

DWC

(Desarrollo Web en entorno cliente)



Angular

Tema 09

Peticiones Http

Índice

1.- Introducción	1
2.- HttpClient	3
2.1.- La clase HttpClient.....	3
2.2.- Uso de HttpClient.....	5
3.- Servicio HttpClient	7
3.1.- Petición get	7
3.2.- Petición delete	10
3.3.- Petición post	12
3.4.- Petición put	14
4.- Gestión de errores	16
5.- Transformando la información	18

1.- Introducción

La mayoría de las aplicaciones front-end necesitan comunicarse con un servidor a través del protocolo HTTP para descargar o cargar datos y acceder a otros servicios back-end.

En Angular podemos utilizar, al igual que en otros frameworks como React o Vue peticiones Http utilizando **Fetch**, ya que fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global `fetch()` que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red mediante el uso de promesas.

Al ser “nativo” de Javascript no deberemos importar ningún modulo externo ni interno a Angular, es decir lo podremos utilizar directamente en los ficheros de TypeScript al igual que hacíamos en JavaScript.

Vamos a hacer una petición a la API de artículos en la url 3000 de json-server y cada artículo en un card. Para ello utilizaremos un componente llamado fetch, en el ts añadiremos el código para traernos los datos en el `ngOnInit` y volcarlos en una variable llamada `artículos` que será la que interpolaremos en la vista mediante un `@for`. Usaremos fetch mediante las herramientas `asyn / await`.

Fichero `fetch.ts`

```
export class FetchComponent {
  articulos!: Artículo[];
  url = 'http://localhost:3000/articulos';

  async ngOnInit() {
    try {
      let response = await fetch(this.url);
      let datos = await response.json();
      if (!response.ok) throw new Error('Error');

      this.articulos = datos;
    } catch (error) {
      console.log(error);
    }
  }
}
```

Fichero fetch.html







```

<h3>Articulos con Fetch</h3>
<div id="contenedor" class="row row-cols-1 row-cols-md-6 g-4">
  @for(articulo of articulos; track articulo.id){
    <div class="card">
      
      <div class="card-body">
        <h5 class="card-title">
          {{ articulo.nombre }} ({{ articulo.unidades }})
        </h5>
        <p class="card-text">{{ articulo.descripcion }}</p>
        <b>
          <p class="card-text text-center">{{ articulo.precio }}</p></b>
        </div>
        <button class="btn btn-primary">Comprar</button>
      </div>
    }
  </div>

```

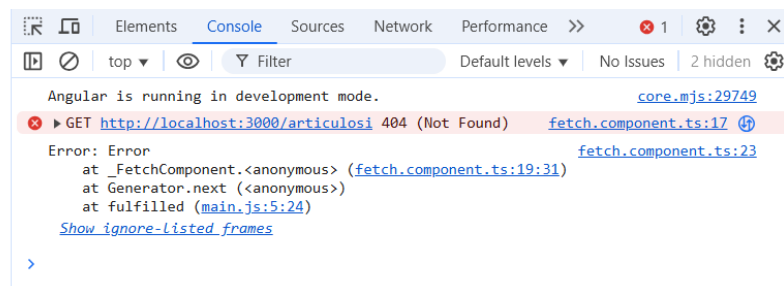
Resultado con petición correcta

Articulos con Fetch

					
Galaxy A32 (10)	Oppo A94 (10)	Galaxy S22 (10)	Apple iPhone (0)	Galaxy Z Flip4 (10)	Note 11 (10)
4GB + 128GB libre	8GB + 128GB libre	5G AMOLED libre	14 Pro móvil libre	5G móvil libre	6GB + 128GB
229	269	859	339	1990	300
Comprar	Comprar	Comprar	Comprar	Comprar	Comprar

Resultado con petición con error.

