6 - Http

> Comunicaciones http en Angular

Las comunicaciones *http* son una pieza fundamental del desarrollo web, y en **Angular** siempre han sido potentes y fáciles. ¿Siempre?, bueno cuando apareció Angular 2 echábamos en falta algunas cosillas... y además la librería *RxJS* y sus *streams* son intimidantes para los novatos.

Pero en la versión Angular 7 **consumir un servicio REST** puede ser cosa de niños si aprendes a jugar con los *observables* y los servicios de la librería `@angular/common/http`. Conseguirás realizar **comunicaciones http asíncronas en Angular**.

Partiendo de la aplicación tal cómo quedó en Servicios inyectables en Angular. Al finalizar tendrás una aplicación que almacena y recupera los datos consumiendo un servicio REST usando las tecnologías de Angular Http.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/6-http
>
> > Tienes una versión desplegada operativa para probar Angular Board

1. El servicio HttpClient

Como demostración vamos a consumir un API pública con datos de cotización de monedas. Crearé un módulo y un componente en el que visualizar las divisas y después las transformaremos para guardarlas en un servicio propio.

```
ng g m 6-http/rates --routing true
ng g c 6-http/rates/rates
```

1.1 Importación y declaración de servicios

La librería `@angular/common/http` trae el módulo `HttpClientModule` con el servicio inyectable `HttpClient`. Lo primero es importar dicho módulo.

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

@NgModule({
  declarations: [RatesComponent],
  imports: [HttpClientModule]
})
export class RatesModule { }
```

En tu componente tienes que reclamar la dependencia al servicio para poder usarla. Atención a la importación pues hay más clases con el nombre `HttpClient`. Debe quedar algo así:

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

@Component({
  selector: 'app-rates',
  templateUrl: './rates.component.html',
  styles: []
})
export class RatesComponent implements OnInit {
  constructor(private httpClient: HttpClient) {}

  ngOnInit() {}
}
```

A partir de este momento sólo queda invocar los métodos REST en la propiedad `this.http`.

1.2 Obtención de datos

Para cada verbo *http* tenemos su método en el servicio `HttpClient`. Su primer parámetro será la *url* a la que invocar. Empecemos por el `get` que automáticamente solicita y devuelve objetos *JSON* desde un API. Por ejemplo para obtener las últimas cotizaciones de las principales divisas lo haremos así:

```
export class RatesComponent implements OnInit {
  private urlapi
    = 'https://api.exchangeratesapi.io/latest';
  public currentEuroRates: any = null;

  constructor(private httpClient: HttpClient) {}

  ngOnInit() {
    this.getCurrentEuroRates();
  }

  private getCurrentEuroRates() {
    const currencies = 'USD,GBP,CHF,JPY';
    const url = `${this.urlapi}?symbols=${currencies}`;
    this.httpClient
      .get(url)
      .subscribe(apiData => (this.currentEuroRates = apiData));
  }
}
```

> El método *get* retorna un objeto observable. Los observables *http* han de consumirse mediante el método *subscribe* para que realmente se lancen. Dicho método *subscribe* admite hasta tres *callbacks*: `susbcribe(data, err, end)` para que se ejecuten en respuesta a eventos. En este ejemplo solo hemos usado el primero.

La presentación en la vista sólo tiene que acceder a la propiedad dónde se hayan cargado las respuestas tratadas en el *callback* de la suscripción.

```
<h2> Currency Rates. </h2>
<h3> From Euro to the world </h3>
<pre>{{ currentEuroRates | json }}</pre>
```

1.3 Envío de datos

Supongamos que, una vez recibidas las cotizaciones, pretendemos transformarlas y almacenarlas en otro servicio. El envío en este caso será con el método *post* al que se le pasará la ruta del *end point* y el objeto *payload* que se enviará al servidor.

