Cliente HTTP

La mayoría de las aplicaciones front-end se comunican con servicios back-end a través del protocolo **HTTP**. La mayoría de los navegadores modernos soportan dos tipos de APIs para realizar pedidos HTTP: la interfaz **XMLHttpRequest** y la **API Fetch**.

El *HttpClient* de Angular, que pertenece a *@angular/common/http* ofrece un cliente de API HTTP simplificado para aplicaciones Angular que se basa en la interfaz *XMLHttpRequest* expuesta por los navegadores.
La clase HttpClient de Angular 5 está representada de la siguiente forma:

```
constructor(handler: HttpHandler)
 request(first: string | HttpRequest<any>, url?: string, options: {...}):
Observable<any>
 delete(url: string, options: {...}): Observable<any>
 get(url: string, options: {...}): Observable<any>
 head(url: string, options: {...}): Observable<any>
 jsonp<T>(url: string, callbackParam: string): Observable<T>
 options(url: string, options: {...}): Observable<any>
 patch(url: string, body: any | null, options: {...}): Observable<any>
 post(url: string, body: any | null, options: {...}): Observable<any>
 put(url: string, body: any | null, options: {...}): Observable<any>
}
```

Cómo puedes ver, todos los métodos devuelven un Observable y podemos realizar cualquier petición HTTP, además, gracias al método request podemos realizar cualquier petición ya que el primer parámetro es el método (GET, POST, PUT, DELETE).
Cada vez que hacemos llamadas a un servidor externo queremos que el usuario pueda continuar interactuando con la página. Es decir, no queremos que nuestra página se congele hasta que el request HTTP retorne desde el servidor externo. Para lograr ese efecto, nuestros pedidos HTTP son asincrónicos.

En JavaScript, generalmente hay tres enfoques para tratar con código asincrónico

- Callbacks
- Promises
- Observables

En Angular la manera preferida para manejar código asincrónico es mediante el uso de objetos *Observables*.

Configuración

Antes de que podamos usar el *HttpClient*, debemos importar el cliente *HttpClientModule*. En la mayoría de las aplicaciones de hace sobre el *AppModule* raíz.

```
import { NgModule }      from '@angular/core';
import { BrowserModule }   from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Una vez que importamos el *HttpClientModule* dentro del *AppModule*, podemos inyectarlo dentro de una clase de la aplicación:

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

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

Creación de un Request Básico

Construimos el componente

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
```

```
@Component({
 selector: 'app-simple-request',
 templateUrl: './simple-request.component.html',
 styleUrls: ['./simple-request.component.css']
})
export class SimpleRequestComponent implements OnInit {

 constructor(private http: HttpClient) { }

 ngOnInit() {
 }

}
```

Como vimos en la unidad anterior, mediante la definición de un atributo privado con su respectivo tipo en los parametros del constructor de la clase, inyectamos la dependencia *HttpClient* y su instancia la llamamos *http.*

Construimos el Template

```
<div class="container">
 <h1>Simple Request</h1>
 <button class="btn btn-info" type="button" (click)="makeRequest">Make
Request</button>
 <div *ngIf="loading">loading...</div>
 <pre>{{datos | json}}</pre>
</div>
```

Nuestro template tiene tres partes interesantes:

- El botón
- El indicador de loading
- Los datos

A través del boton unimos el evento click con un metodo definido en el componente llamado *makeRequest*.

Además queremos decirle al usuario que estamos cargando la solicitud, por eso es que vamos a mostrar el mensaje de *loading…*, y por eso es que a través de la directiva **ngIf** evaluamos la instancia de la variable loading, si es true, mostramos el mensaje, de lo contrario, lo ocultamos.

La variable del template *datos* es un **objeto**. Una buena forma de debuggear objetos es usando el pipe *json*.

No debemos olvidarnos de declarar ambas variables en el componente:

```
export class SimpleRequestComponent implements OnInit {
 datos: Object;
 loading: boolean;
```

Creamos nuestro primer pedido HTTP implementando el método makeRequest

```
makeRequest(): void {
   this.loading = true;
   this.http
     .get("https://jsonplaceholder.typicode.com/posts/1")
     .subscribe(data => {
       this.datos = data;
       this.loading = false
     });
 }
```

Cuando llamamos al método *makeRequest*, lo primero que hacemos es setear el valor de *loading* como true. Esto va a hacer que se muestre el indicador de *loading…* en la vista.
Para hacer un pedido http llamamos al método de *HttpClient* get (this.http.get) y le pasamos como parámetro la URL a la cual nosotros queremos hacerle el requerimiento por HTTP.
El método *http.get* retorna un objeto del tipo *Observable*. Suscribimos a los cambios usando el método *subscribe*.
Cuando nuestra solicitud HTTP retorna, emite un objeto del tipo *Response*. Extraemos el cuerpo de la respuesta como un objeto del tipo *Object* usando *json* y luego lo almacenamos en la variable *datos.*
Como ya tenemos la respuesta, es necesario ocultar el aviso de *loading…*, así que por lo tanto establecemos el valor de la variable *loading* cómo false (this.loading = false).