Articulos con Fetch



Como hemos visto podemos usar fetch con Angular pero no es la mejor solución para peticiones Http ya que Angular nos proporciona un cliente propio para peticiones Http

2.- Http Client

Angular nos proporciona una API HTTP de cliente para aplicaciones Angular, la clase **HttpClient** que se incluye en **@angular/common/http**

El servicio **HTTP utiliza Observables** que es un mecanismo para poder realizar transacciones asíncronas mediante flujo de datos. Para poder utilizar los observables que manipula este servicio deberemos utilizar algunas clases de las librerías **rxjs** y **rxjs/operators**

Para poder utilizar el HttpClient en nuestros proyectos ha habido cambios a partir de la versión 18, al igual que hemos visto en otros elementos.

- Hasta la 18 debíamos importar y declarar el HttpClientModule en el app.module.ts
- A partir de la 18 con componentes standalone deberemos incluir el proveedor del HttpClient mediante una función auxiliar en el fichero app.config.ts.

```
export const appConfig: ApplicationConfig = {  
  providers: [provideRouter(routes), provideHttpClient()],  
};
```

De esta manera, al igual que con el router, le estamos indicando a Angular que vamos utilizar el HttpClient en nuestro proyecto

Una vez definido el HttpClientModule o definido el proveedor providerHttpClient ya podemos utilizar HttpClient en nuestros componentes o servicios, mediante la inyección de dependencias al igual que hacíamos con el router.

2.1.- La clase HttpClient

La clase HttpClient tiene definidas todas las peticiones http como métodos, disponemos de **get**, **post**, **put** y **delete**. El mecanismo es similar a las promesas, definimos la petición en un observable y posteriormente nos subscribimos.

El esquema general podría ser:

```
httpClient.peticion<tipo-datos-respuesta>(url,options)
    .subscribe(datos=>procesar_datos,
               error=> gestionar_error)
```

Esta sería un método que el compilador de VS Code nos dirá que esta deprecated pero que funciona y se utiliza todavía. La alternativa para evitar el mensaje de deprecated es utilizar en la subscripción un objeto con los estados de un observable

```
httpClient.peticion<tipo-datos-respuesta>(url,options)
    .subscribe({next: datos=>procesar_datos,
               error: error=> gestionar_error
               })
```

De esta manera no nos mostrará ningún mensaje de error

Vamos a ver que es cada cosa:

- **httpClient**, es una variable del tipo httpClient que habremos inyectado previamente
- **petición**, serían las peticiones http get, post, put y delete
- **tipo-datos-respuesta**, es el tipo de datos que se devuelve en el observable. Si no tenemos exactamente definido el tipo se suele utilizar el tipo any, es de buenas prácticas utilizar una interface que haga el tipado correctamente
- **(url,options)**, son la url y las opciones de la petición. En las peticiones post y put además deberemos incluir un objeto para mandar los datos a insertar o a modificar
- **subscribe**, es el mecanismo que utilizamos para subscribirnos a la petición asíncrona y cuando acabe podamos gestionar el resultado
- **datos**, es la respuesta recibida al finalizar la petición
- **procesar_datos**, son las acciones para gestionar los datos recibidos
- **gestionar_error**, son las acciones para gestionar el error producido.

Para poder controlar los errores veremos más adelante varias formas de hacerlo.

2.2.- Uso de HttpClient

Cuando realizamos una aplicación en Angular que utiliza HttpClient tenemos dos formas de hacerlo:

- Usar HttpClient directamente en un componente, para ello importamos e inyectamos HttpClient en el componente y definimos las peticiones y las subscripciones en el componente.
- Usar un servicio en el cual importamos e inyectamos HttpClient. En este caso definiremos las peticiones http como métodos de ese servicio. Para usar los datos en el componente se importa e inyecta el servicio. Las peticiones se hacen mediante llamadas a los métodos del servicio con su correspondiente subscripción.

Vamos a ver el acceso a la bd de artículos **usando el HttpClient directamente en el componente**, para ello creamos el componente y hacemos la petición y la subscripción en el ts.

```
import { Component } from '@angular/core';
import { Artículo } from '../Modelos/articulo';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-articulos-http',
  standalone: true,
  imports: [],
  templateUrl: './articulos-http.component.html',
  styleUrls: ['./articulos-http.component.css'],
})
export class ArticulosHttpComponent {
  url = 'http://localhost:3000/articulos';
  articulos!: Artículo[];







  constructor(private miHttp: HttpClient) {}

  ngOnInit() {
    this.miHttp
      .get<Artículo[]>(this.url)
      .subscribe({
        next: (articulos) => (this.articulos = articulos),
        error: (error) => console.log('Error' + error ),
      });
  }
}
```

El html es el mismo que ya hemos utilizado y que interpola la variable artículos en la vista mediante una directiva @for.