Vamos a agregar una propiedad y un par de métodos al `rates-component.ts`. La idea es obtener un array de cotizaciones aa partir del objeto previo, y guardarla una por una.

```
export class RatesComponent implements OnInit {
  private myRatesApi
    = 'https://api-base.herokuapp.com/api/pub/rates';

  public postRates() {
    const rates = this.transformData();
    rates.forEach(rate =>
      this.httpClient
        .post(this.myRatesApi, rate)
        .subscribe()
    );
  }

  private transformData() {
    const current = this.currentEuroRates.rates;
    return Object.keys(current).map(key => ({
      date: this.currentEuroRates.date,
      currency: key,
      euros: current[key]
    }));
  }
}
```

> Atención a los métodos `subscribe()`. Aunque vayan vacíos son imprescindibles para que se ejecute la llamada.

En la vista no hay gran cosa que hacer, salvo agregarle un botón para iniciar el proceso:

```
<input value="Save Rates" type="button" (click)="postRates()" />
```

1.4 Actualización de datos

Un par de ejemplo más para acabar de entender la mecánica básica de `HttpClient`. Podemos fijar el tipo de datos esperado en cualquier llamada. De hecho es recomendable que tengas un interfaz para cada respuesta esperada.

En este tutorial no se ha hecho y no quedamos con el `any`, pero al menos distinguimos entre objetos y arrays. Esto es lo que añado al `RatesComponent` para que muestre los datos guardados en mi API.

```
export class RatesComponent implements OnInit {
 private myRatesApi
  = 'https://api-base.herokuapp.com/api/pub/rates';
 public myRates: any[] = null;

 public getMyRates() {
    this.httpClient
      .get<any[]>(this.myRatesApi)
      .subscribe(apiResult => (this.myRates = apiResult));
  }
}
```

Y en la vista, un nuevo botón y una nueva expresión.

```
<input value="Refresh" type="button" (click)="getMyRates()" />
<pre>{{ myRates | json }}</pre>
```

Por último, en plan repaso, un ejemplo de método para borrar.

```
public deleteMyRates() {
  this.httpClient
    .delete(this.myRatesApi)
    .subscribe();
}
```

Y su botón en en la vista.

```
<input value="Delete Rates" type="button" (click)="deleteMyRates()" />
```

Y hasta aquí lo básico de comunicaciones *http*. ¿Fácil verdad?. Pero la vida real raramente es tan sencilla. Si quieres enfrentarte a algo más duro debes prepararte y dominar los observables *RxJS*.

Lo que viene a partir de ahora requerirá tiempo y concentración. Si continúas adelante esto ya no será *your grandpa´s http anymore*.

2 Observables

Las **comunicaciones** entre navegadores y servidores son varios órdenes de magnitud más lentas que las operaciones en memoria. Por tanto deben realizarse de manera asíncrona para garantizar una buena experiencia al usuario.

Esta experiencia no siempre fue tan buena para el programador. Sobre todo con las primeras comunicaciones *AJAX* basadas en el paso de funciones *callback*. La aparición de las *promises* mejoró la claridad del código, y ahora con los *Observables* tenemos además una gran potencia para manipular la **información asíncrona**.

> El patrón `Observable` fue implementado por Microsoft en la librería *Reactive Extensions* más conocida como `RxJs`. El equipo de Angular decidió utilizarla para el desarrollo de las comunicaciones asíncronas. Esta extensa librería puede resultar intimidante en un primer vistazo. Pero con muy poco conocimiento puedes programar casi todas las funcionalidades que se te ocurran.

Lo primero es importar el código, de forma similar a cualquier otra clase o función. Por ejemplo para empezar basta con `import { Observable } from "rxjs/Observable";`. Tendremos la clase usada por angular para observar el respuesta *http*.

Pero esta es una clase genérica donde sus instancias admiten la manipulación interna de tipos más o menos concretos. Por eso ves en el ejemplo que algunas funciones retornan el tipo esperado `: Observable<MyClass>`, o si no saben que tipo esperar se conforman con `: Observable<any>`.

En cualquier caso, **toda operación asíncrona retornará una instancia observable** a la cual habrá que subscribirse para recibir los datos o los errores, cuando termine. Aunque a veces no se verá el *subscribe*...

