getFullYear();
```

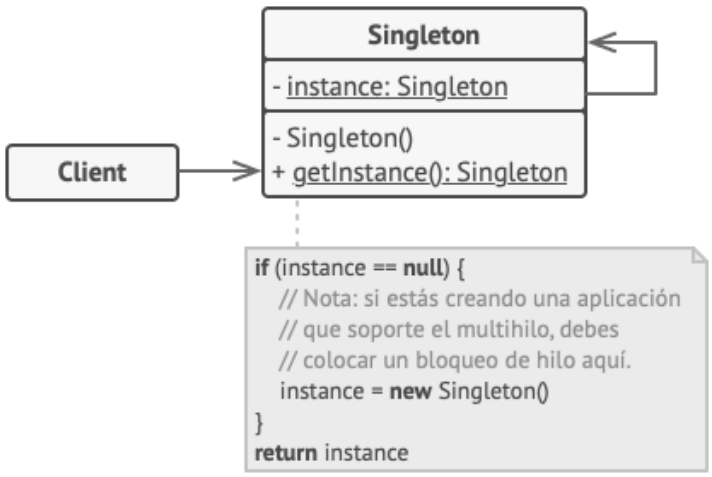
Test de nuestro Singleton:

```
import { Calendario } from "../CalendarioSingleton/Calendario";

//asignamos la variable calendario global a una instancia de la clase calendario
const calendario = Calendario.getInstance();

//este primer test es para verificar que, sin realizar una instancia de la clase externamente,
//la clase calendario maneja su instancia internamente. Si no existe la genera, y si ya existe, la devuelve.
test('El objeto no debe ser nulo', () => {
    expect(calendario !== null).toBeTruthy();
})


//este segundo test es para verificar que, al asignar a distintas variables la misma instancia, ellas son iguales.
// es decir, apuntan a la misma instancia (direccion de memoria) de la clase Calendario, y con ello se asegura que el singleton está funcionando.
test('Las variables del objeto deben ser iguales', () => {
    let calendario2 = Calendario.getInstance();
    expect(calendario === calendario2).toBeTruthy();
})
```

```
21  //cubrimos los métodos del singleton
22
23  test('La fecha debe ser la correcta en formato dia/mes/anio', () => {
24      let fecha = new Date();
25      let dia = fecha.getDate();
26      let mes = fecha.getMonth() + 1;
27      let anio = fecha.getFullYear();
28      let fechaStr = dia + "/" + mes + "/" + anio;
29      expect(calendario.getFecha()).toBe(fechaStr);
30  })
31
32  test('La hora debe ser la correcta en formato hora:minutos:segundos', () => {
33      let fecha = new Date();
34      let hora = fecha.getHours();
35      let minutos = fecha.getMinutes();
36      let segundos = fecha.getSeconds();
37      let horaStr = hora + ":" + minutos + ":" + segundos;
38      expect(calendario.getHora()).toBe(horaStr);
39  })
40
41
42  test('La fecha y hora debe ser la correcta en formato dia/mes/anio hora:minutos:segundos', () => {
43      let fecha = new Date();
44      let dia = fecha.getDate();
45      let mes = fecha.getMonth() + 1;
46      let anio = fecha.getFullYear();
47      let fechaStr = dia + "/" + mes + "/" + anio;
48      let hora = fecha.getHours();
49      let minutos = fecha.getMinutes();
50      let segundos = fecha.getSeconds();
51      let horaStr = hora + ":" + minutos + ":" + segundos;
52      let fechaHoraStr = fechaStr + " " + horaStr;
53      expect(calendario.getFechaHora()).toBe(fechaHoraStr);
54  })
55
56
57  test('La fecha debe ser la correcta en formato dia/mes personalizado', () => {
58      let fecha = new Date();
59      let dia = fecha.getDate();
60      let mes = fecha.getMonth() + 1;
61      let fechaStr = dia + "/" + mes;
62      expect(calendario.getFechaHoraFormato("dd/mm")).toBe(fechaStr);
63  })
```

Repositorio Github: https://github.com/luisgonzalezvalencia/Singleton-pattern

Diagrama de clases Singleton:



Cuando NO utilizarlo:

En muchos ejemplos en la web, nos encontraremos con la aplicación del patrón Singleton asociados a instancias de base de datos. Quiere decir que nuestro proyecto abre una instancia a la base de datos y ya la tiene disponible en toda nuestra aplicación. Esto muchas veces representa un “Anti patrón”, porque en una aplicación de alta concurrencia no es lo más recomendable tener una sola instancia de conexión a la base de datos ya que puede generar cuellos de botellas y reducir la performance.

Referencias

Cristian Henao. ¿Por qué usar el Patrón Singleton? - BD Remota en Android con Volley. (2017, 20 noviembre). [Vídeo]. YouTube. https://www.youtube.com/watch?v=qgKkV1EEqpY

KODOTI. (2021, 1 mayo). Patrones de diseño con TypeScript - Singleton [Vídeo]. YouTube. https://www.youtube.com/watch?v=Qckwuge7dpQ

Leiva, A. (2021, 24 junio). *Patrones de diseño de software*. DevExperto, por Antonio Leiva. https://devexperto.com/patrones-de-diseno-software/

sourcemaking. (s. f.). *Design Patterns and Refactoring*. Recuperado 15 de mayo de 2022, de https://sourcemaking.com/design_patterns/singleton