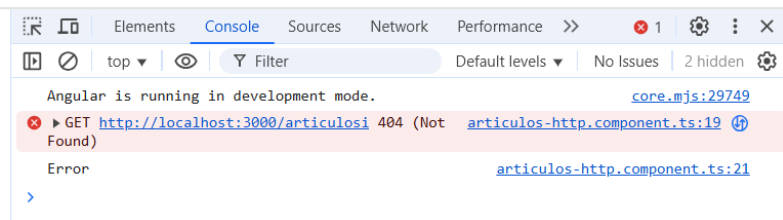
Resultado con petición correcta.

Artículos con httpClient

					
Galaxy A32 (10)	Oppo A94 (10)	Galaxy S22 (10)	Apple iPhone (0)	Galaxy Z Flip4 (10)	Note 11 (10)
4GB + 128GB libre	8GB + 128GB libre	5G AMOLED libre	14 Pro móvil libre	5G móvil libre	6GB + 128GB
229	269	859	339	1990	300
Comprar	Comprar	Comprar	Comprar	Comprar	Comprar

Resultado con error.

Artículos con httpClient



Esta forma de uso del HttpClient no da error y funciona, pero no es la más correcta porque hemos visto que en Angular para la gestión de la información disponemos de los servicios que permiten separar la implementación de las peticiones, de la gestión de las mismas.

Además, el uso de servicios nos permite la reutilización de todas las peticiones a través del servicio

3.- Servicio HttpClient

Lo más habitual y recomendable cuando utilicemos peticiones Http con HttpClient es utilizar un servicio que implemente las peticiones. Para usar este servicio luego en el componente deberemos inyectarlo y subscribirnos a las peticiones creadas en el servicio.

Vamos a definir un servicio para gestionar nuestras peticiones Http con HttpClient. Una vez creado el servicio, deberemos inyectar en el constructor del servicio el HttpClient para poder trabajar con él.

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

@Injectable({
  providedIn: 'root',
})
export class HttpService {
  constructor(private http: HttpClient) {}
}
```

Ahora ya podremos utilizar los métodos Http en nuestros métodos del servicio.

3.1.- Petición get

Vamos a definir un método en nuestro servicio para obtener los artículos de la base de datos. Para ello definimos el **método getAll** que no recibe ningún parámetro y **devuelve en un Observable que será un array de tipo Artículo**, con los artículos. Para poder trabajar con los observables deberemos **importar Observable de la librería rxjs**

El método get() tiene dos argumentos:

- La URL desde la que obtener los datos
- Un objeto de opciones que se utiliza para configurar la solicitud, principalmente utilizado para los headers

```
options: {
  headers?: HttpHeaders | {[header: string]: string | string[]},
  observe?: 'body' | 'events' | 'response',
  params?: HttpParams|{[param: string]: string | number | boolean | ReadonlyArray<string | number | boolean>},
  reportProgress?: boolean,
  responseType?: 'arraybuffer' | 'blob' | 'json' | 'text',
```

```
withCredentials?: boolean,
}
```

El método `get()` necesita las siguientes opciones: `{observe: 'body', responseType: 'json'}`. Estos son los valores predeterminados para esas opciones, por lo que lo habitual es no pasar en las llamadas el objeto de opciones.

Definimos la url como una propiedad del servicio e implementamos el método.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Artículo } from '../Modelos/articulo';

@Injectable({
  providedIn: 'root',
})
export class HttpService {

  constructor(private http: HttpClient) {}

  url = 'http://localhost:3000/articulos';

  getAll(): Observable <Artículo[]>{
    return this.http.get<Artículo[]>(this.url)
  }
}
```

Hemos implementado un método llamado `getAll()` que devuelve un observable con un array de artículos mediante el método `get` del `HttpClient`.