2.1 Async

Para probar otras formas de presentar datos recibidos desde un API, voy a crear un nuevo componente. El `ObseratesComponent`, mezcla de *observables* y *rates*.

```
ng g c 6-http/rates/obserates
```

Y en su vista HTML usaré una función propia de Angular llamada `async`. Dicha función actúa como un *pipe* en una expresión. Igual que el `| json`. Pero a su izquierda espera que le den algo a lo que suscribirse; espera un observable.

```
<h2> Currency Observable Rates. </h2>
<h3> From Euro to the $ world </h3>
<pre>{{ currentEuroRates$ | async | json }}</pre>}
```

En este caso uso la propiedad `currentEuroRates$` finalizada en **$** por convenio. Esa propiedad se rellena en el controlador con el método `get`, no con los datos futuros, si no con el propio *observable*.

```
private ratesApi
  = 'https://api.exchangeratesapi.io/latest';
public currentEuroRates$: Observable<any> = null;

constructor(private httpClient: HttpClient) {}

ngOnInit() {
  this.getCurrentEuroRates();
}

private getCurrentEuroRates() {
  const currencies = 'USD,GBP,CHF,JPY';
  const url = `${this.ratesApi}?symbols=${currencies}`;
  this.currentEuroRates$ = this.httpClient.get(url);
}
```

> Al utilizar el *pipe async* ya es necesaria la suscripción en código. La propia función del framework se ocupa de ello. Por tanto la llamada se realiza igualmente aunque no veamos la suscripción.

Esta es la manera recomendada de consumir datos desde un API. Definir la llamada en el controlador y pasarle el observable a la vista para que lo muestre cuando obtenga los datos.

2.2 Pipe

Tuberías en RxJS .pipe()

Los datos devueltos raramente vienen en el formato preciso para usar en la vista. Con frecuencia hay que transformarlos al vuelo en cuanto se reciben. Recordemos que ***HttpClient* no devuelve los datos tal cual sino un *stream* de estados** de dichos datos. La manipulación será sobre el *stream* no directamente sobre los datos; y, claro, para manipular un torrente hay que encauzarlo en tuberías.

Aquí es donde aparece el método `.pipe(operator1, operator2...)` aplicado a un observable. Suele hacerse en lugar, o antes, del `.susbcribe(okCallback, errCallback)`. Este método canaliza una serie de operadores predefinidos que manipulan el chorro de estados observados.

El operador más utilizado es `map(sourceStream => targetStream)`. Este operador recibe una función *callBack* que será invocada ante cada dato recibido. Esa función tienen que retornar un valor para sustituir al actual y así transformar el contenido del chorro.

```
public myRates$: Observable<any[]> = null;
private getCurrentEuroRates() {
  const url = `${this.ratesApi}?symbols=USD,GBP,CHF,JPY`;
  this.currentEuroRates$ = this.httpClient.get(url);
  this.myRates$ = this.currentEuroRates$.pipe(map(this.transformData));
}
private transformData(currentRates) {
  const current = currentRates.rates;
  return Object.keys(current).map(key => ({
    date: currentRates.date,
```

```
      currency: key,
      euros: current[key]
    }));
  }
```

> En este ejemplo partimos de nuevo de un objeto recibido y lo queremos ver como un *array* de objetos. Para ello lo transformamos usado el operador `map`. Este operador ha de importarse de `rxjs/operators` y aplicarse a un observable dentro de su método `.pipe()`. Es el más sencillo y uno de los más utilizados: recibe y emite datos dentro de un stream de eventos observables. Nada que ver, salvo el nombre, con la sencilla función `array.map(callback)`, que recibe y devuelve datos estáticos.

```
<pre>{{ myRates$ | async | json }}</pre>
```

Por lo demás el consumo se hace igual... pero... tendremos que ver más operadores para solucionar algunos inconvenientes.

2.3 Operators

El código anterior funciona, pero resulta que al haber dos funciones async suscritas provoca que la llamada original se realice dos veces. esto es así porque el segundo observable `myRates$` es una canalización del primero `currentEuroRates$`.

Estos y oros problemas se solucionan usando operadores. Vamos a conocer un par de ellos más y veremos como `pipe(op1, op2, opn)` los ejecuta a todos en orden.

