

CURSO DE PROGRAMACIÓN JAVA FULLSTACK

Angular

4. Acceso a servicios web

Arturo Bernal Mayordomo

Edición: Noviembre 2019

Contenido

1. - Acceso a servicios web (HttpClient).....	3
1.1 Repaso de los métodos básicos HTTP.....	3
1.2 RxJS y Observables.....	4
1.3 Procesando respuestas y errores con Observables.....	4
1.4 Transformación de datos en el servicio de Angular.....	5
1.5 Otros métodos HTTP (POST, PUT y DELETE).....	6
1.6 Manejar token de autenticación con Angular.....	7
1.7 Usando un interceptor para enviar el token.....	8

1. - Acceso a servicios web (HttpClient)

En una aplicación real, los datos se obtienen de un servidor web utilizando peticiones HTTP en segundo plano (llamadas AJAX). Angular integra un servicio llamado **HttpClient** que se encarga de eso. Como viene incluido en el módulo **HttpClientModule**, debemos importarlo primero en el módulo de nuestra aplicación:

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

@NgModule({
  ...
  imports: [
    ...,
    HttpClientModule
  ],
  ...
})
export class AppModule { }
```

A continuación, vamos a modificar el servicio **ProductsService** para que en lugar de devolver un array de productos estático, haga una llamada a un servidor web para obtenerlos primero. Hay que tener en cuenta que para usar el servicio **HttpClient**, lo tenemos que inyectar primero (un servicio se puede inyectar en otro):

```
import { Injectable } from '@angular/core';
import { IProduct } from '../interfaces/i-product';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ProductsService {
  private productURL = 'http://arturober.com/products-angular/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<IProduct[]> {
    return this.http.get<{products: IProduct[]}>(this.productURL).pipe(
      map(response => response.products)
    );
  }
}
```

1.1 Repaso de los métodos básicos HTTP

La petición y respuesta del servidor será mediante el protocolo HTTP. Cuando hacemos una petición HTTP, el navegador envía al servidor: cabeceras (como *useragent* el cual identifica al navegador, las preferencias de idioma, etc.), el tipo de petición HTTP y parámetros o datos (si son necesarios).

Hay muchos [tipos de petición](#) que podemos enviarle al servidor. Los más usados cuando realizamos una llamada AJAX (o accedemos a un servicio web) son:

- **GET** → Consulta datos y normalmente no modifica nada en la base de datos. Equivale a SELECT en SQL. Cuando usamos este método, la información normalmente se envía concatenada en la URL (formato URLEncoded).
- **POST** → Operación para insertar un nuevo dato (no necesariamente) en el servidor. Equivale a INSERT en SQL. Los datos a insertar se envían dentro del cuerpo de la petición HTTP.
- **PUT** → Esta operación actualiza datos existentes en el servidor. Equivale a UPDATE en SQL. La información que identifica el objeto a actualizar es enviada en la URL (como en GET), y los datos a modificar se envían aparte en el cuerpo de la llamada (como en POST).
- **DELETE** → Esta operación elimina un dato del servidor, como la operación DELETE de SQL. La información que identifica al objeto a eliminar será enviada en la URL (como en GET).

1.2 RxJS y Observables

Se podría trabajar con [Promesas](#) para realizar las peticiones HTTP. Sin embargo, Angular trabaja por defecto con la librería [RxJS](#). Es decir, utiliza **Observables**, que son como una versión más avanzada de las promesas.

Estas son algunas diferencias entre usar promesas y observables:

- Una Promesa devuelve un sólo valor futuro (lo que para una petición HTTP es suficiente). Un Observable puede emitir varios valores a lo largo del tiempo.
- Una Promesa se ejecuta (el código interno) cuando se crea. Un observable sólo empieza cuando alguien se suscribe a él (lazy loading).
- Una promesa no puede ser cancelada, mientras que un observable puede dejar de ejecutarse si se cancelan las suscripciones a él.
- Los observables tienen muchos métodos ([operadores](#)) como **map**, **filter**, **reduce**, y muchos más. Las promesas usan el método genérico **then** para todo.

Cualquier llamada http (**get**, **post**, **put**, **delete**) devuelve un Observable (una operación futura). Para obtener los datos que vaya a emitir dicho Observable en el futuro, debemos suscribirnos a él.

Sin embargo, podemos establecer un procesamiento intermedio de los datos, entre la respuesta que nos llega en “crudo” del servidor y el dato que realmente queremos guardar/mostrar, usando operadores intermedios como **map**, **filter**, **tap**, ...

Esto es lo que hacen los métodos nombrados anteriormente:

- **map** → Obtiene el dato devuelto por el Observable (o el operador anterior si concatenamos varios), aplicamos alguna transformación al dato y la devolvemos (el método devolverá Observable<DatoTransformado>).
- **tap** → Se utiliza normalmente para operaciones de depuración (mostrar datos por consola, etc.). No devolvemos nada en este método porque automáticamente devuelve un Observable con el mismo dato que recibe (no modifica nada).
- **filter** → Cuando el observable devuelve más de un dato a lo largo del tiempo (por ejemplo si trabajamos con eventos). Este método sólo dejará pasar aquellos que cumplan una determinada condición (devolvemos un booleano).

Los operadores se importan de ‘**rxjs/operators**’, y se aplican como parámetros del método **pipe** de la clase Observable en el orden establecido (encadenados).

1.3 Procesando respuestas y errores con Observables

Cada vez que un observable emite un valor, este se puede procesar con los métodos u operadores intermedios nombrados antes. Sin embargo, el observable no empieza a emitir valores (no se ejecuta su código interno) hasta que nos suscribimos a él (pueden haber varias suscripciones al mismo observable).

Para suscribirnos debemos llamar al método **subscribe**. Este método es un **método final**, lo que significa que ya no podemos encadenar nada más después.

Este método puede recibir hasta 3 parámetros, que deben ser funciones. La **primera función** recibirá el resultado final devuelto por el observable (y procesado por los métodos intermedios). Si se produce algún error en el observable (o lo lanzamos nosotros en algún método intermedio) se llamará en su lugar a la **segunda función** (opcional), que recibirá dicho error. La **tercera función** (opcional) se ejecutará al final siempre independientemente de si se produce algún error o no. Es el mismo concepto que el bloque **finally** de una estructura **try...catch**.