Para poder usarlo en el componente deberemos inyectar el servicio y en el `ngOnInit` subscribirnos al método `getAll` para obtener la lista de artículos. En la vista interpolaremos los artículos con la directiva `@for`

```
import { Component } from '@angular/core';
import { Artículo } from '../../Modelos/articulo';
import { HttpService } from '../../Services/http.service';

@Component({
  selector: 'app-articulos',
  standalone: true,
  imports: [],
  providers: [],
  templateUrl: './articulos.component.html',
  styleUrls: ['./articulos.component.css'],
})
```

```
export class ArticulosComponent {
  articulos!: Articulo[];

  constructor(private miServicio: HttpService) {}

  ngOnInit() {
    this.miServicio.getAll().subscribe({
      next: (datos) => (this.articulos = datos),
      error: (error) => console.log('ERROR ' + error.status),
    });
  }
}
```

Definimos un componente que mostrará los artículos recibidos en una tabla, de esa manera iremos desarrollando una interface típica para implementar un CRUD sobre una entidad.

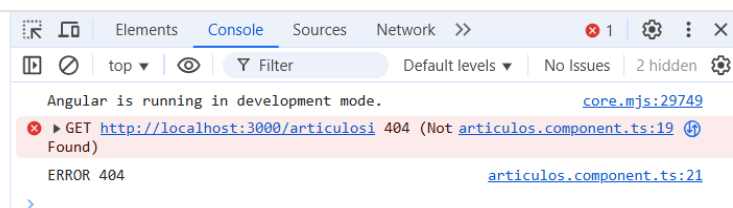
Resultado de petición correcta

Articulos con Servicio Http

Id	Nombre	Descripción	Precio	Unidades
m1	Galaxy A32	4GB + 128GB libre	229	10
m2	Oppo A94	8GB + 128GB libre	269	10
m3	Galaxy S22	5G AMOLED libre	859	10
m4	Apple iPhone	14 Pro móvil libre	339	0
m5	Galaxy Z Flip4	5G móvil libre	1990	10
m6	Note 11	6GB + 128GB	300	10

Resultado de petición con error

Articulos con Servicio Http



Vamos a implementar una petición get para un artículo en concreto, para ello deberemos de tener su id y acceder a la ruta indicada por el id.

Definimos el método en el servicio

```
get(id: string): Observable<Articulo> {
```

```
return this.http.get<Articulo>(`${this.url}/${id}`);
}
```

En este caso estamos recibiendo un parámetro en el método get que utilizamos para completar la ruta hasta el endpoint del artículo indicado. Destacar que en este caso lo que devuelve el método es un observable del tipo Articulo, sólo uno, no un array como en el método getAll().

Para su uso en el componente añadiremos un botón de Ver para cada artículo que tendrá en el evento click la llamada al método ver.

```
<button class="btn btn-primary" (click)="ver(articulo.id)">Ver</button>
```

El método ver será el que llame a la petición get del servicio http y con el dato recibido lo mostraremos por la consola

```
ver(id: string) {
  this.miServicio
    .get(id)
    .subscribe({
      next: (articulo) => console.log(articulo),
      error: (error) => console.log('Error' + error.status),
    });
}
```

Resultado de petición correcta

Articulos con Servicio Http

Id	Nombre	Descripción	Precio	Unidades	
m1	Galaxy A32	4GB + 128GB libre	229	10	Ver
m2	Oppo A94	8GB + 128GB libre	269	10	Ver
m3	Galaxy S22	5G AMOLED libre	859	10	Ver

Angular is running in development mode. [core.mjs:29749](#)

[articulos.component.ts:29](#)

```
{id: 'm2', nombre: 'Oppo A94', descripcion: '8GB + 128GB Libre', unidades: 10, precio: 269}
```

3.2.- Petición delete

La petición delete es similar a la petición get, deberemos pasar un id y acceder con el método delete al endpoint indicado para proceder a su borrado.

