educación IT
CODE YOUR FUTURE

Bootcamp Java Developer

Fase 2 - Java Web Developer

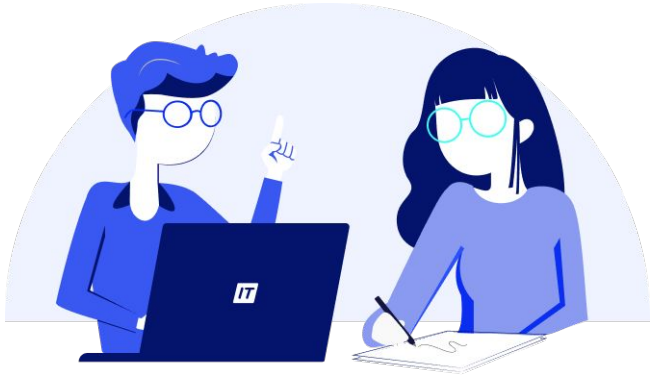
Módulo 19

Operaciones

Casos prácticos

Vemos ahora algunas aplicaciones prácticas. Para ello, vamos a usar la función **`fetch`**. Esta función recibe una URL y devuelve una promesa que, cuando resuelve nos da un objeto del tipo **`Response`**.

Ese objeto tiene métodos para convertir el contenido de la respuesta en formatos procesables. Uno de ellos es **`.json()`** que devuelve una promesa que, cuando resuelve, devuelve la respuesta con formato JSON.



Lectura desde API

La función **`fetch()`** ofrece un escenario particular. Su retorno es una promesa. Eso está bien. Pero para tener los datos en JSON necesito llamar a **`Response.json()`** que devuelve... ¡Otra promesa!

Entonces, podemos aplicar el encadenamiento de promesas, que hemos visto antes.

```
1  fetch('/posts')
2      .then((res) => res.json())
3      .then((datos) => console.log(datos))
4
```

Lectura y filtrado

En este caso aplicamos el encadenamiento para crear nuestro propio filtrado de datos.

```
1  const filtrarPorUsuario = (usuario, datos) => new Promise(function(resolver)
2  {
3      resolver(datos.filter((item) => item.username == usuario))
4  });
5
6  fetch('/posts')
7      .then((res) => res.json())
8      .then((datos) => filtrarPorUsuario('aperez', datos))
9      .then((resultados) =>  console.log(resultados))
```

Lectura, filtrado y reducción

Si bien podríamos haber hecho estas operaciones sin dedicarles una promesa a cada una, es una buena práctica ya que favorece el encadenamiento.

```
1  const filtrarPorUsuario = (usuario, datos) => new Promise(function(resolver)
2  {
3      resolver(datos.filter((item) => item.username == usuario))
4  });
5
6  const obtenerCantidad = (datos) => new Promise(function(resolver)
7  {
8      resolver(datos.length);
9  });
10
11 fetch('/posts')
12     .then((res) => res.json())
13     .then((datos) => filtrarPorUsuario('aperez', datos))
14     .then((datos) => obtenerCantidad(datos))
15     .then((resultado) =>  console.log('Cantidad de posts de aperez: ', resultado))
```

La biblioteca RxJS

La programación reactiva es un paradigma de programación asíncrona que se ocupa de los flujos de datos y la propagación del cambio.

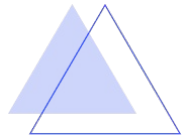
RxJS (Extensiones reactivas para JavaScript) es una biblioteca para la programación reactiva que utiliza observables que facilita la composición de código asíncrono o basado en devolución de llamada.

RxJS proporciona una implementación del tipo ***observable,*** que es necesario hasta que el tipo se convierta en parte del lenguaje y hasta que los navegadores lo admitan.

La biblioteca también proporciona funciones de utilidad para crear y trabajar con observables.

Estas funciones de utilidad se pueden usar para:

- Conversión de código existente para operaciones asíncronas en observables.
- Iterando a través de los valores en una secuencia.
- Mapeo de valores a diferentes tipos.
- Filtrado de flujos.
- Componer múltiples secuencias.
- Funciones de creación observables.
- RxJS ofrece una serie de funciones que se pueden usar para crear nuevos observables. Estas funciones pueden simplificar el proceso de creación de observables a partir de cosas como eventos, temporizadores y promesas.



Ejemplo simple

Crear un observable a partir de un evento:

```
import { fromEvent } from 'rxjs';

const el = document.getElementById('my-element');
const mouseMoves = fromEvent<MouseEvent>(el, 'mousemove');

const subscription = mouseMoves.subscribe(evt => {
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

Operadores

Son **funciones que se basan en la base de los observables para permitir una manipulación sofisticada de las colecciones**. Por ejemplo, RxJS define operadores como **`map()`**, **`filter()`**, **`concat()`** y **`flatMap()`**.

Los operadores toman opciones de configuración y devuelven una función que toma una fuente observable. Al ejecutar esta función devuelta, el operador observa los valores emitidos del observable de origen, los transforma y devuelve un nuevo observable de esos valores transformados.

Operadores comunes

RxJS proporciona muchos operadores, pero solo unos pocos se usan con frecuencia.

Para obtener una lista y ejemplos de uso, visita la documentación de la API de RxJS.

Creación	`from, fromEvent, of.`
Combinación	`combineLatest, concat, merge, startWith , withLatestFrom, zip.`
Filtrado	`debounceTime, distinctUntilChanged, filter, take, takeUntil.`
Transformación	`bufferTime, concatMap, map, mergeMap, scan, switchMap.`
Utilidad	`tap.`
Conversión	`share.`
Manejo de errores	`catch.`

Pipes

Se utiliza para vincular operadores entre sí. La función `pipe()` toma como argumentos las funciones que desea combinar y devuelve una nueva función que, cuando se ejecuta, ejecuta las funciones compuestas en secuencia.

Un conjunto de operadores aplicados a un observable es una receta, es decir, un conjunto de instrucciones para producir los valores deseados. Por sí misma, la receta no hace nada. **Debe llamar a `subscribe()` para producir un resultado** a través de la receta.

Para aplicaciones Angular, preferimos combinar operadores con tuberías, en lugar de encadenar. El encadenamiento se usa en muchos ejemplos de RxJS.

¡Sigamos trabajando!