Angular - Clase 3

Servicios, Observables y HTTP

En esta clase veremos lo que son los servicios, cómo nos pueden ayudar en el desarrollo de nuestra aplicación y cómo crearlos.

También veremos qué es la inyección de dependencias y cómo inyectar un servicio a un componente.

Aprenderemos acerca de Observables, su uso y modificadores.
Finalmente veremos cómo usar el servicio HTTP de Angular en conjunto con los observables para realizar llamadas al Backend

Servicios

Los Servicios de Angular son un tipo de objeto en Angular que se utilizan para compartir datos y lógica entre componentes. Los servicios son una forma de separar la lógica de negocio de los componentes y promover la reutilización de código.

En Angular, los servicios se definen como clases que pueden tener propiedades, métodos y dependencias. Pueden ser inyectados en componentes u otros servicios a través de la inyección de dependencias.

Los servicios pueden ser utilizados para realizar operaciones de red, interactuar con APIs, almacenar datos en caché, gestionar el estado de la aplicación, entre otras tareas. Al separar la lógica de negocio en servicios, se mejora la modularidad y la capacidad de prueba de la aplicación.

En resumen, los servicios son una pieza clave de la arquitectura de Angular que permiten compartir datos y lógica entre componentes y promover la reutilización de código.

Servicios y sus usos

Normalmente al iniciar una aplicación tendemos a escribir la lógica de negocio en los componentes de las pantallas a mostrar. Esto debemos evitarlo en caso de ser cosas que necesitan ser compartidas en muchos lugares. Ej: el carrito de compras debe ser accesible desde Productos y la propia página de carritos, es por esto que la lista de productos es una buena idea guardarla en un “servicio de carrito” para que sea accesible desde muchos lugares y que no se repita la información.

Otro uso de los servicios es manejar estados. Supongamos que tenemos una página con Login, si un usuario está logueado, queremos mantener la información de sesión guardada en un solo lugar y que todos puedan acceder a ella.

También nos sirve para reutilizar lógica de programación. En caso de necesitar realizar un cálculo muy específico en un producto en específico es posible que debamos hacerlo en varios más. Dado el caso, podríamos crear una función que se encargue de dicho cálculo en un servicio y usarlo siempre que se lo requiera. De esta manera no tendremos que repetir código en varios lugares y en caso de modificarse alguna condición o variable, podríamos actualizar el código una sola vez y no en todos los lugares donde lo hayamos implementado en caso de no usar un servicio.

Podemos crear servicios usando el siguiente comando:

ng generate service path/nombre-del-servicio

Inyección de dependencias

La inyección de dependencias es un patrón de diseño que nos permite delegar la creación de clases para evitar que una clase se cree varias veces. Recordemos que un servicio solo debe instanciarse una vez y todas las veces que se lo requiera, se debe apuntar a esa única instancia.

La inyección de dependencias nos permite definir una clase en el constructor del componente o directiva en la que estemos trabajando y Angular se encargará de crear la instancia o bien apuntar a la instancia que ya existe.

Para hacer uso de la inyección de dependencias, debemos definir la clase en el constructor.

```
constructor(private service: MyService) { }
```

De esta manera el servicio será accesible desde el componente donde se lo inyectó y se podrán hacer uso de todas sus variables y métodos que no fueron definidos como privados.

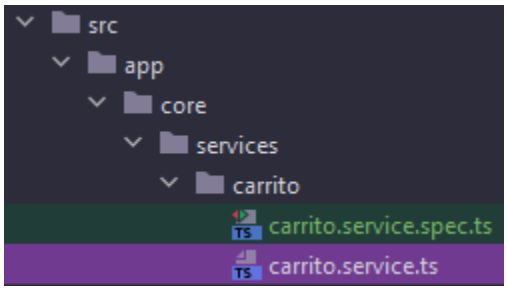
Creando nuestro primer Servicio

Vamos a crear un servicio. Los servicios tal como los componentes se deben definir en un módulo para ser utilizados. En la estructura de módulos que definimos en la clase anterior, los servicios se ubican en el módulo CORE. Dicho módulo recordemos, se importa en el módulo base de la aplicación por lo que los servicios no se deben importar en ningún otro lugar. Éstos están siempre disponibles.