Para ello añadimos un botón de borrar para cada elemento de la tabla, y le asignamos en el click el método borrar que recibirá el id a borrar y llamara a la petición delete de nuestro servicio http, un procedimiento similar al get de un artículo que acabamos de hacer.

Definimos la petición delete en nuestro servicio

```
delete(id: string): Observable<Articulo> {
  return this.http.delete<Articulo>(`${this.url}/${id}`);
}
```

Estamos devolviendo un Observable de tipo articulo porque nuestra api para el borrado devuelve un objeto de tipo Articulo.

Añadimos al componente el botón para borrar

```
<button class="btn btn-danger" (click)="borrar(articulo.id)">Borrar</button>
```

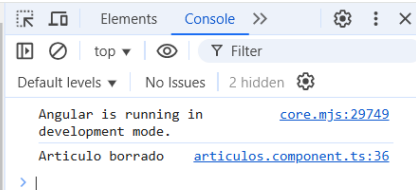
Definimos el componente el método borrar.

```
borrar(id: string) {
  this.miServicio.delete(id).subscribe({
    next: (dato) => console.log('Articulo borrado'),
    error: (error) => console.log('ERROR ' + error.status),
  });
}
```

Resultado de petición correcta

Articulos con Servicio Http

Id	Nombre	Descripción	Precio	Unidades		
m1	Galaxy A32	4GB + 128GB libre	229	10	Ver	Borrar
m2	Oppo A94	8GB + 128GB libre	269	10	Ver	Borrar
m3	Galaxy S22	5G AMOLED libre	859	10	Ver	Borrar
m4	Apple iPhone	14 Pro móvil libre	339	0	Ver	Borrar



Como vemos el artículo ha sido borrado, pero no se ha actualizado la tabla, para reflejar esos cambios deberíamos de volver a realizar una petición getAll para traernos todos los artículos existentes y ver que el borrado ya no aparece.

Para ello y optimizar código, lo que vamos a hacer es crear un método llamado cargaArticulos, que será el que llame a la petición getAll, y ese método lo utilizaremos en todos los sitios que requieran de la actualización de la tabla de artículos. El código de ese método será el mismo que tenemos ya definido en el ngOnInit, con lo cual, lo que haremos será sacar ese código al método cargaArticulos y en el ngOnInit llamarlo, de esta manera simplificaremos el código.

Con lo cual quedaría así

```
ngOnInit() {  
  this.cargaArticulos()  
}  
  
cargaArticulos(){  
  this.miServicio.getAll().subscribe({  
    next: (datos) => (this.articulos = datos),  
    error: (error) => console.log('ERROR ' + error.status),  
  });  
}
```

Y en el método de borrar, después de que se realice el borrado llamaremos a cargaArticulos para actualizar la tabla.

```
borrar(id: string) {  
  this.miServicio.delete(id).subscribe({  
    next: (dato) => {  
      console.log('Articulo borrado');  
      this.cargaArticulos();  
    },  
    error: (error) => console.log('ERROR ' + error.status),  
  });  
}
```

Ahora cada vez que borremos, se volverá a cargar la tabla y se actualizarán los registros.

3.3.- Petición post

Para implementar la petición post en el servicio vamos a crear un método que reciba un objeto de tipo Artículo y que devuelva un Observable con el dato del artículo insertado a través de la petición post.

En la petición post a parte de la url, deberemos incluir en el body el objeto a insertar.

Definimos el método post en el servicio

```
post(articulo:Articulo):Observable<Articulo>{  
  return this.http.post<Articulo>(this.url,articulo)  
}
```

En el componente añadimos un botón para Nuevo artículo y en su click llamamos al método nuevo que nos pedirá por el artículo nuevo y luego llamará al método post del servicio para añadirlo.

En este ejemplo, por centrarnos sólo en las peticiones Http, leeremos los datos del nuevo artículo mediante prompts. En un proyecto real leeríamos los datos mediante un componente nuevo y con un formulario, como ya hemos hecho en ejemplos anteriores utilizando el router.