```
Observable.pipe( map(...), catchError(...) ).subscribe(  
  (result) => // Procesar los datos devueltos  
  (error) => // Procesar el error  
  () => // Se ejecutará siempre al final (sin parámetro)  
)
```

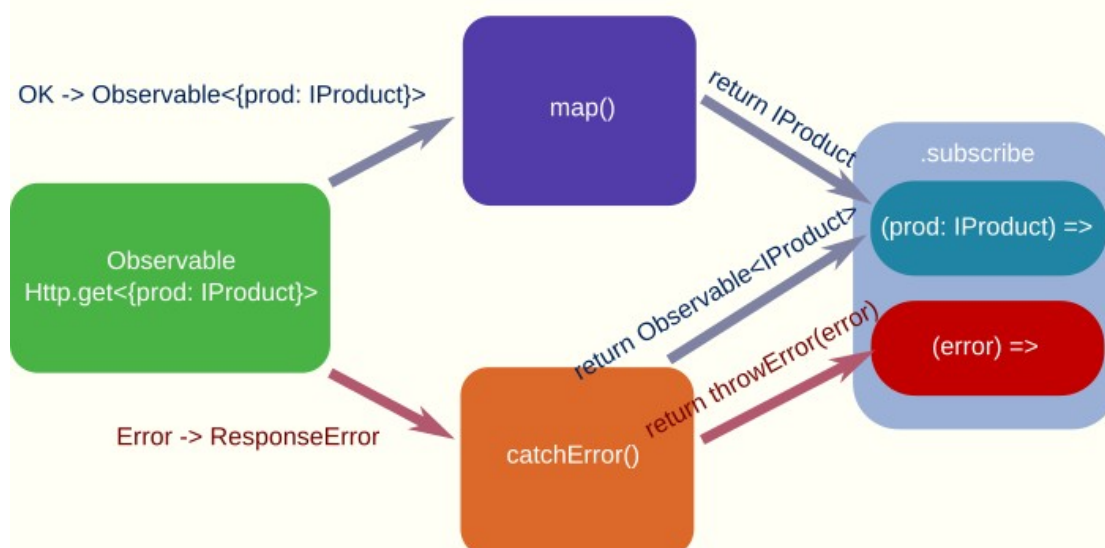
Sin embargo, a veces podríamos querer recuperarnos de un error, o simplemente procesar el error y devolverlo en un formato diferente. Para ello tenemos el operador **catchError**. Este método intermedio sólo se ejecuta si se produce algún error en el observable o en un método intermedio anterior.

Si queremos recuperarnos del error, debemos devolver un nuevo observable con datos correctos (podría ser otra llamada HTTP por ejemplo). Por otro lado, si queremos seguir con el error pero cambiando el formato, debemos devolver un observable con error (método **throwError**).

El siguiente esquema muestra un ejemplo de una llamada HTTP, y qué podríamos hacer si todo va bien o se produce un error (hay muchas posibilidades). En este caso el método `catchError` podría devolver datos válidos de alguna manera (recuperación de error) o formatear el error.

Finalmente, como dijimos anteriormente, si al método `subscribe` le llega un dato sin error se ejecutará la primera función, mientras que en caso de error se ejecutará la segunda en su lugar.

`Http.get<{prod: IProduct}>.pipe(map(...),catch(...)).subscribe(...);`