Vamos a crear un servicio:

ng generate service core/services/carrito/carrito

Nótese que repetí al final la palabra carrito, esto se debe a que quiero que se genere una carpeta antes de generarse los archivos del servicio.



Veamos el archivo de servicios

Carrito.service.ts

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class CarritoService {
  constructor() { }
}
```

El decorador "*@Inyectable"* es aquel que se encarga de marcar esta clase como un servicio. Como el nombre lo indica es una clase que se puede inyectar, usando el sistema de inyección de dependencias. Pero como en Angular normalmente solo se inyectan Servicios, lo vamos a llamar simplemente servicio y no "inyectable".

Dentro del decorador podemos observar un objeto con una key: *`providedIn: 'root'`*

Dicha key indica que el servicio debe ser accesible desde todos lados. Por ahora no nos centramos en esta cuestión (además de que angular está pronto a deprecar dicho objeto).

Finalmente bajo el decorador vemos definida la Clase *CarritoService.* Como podemos observar, un servicio no es más que una clase la cual puede ser inyectada con el sistema de inyección de dependencias.

Accediendo al servicio

Para acceder a las variables y métodos del servicio, debemos declararlo en el constructor del componente como parámetro.

```
export class CarritoPageComponent implements OnInit {
  constructor(
    private carritoService: CarritoService,
  ) { }
  ngOnInit(): void {
    this.carritoService.tremendaFuncion();
  }
}
```

Una vez definido como parámetro y definido el tipo de dato del parámetro como el servicio, éste ya es accesible desde el componente.

Para ello simplemente debemos usar la palabra clave “this” seguido del nombre del parámetro.

```
this.carritoService.tremendaFuncion();
```

Observables

Los observables son una clase que emite de forma asíncrona un flujo de datos a los cuales podemos reaccionar mediante el uso de una callback.

Los observables constan de dos partes fundamentales: el propio observable y un observador. El Observable realizará una tarea específica y el observador será quien reciba el resultado de dicha tarea.

Supongamos que tenemos una fábrica de robots, y en la línea de producción, cada robot tiene una tarea específica que realizar. Los robots son como los observables, y los resultados de las tareas que realizan son como los eventos que los observables emiten.

Cada robot puede enviar una señal cuando ha completado su tarea, lo que es como la emisión de un evento por parte del observable. Esta señal puede ser recogida por otros robots en la línea de producción, que podrían necesitar esa información para realizar su propia tarea. Estos robots son como los observadores de los eventos emitidos por el observable.

Un observable por sí solo no realiza ninguna acción hasta que alguien espera algo de él. El término correcto para esto es “suscribirse a un observable”.

Para ello, vamos a invocar la función “*subscribe()*” del observable.

```
Observable.subscribe()
```

Un observable cuenta con 3 eventos, los cuales nos ayudan a saber el estado en el que se encuentra la tarea que realiza el observable.

- Next: el evento next se dispara cuando el observable emite un valor.
- Error: el evento error se dispara cuando el observable encuentra un error.
- Complete: el evento complete se dispara cuando el observable no va a emitir más valores, dado que su tarea se completó.

Cuando nos vamos a suscribir a un observable, podemos pasar un objeto con keys que corresponden a dichos eventos. A dichas keys le vamos a asignar una callback function la cual se ejecutará al dispararse dicho evento.

```
Observable.subscribe({
  next: value => {

  },
  error: err => {

  },
  complete: () => {

  }
});
```

No es necesario ejecutar algo cada vez que nos suscribimos, es posible que solo queramos obtener el valor por lo que definir solo el evento next es suficiente. Aún así es buena práctica siempre agregar el evento error en caso de que el observable falle por algún motivo.

Next

El evento next como mencionamos anteriormente, se ejecuta cuando el observable emite un nuevo valor. Esto puede ser en caso de ser un observable que espera una petición HTTP, cuando llega dicha respuesta. O bien cuando en caso de ser un observable que emite muchos valores, cuando una nueva respuesta es enviada a través del observable. Esto implica que la callback del evento next se ejecutará varias veces.