Implementamos el botón de Nuevo artículo en la vista del componente y le asignamos el método nuevo en el evento click

```
<button class="btn btn-success" (click)="nuevo()">Nuevo artículo</button>
```

Definimos el método nuevo, que leerá el nuevo artículo y lo insertará en la BD mediante la llamada al método post del servicio.

```
nuevo() {
  let articulo: Articulo = {
    id: prompt('Id del artículo:') || '',
    categoria: prompt('Categoría del artículo:') || '',
    nombre: prompt('Nombre del artículo:') || '',
    descripcion: prompt('Descripción del artículo:') || '',
    precio: Number(prompt('Precio del artículo:') || 0),
    unidades: Number(prompt('Unidades del artículo:') || 0),
  };
  this.miServicio.post(articulo).subscribe({
    next: (dato) => {
      console.log('Artículo insertado');
      this.cargaArticulos();
    },
    error: (error) => console.log('Error ' + error.status),
  });
}
```

El resultado para una petición correcta

Artículos con Servicio Http

Nuevo artículo

Id	Nombre	Descripción	Precio
m4	Apple iPhone	14 Pro móvil libre	339
m6	Note 11	6GB + 128GB	300
m7	Realme 9	8GB + 128GB	300

localhost:4200 dice

Id del artículo:

Aceptar

Cancelar

10

Ver

Borrar

Después de introducir todos los datos del artículo a1, éste aparecerá en la tabla de artículos.

Artículos con Servicio Http Nuevo artículo

Id	Nombre	Descripción	Precio	Unidades		
t3	Samsung UE305	Full HD, HDR	350	10	Ver	Borrar
m1	Nuevo	Nuevo artículo	100	100	Ver	Borrar
a1	Artículo nuevo	Insertado con POST	10	100	Ver	Borrar

Resultado con error

Artículos con Servicio Http Nuevo artículo

Id	Nombre	Descripción	Precio	Unidades		
m1	Galaxy A32	4GB + 128GB libre	229			
m2	Oppo A94	8GB + 128GB libre	269			
m3	Galaxy S22	5G AMOLED libre	859	10	Ver	Borrar

localhost:4200 dice
Id del artículo:
m1
Aceptar Cancelar

Intentamos dar de alta un artículo nuevo con el id m1, que ya existe en la BD

Artículos con Servicio Http Nuevo artículo

Id	Nombre	Descripción	Precio	Unidades		
m1	Galaxy A32	4GB + 128GB libre	229	10	Ver	Borrar
m2	Oppo A94	8GB + 128GB libre	269	10	Ver	Borrar
m3	Galaxy S22	5G AMOLED libre	859	10	Ver	Borrar

Angular is running in development mode.
core.js:29749
articles.component.ts:56
POST http://localhost:3000/articulos 500 (Internal Server Error)
Error: 500 articles.component.ts:61

3.3.- Petición put

Para implementar la petición put en el servicio vamos a crear un método que reciba un objeto de tipo Artículo y que devuelva un Observable con el dato del artículo modificado a través de la petición put.

En la petición put a parte de la url, con el endpoint del id del artículo deberemos incluir en el body el objeto con los datos a modificar.

Definimos el método put en el servicio

```
put(id:string,articulo: Artículo): Observable<Artículo> {
  return this.http.put<Artículo>(`${this.url}/${id}`, articulo);
}
```


Implementamos en la vista el botón de modificar y le asignamos al evento click el método modificar, que pedirá los datos nuevos para el artículo y los modificará mediante la petición put del servicio http.

```
<button class="btn btn-dark" (click)="modificar(articulo)">Modificar</button>
```

El método modificar

```
modificar(articulo: Artículo) {
  let artModificado: Artículo = {
    id: articulo.id,
    categoria: prompt('Nueva Categoría: ', articulo.categoria) || '',
    nombre: prompt('Nuevo Nombre: ', articulo.nombre) || '',
    descripcion: prompt('Nueva Descripción: ', articulo.descripcion) || '',
    precio: Number(prompt('Nuevo Precio: ', articulo.precio.toString()) || 0),
    unidades: Number(prompt('Nuevas Unidades: ', articulo.unidades.toString()) || 0)
  };

  this.miServicio.put(articulo.id, artModificado).subscribe({
    next: (dato) => {
      console.log('Artículo modificado');
      this.cargaArticulos();
    },
    error: (error) => console.log('Error ' + error.status),
  });
}
```