Escribiendo un servicio

En el ejemplo anterior llamamos al método *http.get()* desde dentro del componente, lo cual no refleja una buena práctica. De esta manera el Componente se llena rápidamente de métodos de acceso a datos. Se vuelve difícil de entender difícil de testear, y el acceso lógico a los datos no puede ser reutilizado u estandarizado.
Es por todos estos motivos que resulta una buena práctica encapsular el acceso a los datos en un servicio separado y delegar a ese servicio en el componente.

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

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

 constructor(private http: HttpClient) { }
}
```

Chequeando el tipo del Response

En el ejemplo anterior el método HttpClient.get() parseó el JSON que recibió como respuesta en un objeto del tipo *Object*. Este no sabe como está compuesto el objeto que recibimos como respuesta.

Sin embargo le podemos decir a HttpClient el tipo de respuesta que vamos a consumir y devolver el tipo especificado.

Primero definimos una interfaz con la forma correcta:

```
export interface IPost {
 userId: number;
 title:string;
 body:string;
}
```

Luego especificamos esa interfaz como el parámetro de tipo de llamada del HttpClient.get() en el servicio:

```
getPost(idPost : number){
   return
this.http.get<IPost>("https://jsonplaceholder.typicode.com/posts/1");
}
```

Después el método callback definido en el componente recibe un objeto tipado, el cual es mas sencillo y seguro de consumir:

```
export class RequestTipadoComponent implements OnInit {
 post: IPost;
 loading: boolean;

 constructor(private accesoADatosService : AccesoADatosService) { }

 ngOnInit() {
 }
```

```
makeRequest(){
  this.loading = true;
  this.accesoADatosService.getPost(1)
    .subscribe((data: IPost) => {
      this.loading = false;
      this.post = { ...data}
    });
}
```

Obteniendo el Response completo.

Hay veces que el cuerpo del response no devuelve toda la información que necesitamos. Por lo general los servidores retornan cabeceras especiales o códigos de estado para indicar ciertas condiciones que son importantes para el flujo de nuestra aplicación.
Para poder obtener el Response completo a través de la opción *observe*, se lo tenemos que especificar al HttpClient.

```
getPostResponse(idPost : number){
   return this.http.get<IPost>(
"https://jsonplaceholder.typicode.com/posts/1",{observe:'response'});
}
```

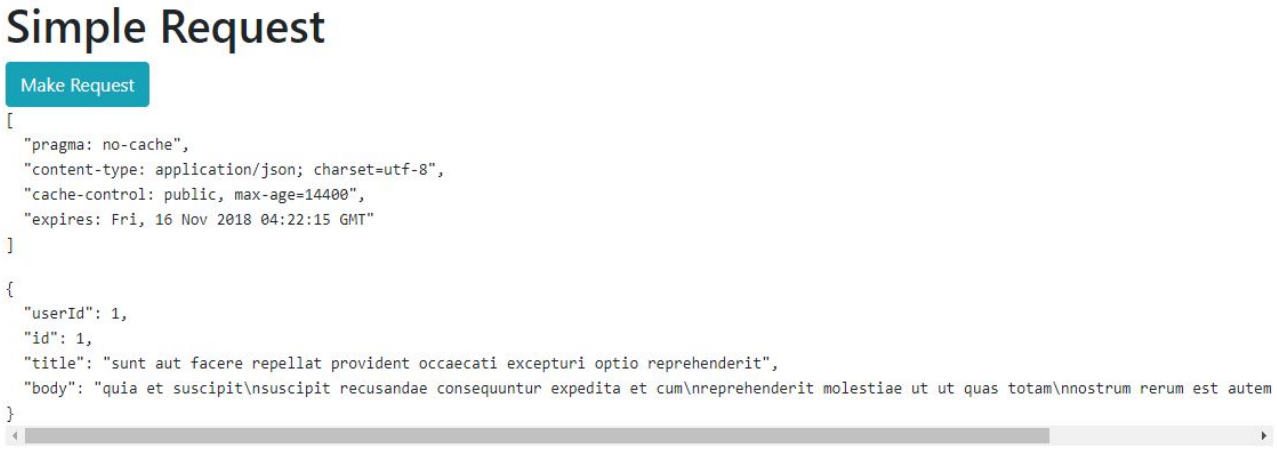
De esta manera, el método HttpClient.get() retorna un objeto Observable del tipo *HttpResponse* y no simplemente datos en formato JSON.
Ahora nos queda crear un método en el Componente capaz de mostrar los headers de la respuesta como así también los datos del post solicitado:

```
export class FullResponseComponent implements OnInit {
 post: IPost;
 loading: boolean;
 headers: string[];

 constructor(private accesoADatosService : AccesoADatosService) { }

 ngOnInit() {
 }
 makeRequest(){
   this.loading = true;
   this.accesoADatosService.getPostResponse(1)
     .subscribe(res => {
       this.loading = false;
       const keys = res.headers.keys();
       this.headers = keys.map(key =>
         `${key}: ${res.headers.get(key)}`);
       this.post = { ...res.body}
     });
 }
}
```

A la hora de mostar los datos en la vista, podemos aplicar el mismo criterio que en ejemplo anterior:

```
<div class="container">
 <h1>Simple Request</h1>
 <button class="btn btn-info" type="button"
(click)="makeRequest()">Make Request</button>
 <div *ngIf="loading">loading...</div>
 <pre>{{headers | json}}</pre>
 <pre>{{post | json}}</pre>
</div>
```

Y lo que vamos a ver en la pantalla del navegador después de presionar el botón *Make Request* es:



Manejo de Errores.

Cada vez que hacemos llamadas al servidor a través del protocolo HTTP podemos tener dos tipos de errores. El backend del server puede rechazar el requerimiento, devolviendo una respuesta HTTP con un código de estado tal como 404 o 500.

O algo puede salir mal en el lado del cliente, tal como un error en la red que previene que el requerimiento sea completado satisfactoriamente o una excepción en el operador RxJS. Estos tipos de errores producen objetos JavaScript del tipo *ErrorEvent*.

El *HttpClient* captura ambos tipos de errores a través del *HttpErrorResponse* y a través de él podemos inspeccionar la respuesta para ver qué fue realmente con lo que pasó.

Tanto la inspección, como la interpretación, como la resolución de los errores lo hacemos siempre en el servicio, no en el componente.

```
getPostError(idPost : number){
    return this.http.get<IPost>("https://noexiste.com")
      .pipe(
        catchError(this.handleError)
      )
```

```
  }

  private handleError(error: HttpErrorResponse){
    if(error.error instanceof ErrorEvent){
      // Error en el lado del cliente

      console.error('Ocurrió un error:', error.error.message);
    }else{
      // El backend devolvió un código de respuesta de error.

      console.error(`El Backend retornó el código ${error.status},`+
      `El cuerpo del mensaje del error es: ${error.message}`);
    }
    return throwError('Algo malo sucedió; por favor intente más
tarde.');
  }
```

El manejador de errores retorna un *ErrorObservable* de **RxJS** con un mensaje amigable de error.
Los consumidores de los servicios esperan métodos que retornen un objeto *Observable*, aunque este sea "malo".
Luego de obtener el *Observable* retornado por el *HttpClient* lo conducimos a través del Error Handler.

Observables.

Los *Observables* nos proveen soporte para poder enviar mensajes entre un publicador y suscriptor en nuestra aplicación. El uso de *Observables* ofrece grandes beneficios en relación a otras técnicas en relación al manejo de eventos, programación asincrónica y manejar múltiples valores.
Los Observables son declarativos, es decir, definimos un método para publicar valores, pero este no es ejecutado hasta que un consumidor no se suscribe. El consumidor suscripto recibe notificaciones hasta que el método se complete, o hasta que se desuscribe.
Un Observable puede devolver varios tipos de valores: literales, mensajes o eventos, dependiendo del contexto.

Librería RxJS.

La programación *reactiva* es un paradigma de programación asincrónica que se ocupa de los flujos de datos y la propagación de los cambios (Wikipedia). **RxJS**(Extensiones Reactivas para Javascript o Reactive Extensions for JavaScript) es una librería para programación reactiva que usa *observables* que hacen más fácil componer código asíncrono o basado en código basado en callbacks.

RxJS provee una implementación del tipo *Observable*, que es necesario hasta que este tipo de dato se convierta en parte del lenguaje JavaScript y sea admitido por los navegadores.
Esta librería provee funciones útiles para crear y trabajar con observables. Estas utilidades se pueden usar para:

- Convertir código existente para operaciones asíncronas en observables.
- Iterar a través de datos en una cadena o flujo de datos.
- Mapear valores de diferente tipo.
- Filtrar flujos de datos.

Enviar datos al servidor.

Ademas de recuperar datos del servidor, HttpClient nos permite enviar datos a través de otros métodos HTTP tales como **PUT, POST,** y **DELETE**.
Para este tipo de operaciones muchos servidores requieren que establezcamos determinadas opciones en el header de la solicitud. Por ejemplo, puede requerir un header del tipo *'Content-Type'* para declarar los tipos MIME en el cuerpo del request.