```
Observable.subscribe({
  next: value => {
    // value, tiene la respuesta del observable
    console.log(value);
  },
});
```

Error

El evento Error, se ejecuta solo UNA VEZ por cada vez que nos SUSCRIBIMOS a un observable. Una vez se ejecuta el evento error, el observable se completa automáticamente por lo que una vez tengamos un error, el observable no emitirá ningún resultado nuevo hasta que nos suscribimos nuevamente a él.

El parámetro del error, contiene el objeto de error o mensaje de error que el observable emitió.

```
Observable.subscribe({
  error: err => {
    // err, tiene la respuesta de error del observable
    console.log(err);
  },
});
```

Complete

El evento complete se ejecuta al finalizar un observable. Esto indica que ya no se enviarán más valores a no ser que vuelvas a suscribirte al observable.

Este evento es útil cuando tienes un límite en el número de respuestas o deseas limitar la cantidad de respuestas que se manejan en el observable.

El evento complete no envía ningún parámetro.

```
Observable.subscribe({
  complete: () => {
    // complete, no tiene parámetros
    console.log('observable completado');
  },
});
```

Pipes de Observables

Las pipes son operadores que se utilizan para componer y transformar y/o modificar secuencias de datos de los observables.

Las pipes permiten manipular y transformar los datos que fluyen a través de un observable antes de que lleguen al suscriptor, pero sin modificar el observable original.

Una Pipe se crea utilizando el operador pipe(), que acepta uno o más operadores como argumentos separados por comas. Estos operadores se aplican en orden, de izquierda a derecha (aunque normalmente los escribimos de arriba a abajo), al flujo de datos que pasa a través de la pipe.

Cada operador en la pipe realiza una tarea específica, como filtrar eventos, transformar datos, combinar múltiples fuentes de datos, retrasar eventos, muestrear valores, etc. Al encadenar varios operadores en una tubería, puedes construir un flujo de datos que se ajuste a tus necesidades particulares.

Existen muchos tipos de pipes, cada una con un propósito distinto, y las diferentes combinaciones de pipes nos permiten tener un control mucho mayor sobre el resultado de los observables.

Vamos a crear un observable de prueba con el operador “of” el cual nos crea un observable que devuelve cada valor que le pasamos en sus parámetros.

Nota: los nombres de variable de un observable, deben terminar con el símbolo “$”. Esto no es obligatorio pero es algo estándar al escribir los nombres de observables.

```javascript
import { of } from 'rxjs';

// Crea un flujo de datos con el operador `of`
const numbers$ = of(1, 2, 3, 4, 5);
numbers$.subscribe({
  next: result => {
    console.log('valor: ', result);
  }
});
// valor: 1
// valor: 2
// valor: 3
// valor: 4
// valor: 5
```

Hemos creado un observable que nos devuelve el resultado del observable. Como podemos observar, la callback next se ejecuta una vez por cada valor que le pasamos. Por consola sale cada valor.

Puedes ver este código en acción si copias y pegas el código de los bloques en: https://playcode.io/rxjs

Veamos qué sucede si aplicamos el pipe filter:

```javascript
import { of } from 'rxjs';
import {filter} from 'rxjs/operators'

// Crea un flujo de datos con el operador `of`
const numbers$ = of(1, 2, 3, 4, 5);
numbers$.pipe(
  filter(val => {
    return val > 1;
  })
).subscribe({
  next: result => {
    console.log('valor:', result);
  }
});
// valor: 2
// valor: 3
// valor: 4
// valor: 5
```

La pipe filter filtra los resultados de la callback según si se retorna true o false.

Como podemos observar, el valor final al suscribirnos cambió respecto al valor que teníamos en el observable. Veamos más ejemplos:

```javascript
import { of } from 'rxjs';
import {filter, map} from 'rxjs/operators'

// Crea un flujo de datos con el operador `of`
const numbers$ = of(1, 2, 3, 4, 5);
numbers$.pipe(
  filter(val => {
    return val > 1;
  }),
  map(val => {
    return val * 10;
  })
).subscribe({
  next: result => {
    console.log('valor:', result);
  }
});
// valor: 20
// valor: 30
// valor: 40
// valor: 50
```

La pipe “map” modifica cada valor y lo reemplaza por el valor que le retornamos a su callback. En este caso, cada valor fue multiplicado por 10. Podemos realizar las modificaciones que deseemos, pero debemos tener en cuenta el tipo de dato que retornamos, pues si recibimos un number y retornamos un string, la siguiente pipe o la suscripción recibirán un string.

Si tienes problemas para ejecutar los ejemplos, asegúrate que los operadores y pipes que utilices fueron importados.

Sistema de suscripción de observables

Un observable requiere que alguien se suscriba a él para ejecutar su función. Cuando nos suscribimos, se crea lo que se denomina una “suscripción” la cual se mantiene en ejecución en segundo plano.

Como una suscripción no sabe cuando el observable dejará de enviar resultados, la misma seguirá activa.

Esto implica que en segundo plano, una suscripción seguirá ejecutando las callback de eventos que creamos de forma automática e indefinida. Esto significa que manualmente debemos “desuscribirnos” del observable en cuestión a fin de evitar errores.

Supongamos que tenemos un Observable que nos devuelve cada 5 segundos el movimiento de bolsa de una empresa. Si nos suscribimos una vez, luego abandonamos la página y volvemos a entrar, una segunda suscripción se creará, pero la anterior al no haber sido limpiada, implica que nuestra callback se llama dos veces por cada respuesta del observable. Esto se puede repetir de forma indefinida hasta que la página empiece a generar muchísima carga para el sistema. Esto se conoce como “Memory leak”.

Tenemos dos criterios para desuscribirnos de un observable: solo nos importa la primera (y generalmente única) respuesta o nos interesan todas las respuestas del mismo.

Desuscribirnos de la primer respuesta

Para desuscribirnos de un observable cuya primera respuesta es la única que importa, haremos uso de la pipe “take”.

Normalmente los observables cuya primera respuesta es la única que nos interesa, es aquellos que cumplen un propósito único, como una llamada HTTP, donde no tendremos muchas respuestas, si no solo una.

La pipe “take” toma como argumento el número de respuestas que queremos recibir del observable. Una vez alcanzado dicho número, se desuscribe del observable.

```JavaScript
import { of } from 'rxjs';
import {take} from 'rxjs/operators'

const numbers$ = of(1, 2, 3, 4, 5);
numbers$.pipe(
  take(1)
).subscribe({
  next: result => {
    console.log('valor:', result);
  }
});
// valor: 1
```

La pipe “take” tomó solo el primer valor y luego se desuscribe del observable. Es por esto que, por consola, solo se mostró el primer valor y no los demás. Esto también significa que si por error utilizamos este método en un observable que envía muchos valores, solo tomaremos el primero.

Desuscribirnos de un observable con muchas respuestas.

En caso de que nuestro observable tenga muchas respuestas, vamos a almacenar la suscripción que creamos en una variable. Luego, cuando el momento sea idóneo, nos vamos a desuscribir.

Al ejecutar la función “subscribe()” de un observable, se retorna una suscripción. Ésta la podemos almacenar en una variable para posteriormente desuscribirnos.