Resultado petición correcta. Modificamos el artículo m1

Artículos con Servicio Http

Nuevo artículo

Id	Nombre	Descripción	Precio	Unidad	
m1	Galaxy A32	4GB + 128GB libre	229	10	
m2	Oppo A94	8GB + 128GB libre	269	10	
m3	Galaxy S22	5G AMOLED libre	859	10	Ver Borrar Modificar
m4	Apple iPhone	14 Pro móvil libre	339	0	Ver Borrar Modificar

localhost:4200 dice

Nueva Categoría:

Aceptar Cancelar

El resultado

Artículos con Servicio Http

Nuevo artículo

Id	Nombre	Descripción	Precio	Unidades	
m1	Galaxy A32 modificado	4GB + 128GB libre modificado	100	100	Ver Borrar Modificar
m2	Oppo A94	8GB + 128GB libre	269	10	Ver Borrar Modificar
m3	Galaxy S22	5G AMOLED libre	859	10	Ver Borrar Modificar

4.- Gestión de errores

Hemos visto cómo utilizar un servicio Http mediante subscripciones a los métodos del HttpClient implementados en el servicio, si la petición es correcta, se realiza la acción correspondiente y si diera algún error, se genera un error.

Puede resultar más interesante controlar el error en la propia petición, de esta manera se podría controlar el tipo de error en la propia petición y no esperar a que nos llegue al componente. Por ejemplo, podríamos utilizar una función de error que nos comprobara que tipo de error se ha producido en nuestra petición y que devolviera directamente al componente el mensaje que nosotros quisiéramos, de esta manera, no tendríamos que analizar nada en el componente.

Definimos la función handleError

```
private handleError(error: HttpErrorResponse) {
  let mensajeError = 'Error desconocido';
  if (error.error instanceof ErrorEvent) {
    // Error del lado del cliente (red o error en el código)
    mensajeError = `Error: ${error.error.message}`;
  } else {
    // Error del lado del servidor
    switch (error.status) {
      case 400:
        mensajeError = 'Solicitud incorrecta (Bad Request)';
        break;
      case 401:
        mensajeError = 'No autorizado (Unauthorized)';
        break;
      case 403:
        mensajeError = 'Prohibido (Forbidden)';
        break;
      case 404:
        mensajeError = 'No encontrado (Not Found)';
        break;
      case 500:
        mensajeError = 'Error interno del servidor (Internal Server Error)';
        break;
      case 502:
        mensajeError = 'Bad Gateway';
        break;
      // Podemos agregar más casos para otros códigos de error
      default:
        mensajeError = `Error del servidor: ${error.status}`;
    }
  }
}
```

```
// Relanzamos el error para que pueda ser manejado por el componente que
// llame al servicio
return throwError(() => new Error(mensajeError));
}
```

Al incluir la función manejadora de error aparecen dos errores en nuestro código, uno para **HttpResponseError** y otro para **throwError**. Para evitar estos errores deberemos de importarlos de las librerías **@angular/common/http** y **rxjs** en nuestro servicio

En las peticiones de HttpClient **podemos interceptar el error**, para ello deberemos **utilizar catchError**. Para poder utilizar catchError en nuestra petición **deberemos utilizar el operador pipe**, este operador lo que nos permite es poder encadenar a la petición get el catchError, es decir le permite pasar el error después de hacer el get, Para poder utilizar el operador pipe, deberemos importarlo de la librería **rxjs**

Nuestra petición getAll quedaría así:

```
getAll(): Observable<Articulo[]> {
  return this.http
    .get<Articulo[]>(this.url)
    .pipe(catchError(this.handleError));
}
```

Ahora, al igual que hemos hecho esta modificación en la petición getAll, la podríamos realizar en cualquier petición del servicio. En las subscripciones no tendemos que hacer nada, ya que, si algo va mal, seguirán recibiendo un error, pero ya “tratado” desde la propia petición.