```
const httpOptions = {
 headers: new HttpHeaders({
   'Content-Type': 'application/json',
   'Authorization': 'my-auth-token'
 })
};
```

Peticiones de envíos de datos.

Las aplicaciones generalmente envian datos al servidor. El envío por lo general se produce cuando se hace un submit de un formulario. Veamos como enviamos un nuevo post a JSONPlaceholder.

```
addPost(post:IPost): Observable<IPost> {
   return
this.http.post<IPost>("https://jsonplaceholder.typicode.com/posts",post
,httpOptions)
      .pipe(
        catchError(this.handleError)
      );
 }
```

El método HttpClient.post() es similar al get() in que también tiene un parámetro definiendo el tipo(esperamos que el servidor nos devuelva un nuevo post) y necesita una url para acceder al recurso.
Pero además requiere dos parámetros adicionales:

- *post* - Los datos que serán enviados en el cuerpo de la solicitud.
- *httpOptions*: son las opciones que establecemos en la cabecera para poder enviar los datos.

El componente inicia el envío de datos cuando se suscribe al Observable retornado por el método del servicio.

```
makeRequest(){
    const post:IPost = {
      title : "Clase Angular",
      body: "Curso desarrollado para el aprendizaje de este framework
desarrollado por la gente de Google",
      userId:1,
      id: 0
    };
    this.enviarDatosService.addPost(post)
      .subscribe(post => this.postResponse = post);
 }
```

Cuando el servidor responde satisfactoriamente con el nuevo post agregado, el componente lo muestra en el template.

```
<div class="container">
 <h1>Enviar Datos a través de Post</h1>
 <button class="btn btn-info" type="button"
(click)="makeRequest()">Make Request</button>
 <pre>{{postResponse | json}}</pre>
</div>
```

Petición para borrar datos.

```
deletePost(id: number): Observable<{}> {
    const url = `https://jsonplaceholder.typicode.com/posts/${id}`;
    return this.http.delete(url, httpOptions)
      .pipe(
        catchError(this.handleError)
      )
 };
```

Como el componente no espera ningún resultado de la operación de *delete*, entonces nos suscribimos sin establecer ningún *callback*. Pero, aunque no esperemos resultado alguno, debemos de igual manera suscribirnos. Llamando al método *subscribe()* ejecutamos el observable, el cual inicia el requerimiento del tipo DELETE.

```
deletePost(){
   this.enviarDatosService.deletePost(1).subscribe();
 }
```

Un método *HttpClient* no ejecuta su petición HTTP hasta que no lo llamemos a través del *subscribe()*.

Petición para actualizar datos.

Una aplicación envía requerimientos PUT para reemplazar un recurso con datos actualizados. Veamos un ejemplo:

```
 actualizarPost(post: IPost): Observable<IPost> {
   return
this.http.put<IPost>("https://jsonplaceholder.typicode.com/posts/1",pos
t,httpOptions)
     .pipe(
       catchError(this.handleError)
     );
 }
```

En este caso llamamos al método HttpClient.put() pasándole como parámetro un objeto del tipo IPost, con los datos actualizados.

```
updatePost(){
   const post:IPost = {
     title : "Clase Angular - Modificamos el post",
     body: "Post con contenido actualizado",
     userId:1,
     id: 101
   };
   this.enviarDatosService.actualizarPost(post)
     .subscribe(post => this.postModificado = post);
 }
```

En el componente, como dijimos anteriormente, nos suscribimos al observable a través del método *subscribe()*. Debemos recordar que esta es la única manera de que el observable realice la petición HTTP.