```JavaScript
import { of } from 'rxjs';

// Crea un flujo de datos con el operador `of`
const numbers$ = of(1, 2, 3, 4, 5);
const subscription = numbers$.subscribe({
  next: result => {
    console.log('valor:', result);
  }
});

// esperamos al momento idóneo y ejecutamos
subscription.unsubscribe();
```

En el contexto de angular, lo que normalmente se hace es crear una variable, asignarla al suscribirnos y desuscribirnos en el evento ngOnDestroy() del ciclo de vida del componente

```javascript
export class CoolComponent implements OnDestroy {

  sub = new Subscription();
  constructor(
  ) {
    this.sub = of(1, 2, 3, 4, 5).subscribe();
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

Servicio HTTP

Si necesitamos conectarnos con alguna ubicación en internet, ya sea una api pública, un servicio externo o nuestro propio backend, necesitamos realizar peticiones HTTP.

El servicio HTTP de Angular es una característica integrada en el framework Angular que proporciona una forma conveniente de realizar solicitudes HTTP a servidores remotos. Proporciona métodos y funcionalidades para enviar solicitudes HTTP como GET, POST, PUT, DELETE, etc., y manejar las respuestas recibidas a través del uso de Observables.

Para utilizar el servicio HTTP de Angular, primero debes importar el módulo *HttpClientModule* en tu aplicación. Luego, puedes inyectar la clase *HttpClient* en tus componentes, servicios u otros lugares donde desees realizar solicitudes HTTP.

JavaScript

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

this.http.get('https://api.example.com/data').subscribe({
  next: data => {
    console.log(data); // Procesa los datos recibidos
  },
  error: err => {
    console.error(err); // Maneja cualquier error ocurrido
  }
});

this.http.post('https://api.example.com/data', {param:
'value'}).subscribe({
  next: data => {
    console.log(data); // Procesa los datos recibidos
  },
  error: err => {
    console.error(err); // Maneja cualquier error ocurrido
  }
});
```

En este ejemplo, se utiliza el método get() y post del servicio HttpClient para enviar una solicitud GET a la URL especificada ('https://api.example.com/data'). Luego, se suscribe al observable devuelto para recibir la respuesta del servidor. Dentro de la función de suscripción, puedes procesar los datos recibidos o manejar cualquier error que ocurra mediante el evento error.

Además de get() y post(), el servicio HTTP de Angular también proporciona otros métodos como put(), delete(), etc., para realizar diferentes tipos de solicitudes HTTP. También puedes configurar encabezados, enviar datos en el cuerpo de la solicitud, manejar autenticación y otras funcionalidades avanzadas utilizando las opciones proporcionadas por el servicio HTTP.

Dado que la respuesta de los métodos get, post, etc es un observable, todo lo anterior visto acerca de observables y pipes es compatible. Por ende podemos modificar los valores de las respuestas que nos llegan desde internet de la manera que deseemos.

Es importante aclarar que los observables son clases que funcionan de forma asíncrona, por lo que esperan la respuesta del servidor sin importar lo que este tarde en responder. Es por esto que si queremos utilizar los datos que nos llegan desde internet, debemos definir toda la lógica de programación relacionada dentro de la suscripción para poder tener acceso a dichos datos.

El servicio también provee las funciones para realizar peticiones http con la posibilidad de ingresar un tipo genérico en la función para así asignar un tipo de dato a la respuesta del evento next. Esto se hace indicando el tipo de dato que tendrá la respuesta entre <>. Podemos usar primitivos como bool y string, pero también podemos indicar interfaces.

JavaScript

```
@Injectable({
  providedIn: 'root'
})
export class ApiService {

  constructor(private httpService: HttpClient) {

this.httpService.get<IPikachu>('https://pokeapi.co/api/v2/pokemon
/pikachu').subscribe({
      next: pikachu => { // <- ahora es tipo de dato "IPikachu"
        console.log('pikachu info', pikachu)
      },
      error: err => {
        console.error(err);
      }
    })
  }
}

interface IPikachu {
  nombre: string
  // campos
}
```

De esta manera si nuestro IDE está debidamente configurado para trabajar con typescript, podremos beneficiarnos de las bondades del autocomplete. En caso de no pasar un tipo genérico, la respuesta se trata como de tipo *Object*.