Cuando sucede un error en la petición **Angular nos permite poder volver a lanzar la petición para ver si en ese intervalo de tiempo se ha podido subsanar el error**. Para poder realizar esta opción deberemos utilizar el operador **retry**. Este operador lo colocaremos antes de la gestión del error, para que pasemos al error una vez finalizados los intentos de conexión especificados en el operador retry. El operador retry recibe como parámetro el número de intentos de conexión a realizar si se detecta un error. Para poder utilizarlo deberemos **importarlo de la librería rxjs**

Nuestra petición quedara así

```
getAll(): Observable<Articulo[]> {  
  return this.http  
    .get<Articulo[]>(this.url)  
    .pipe(retry(3), catchError(this.handleError));  
}
```

5.- Transformando la información

Es bastante habitual que la API que consumamos nos devuelva los datos en un formato que no se ajusta perfectamente a las necesidades de nuestra aplicación. Por ejemplo, la API podría devolver datos con nombres de propiedades diferentes, o podría faltar información que necesitamos calcular. En estos casos deberemos realizar una transformación de la información recibida al formato que necesita nuestra aplicación.

Podemos usar los operadores de transformación de RxJS dentro del pipe de nuestro Observable. Uno de los operadores más utilizados para este propósito es el operador map.

El operador map lo que nos permitirá es transformar la información recibida en la petición http en un nuevo formato.

Vamos a implementar un método en nuestro servicio que le pasemos una categoría de artículos y nos devuelva un objeto con el nombre de la categoría y el total de artículos que tiene.

Para realizarlo solo disponemos de los métodos estándar del HttpClient, con lo cual lo que deberemos es implementar un método que nos devuelva en este caso un observable del tipo que nos piden, es decir un objeto con dos propiedades. Para obtener los datos lo que haremos es acceder mediante una petición get y posteriormente con los datos recibidos realizaremos una transformación para obtener el resultado que queremos.

Vamos a implementa el método totalCategoria

```
totalCategoria(
  categoria: string
): Observable<{ nombre: string; total: number }> {
  return this.http.get<Articulo[]>(this.url).pipe(
    retry(3),
    catchError(this.handleError),
    map((datos) => {
      let total = datos.filter((d) => d.categoria == categoria).length;
      return { nombre: categoria, total: total };
    })
  );
}
```

En este caso lo que va a devolver nuestro método es un observable de un objeto con dos propiedades nombre y total. Podíamos haber implementado una interface para este tipo de datos y haberla utilizado en el tipado del observable.

Como vamos a utilizar operadores RXJs debemos añadir el operador pipe, en este caso hemos dejado también los operadores retry y catchError, con lo cual para usar el map lo añadimos detrás del último que teníamos, detrás del catchError.

El map lo que va a recibir es el conjunto de datos que devuelve la petición get, con esos datos lo que vamos a hacer es un filtro por la categoría que nos han dicho y calcular el total. Una vez que tengamos el total, devolvemos el objeto que tendrá el nombre y el total.

Para consumir este nuevo método desde el componente será igual que con cualquier otro método del servicio.

Para probarlo vamos subscribirnos al método en el ngOnInit del componente artículos y así al cargar los componentes nos mostrara por consola el total de artículos de una categoría, en mi caso voy a probarlo con Moviles.

```
ngOnInit() {
  this.miServicio.totalCategoria('Moviles').subscribe({
    next: (datos) => console.log(datos),
    error: (error) => console.log(error),
  });
  this.cargaArticulos();
}
```

El resultado será

Articulos con Servicio Http

[Nuevo artículo](#)

Id	Nombre	Descripción	Precio	Unidades			
m1	Galaxy A32	4GB + 128GB libre	229	10	Ver	Borrar	Modificar
m2	Oppo A94	8GB + 128GB libre	269	10	Ver	Borrar	Modificar
m3	Galaxy S22	5G AMOLED libre	859	10	Ver	Borrar	Modificar