```
getProducts(): Observable<IProduct[]> {
  return this.http.get<{products: IProduct[]}>(this.productURL).pipe(
    map(response => response.products),
    catchError((resp: HttpResponse) => throwError(`Error obteniendo productos. Código de
servidor: ${resp.status}. Mensaje: ${resp.message}`))
  );
}
```

Otro método útil para la recuperación de errores es **retry**. Cuando llega la cadena de procesamiento a este método, si se detecta que ha habido un error, se reinicia el observable (se vuelve a ejecutar desde el principio) tantas veces como le indiquemos por parámetro. Si se supera el número de reintentos y sigue habiendo un error, entonces se deja pasar. Por ejemplo, para repetir la llamada HTTP 3 veces antes de darnos por vencidos (por si la red funcionara mal):

```
http.get(...).pipe( retry(3), map(...), catchError(...) ).subscribe(...);
```

1.4 Transformación de datos en el servicio de Angular

Muchas veces, el servidor nos devolverá una respuesta en formato JSON. Generalmente, los datos que queremos obtener se encontrarán dentro de ese objeto JSON, o también puede que queramos hacer ciertas transformaciones intermedias como pasar las fechas de string a objeto Date, etc. Para ello usamos el método **map**.

Además de crear interfaces para cada tipo de objetos que maneje el programa (productos, usuarios, etc.), es una buena idea crear una o varias interfaces para “mapear” la respuesta del servidor (y activar el autocompletado de TypeScript):

interfaces/responses.ts

```
export interface ResponseProducts {
  products: IProduct[];
}
```

En nuestro caso, queremos que la llamada HTTP devuelva el array de productos directamente en lugar de devolver el objeto `ResponseProducts` directamente tal como viene del servidor. Esta y cualquier otra transformación de los datos se puede hacer en el método `map`.

```
...
import { Observable, throwError } from 'rxjs';
import { map, catchError, retry } from 'rxjs/operators';

export class ProductService {




  private productURL = 'http://arturober.com/products-angular/products';

  constructor(private http: HttpClient) {}

  getProducts(): Observable<IProduct[]> {
    return this.http.get<ResponseProducts>(this.productURL).pipe(
      map(response => response.products),
      catchError((resp: HttpResponse) => throwError('Error obteniendo productos. Código de servidor: ${resp.status}. Mensaje: ${resp.message}'))
    );
  }
}
```

Finalmente, en la clase `ProductListComponent`, nos suscribiremos al observable y asignaremos el array de productos a mostrar cuando este sea devuelto. Ahora ya tenemos una funcionalidad más realista implementada.

```
ngOnInit() {
  this.productsService.getProducts()
    .subscribe(
      prods => this.products = prods, // Success function
      error => console.error(error), // Error function (optional)
      () => console.log('Products loaded') // Finally function (optional)
    );
}
```

Ocultar imágenes	Producto	Precio	Disponible	Puntuación
	Toshiba SSD Q300 480GB	€119.00	02/11/2016	★★★★★
	Sony PS4 500GB	€249.95	14/11/2016	★★★★☆
	DDR3 2x4GB 1600Mhz	€99.00	01/11/2017	★★★☆☆

1.5 Otros métodos HTTP (POST, PUT y DELETE)

La diferencia entre GET/DELETE y POST/PUT, es que estos últimos envían datos al servidor en el cuerpo de la petición (para insertar o modificar algo). Estos datos se envían como segundo parámetro después de la URL en la petición.

En el siguiente ejemplo vamos a cambiar la puntuación de un producto en el servidor. Llamaremos a un servicio `PUT` (modificación) con la URL `‘/products/rating/{idProduct}’`, y enviaremos la nueva puntuación en un objeto JSON: `{rating: newRating}`. Primero añadimos la interfaz con los datos de respuesta del servidor:

`interfaces/responses.ts`

```
export interface OkResponse {
  ok: boolean;
  error?: string; // Mensaje de error opcional si algo falla
}
```

Luego añadimos el método correspondiente en el servicio `ProductsService`:


```
export class ProductService {
  ...

  changeRating(idProduct: number, rating: number): Observable<boolean> {
    return this.http.put<OkResponse>(this.productURL + '/rating/' + idProduct,
      {rating: rating}).pipe(
        catchError((resp: HttpResponse) => throwError('Error cambiando puntuación!. Código de
servidor: ${resp.status}. Mensaje: ${resp.message}')),
        map(resp => {
          if (!resp.ok) { throw resp.error; }
          return true;
        })
      );
  }
}
```

Desde el componente **ProductItemComponent**, llamaremos al servicio cuando detectemos un cambio de la puntuación, y no actualizaremos la propiedad del producto hasta que el servidor no nos haya respondido. Recarga la página si quieres para comprobar que este cambio es permanente en el servidor.

```
export class ProductItemComponent implements OnInit {
  ...
  constructor(private productService: ProductService) {}
  ...
  changeRating(rating: number) {
    this.productService.changeRating(this.product.id, rating).subscribe(
      ok => this.product.rating = rating,
      error => console.error(error)
    );
  }
}
```

Por ahora no vamos a añadir un formulario para añadir un producto (lo haremos en un futuro, pero vamos a ver como enviaríamos al servidor un producto para insertarlo. En este caso, el servidor nos devolverá un objeto de respuesta con el producto insertado (ya que le habrá asignado una id o clave primaria, una URL con la imagen guardada en el servidor, etc.), que estaría definida por esta interfaz:

```
export interface ProductResponse {
  ok: boolean;
  product: IProduct;
  error?: string;
}
```

El método de ProductService encargado de llamar al servidor para añadir un producto usando POST, recibirá el objeto (IProduct) a insertar y lo enviará tal cual, devolviendo a su vez el producto (insertado) devuelto por el servidor:

```
addProduct(product: IProduct): Observable<IProduct> {
  return this.http.post<ProductResponse>(this.productURL, product).pipe(
    catchError((resp: HttpResponse) => throwError('Error insertando producto!. Código de
servidor: ${resp.status}. Mensaje: ${resp.message}')),
    map(resp => {
      if (!resp.ok) { throw resp.error; }
      return resp.product;
    })
  );
}
```

1.6 Manejar token de autenticación con Angular

Las aplicaciones cliente independientes se suelen ejecutar en entornos diferentes al servidor web donde están alojados los servicios web (otro dominio, un dispositivo móvil, etc.) lo cual generalmente no permite el uso de cookies, limitadas al mismo dominio del servidor. Por ello, la autenticación cuando trabajamos con servicios web se suele gestionar con el envío de un token al cliente, normalmente en formato [JWT](#).

Para demostrar que estamos autenticados, nuestra aplicación cliente debe enviar en cada petición al servidor, el token recibido en el login. Normalmente en una cabecera HTTP llamada **Authorization**. Esta cabecera se puede pasar como último parámetro al método HTTP que usemos en la petición así:

```
export class ProductService {
  ...
  getProducts(): Observable<IProduct[]> {
    let options = {
      headers: new HttpHeaders().set('Authorization', localStorage.getItem('token'))
    };
    return this.http.get(this.productsURL, options);
  }
  ...
}
```

1.7 Usando un interceptor para enviar el token

Los [Interceptores](#) son servicios que “interceptan” todas las peticiones HTTP que realizamos de forma automática y permiten alterar tanto la petición (antes de enviarla al servidor) como la respuesta que nos envíe. Un Interceptor se crea como un servicio que implementa la interfaz [HttpInterceptor](#). Esto nos obliga a crear un método llamado **intercept**.

Para el ejemplo, vamos a crear un interceptor que compruebe si tenemos guardado un token de autenticación, y lo envíe automáticamente en cada petición HTTP en el caso de tenerlo. Podemos crear el servicio en una carpeta llamada interceptors:

ng g service auth-interceptor (quitar sufijo Service en la clase)

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs';

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

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('token'); // Obtenemos el token del LocalStorage

    if (token) {
      // Clonamos la petición y le añadimos el encabezado Authorization
      const authReq = req.clone({headers: req.headers.set('Authorization', token)});
      // Enviamos al servidor la petición clonada con el token
      return next.handle(authReq);
    }
    return next.handle(req); // Si no hay token, enviamos petición original
  }
}
```

El método intercept recibe 2 parámetros:

- **req: HttpRequest** → Esta es la petición original que se envía desde cualquier punto de nuestra aplicación Angular. Para modificarla debemos **clonarla** añadiendo nuevas cabeceras HTTP, etc.
- **next: HttpHandler** → Si sólo tenemos un interceptor en la aplicación, **next** representa al servicio interno de Angular que envía la petición al servidor (y recibe su respuesta). Si tuviéramos más interceptores, sería el siguiente de la lista (se ejecutan en cadena, uno detrás de otro).

Para indicar a Angular que las peticiones deben pasar por el interceptor, debes declararlo como un servicio más (dentro del array **providers** en el módulo de la aplicación), pero usando una sintaxis especial (usando un objeto):

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

@NgModule({
  providers: [
```



```
{  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthInterceptorService,  
  multi: true,  
}  
],  
})  
export class AppModule {}
```

Implementar un interceptor para tareas habituales en las peticiones HTTP como enviar el token de autenticación te permiten automatizar dicha tarea en un sólo sitio de la aplicación, en lugar de tener que hacerlo en todos los métodos que realizan dichas peticiones al servidor.