El operador `share()` permite compartir el resultado de una primera llamada con subsiguientes suscriptores. Evitando de ese modo la repetición de costosas peticiones http.

```
private getCurrentEuroRates() {
const url = `${this.ratesApi}?symbols=USD,GBP,CHF,JPY`;
  this.currentEuroRates$ = this.httpClient.get(url)
      .pipe(share());
  this.myRates$ = this.currentEuroRates$
      .pipe(
        tap(d=>console.log(d)),
        map(this.transformData),
        tap(t=>console.log(t))
      );
}
```

> El operador `tap(callback)` es similar en nombre al `map()`. Pero la gran diferencia es que está pensado para no manipular los datos que recibe. Los usa y puede causar otros efectos colaterales, pero nunca modifica el propio stream. Es muy utilizado para inspeccionar o auditar el flujo de otros operadores.

Reconozco que en un primer vistazo este código pueda resultar complejo. Tómate tu tiempo. Fíjate en los datos de entrada y salida de cada función. Esto es solo el principio del trabajo con la librería *RxJS* y la manipulación de *streams de eventos observables*.

3. Interceptores

Los interceptores tienen un nombre intimidante pero un propósito sencillo y muy útil: **interceptar todas las comunicaciones http** y ejecutar código personalizado para cada uno. Por ejemplo un gestor centralizado de errores http o el control de los tokens de seguridad de la aplicación.

Pero antes de eso habrá que aprender unos conceptos básicos. Vamos a ver un ejemplo sencillo que audite todas las llamadas http. Todo empieza con un servicio:

```
ng g s rates/AuditInterceptor
```

3.1 La interfaz HttpInterceptor

Al servicio genérico recién creado hay que hacerle cumplir una interfaz `HttpInterceptor` que viene con `HttpClientModule`. Esta interfaz solo necesita un método, el `intercept(req, next)` pero sus tipos e implementación mínima la hacen complicada de entender a la primera.

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

@Injectable({
  providedIn: 'root'
})
export class AuditInterceptorService implements HttpInterceptor {
  public intercept( req: HttpRequest<any>, next: HttpHandler )
    : Observable<HttpEvent<any>> {
    return next.handle(req);
  }

  constructor() { }
}
```

Todo este código para nada. Porque eso es lo que hace, nada. Eso sí, tampoco rompe ni bloquea nada. Vayamos por partes, primero los parámetros, luego el tipo de la respuesta y por último la implementación

- `req: HttpRequest<any>` puntero a la petición en curso
- `next: HttpHandler` puntero a la siguiente función que maneje la petición
- `: Observable<HttpEvent<any>>` retornamos un stream de eventos http para cada petición
- `return next.handle(req);` que el siguiente procese la petición, sin hacerle nada en absoluto

> Para entenderlo mejor puede ser útil la siguiente analogía. Cuando usas httpClient.get() es como si pides algo a Amazon y te suscribes, es decir esperas el paquete. Pasado el tiempo el paquete llegará o no llegará, pero ya no lo gestionas tú. Con los interceptores es como si espiases cada proceso de tu pedido: stock, picking, packaging, shipping... Cada pedido es tratado en una sucesión de eventos. Con un interceptor observas !todos los eventos de todos los pedidos!

3.2 Inversión del control vía token

Tenemos un servicio que cumple una interfaz compleja. Pero dicho servicio debe ser proveído en algún módulo antes de ser reclamado como dependencias por alguien. Pero ¿por quién?

Técnicamente lo necesita el propio `HttpClient` del framework. Pero, obviamente, no pueden reclamar por tipo una clase que acabo de inventarme yo. Adelante con la **inversión del control**.

Realmente `HttpClient` depende de algo que por convenio llaman token de tipo `HTTP_INTERCEPTORS`. Nuestro trabajo consiste en que cuando reclame su dependencia, le demos la nuestra. El típico gato por liebre. Así en nuestro módulo pondremos la siguiente configuración.

```
providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: AuditInterceptorService,
    multi: true
  }
]
```

El parámetro `multi:true` en este caso le indica que puede haber más de un interceptor. Concretamente debe añadirlo a la lista y admitir más. Hecho esto, sobraría el `providedIn: 'root'` autogenerado en el decorador del servicio.

3.3 Un auditor de llamadas

Pues ya estamos listos para aportar algo de funcionalidad. Nuestro objetivo es escribir en el *log* un texto para cada llamada terminada y el tiempo que le tomó. La idea es aprovechar que todo es un *stream* observable y canalizarlo en una tubería con una serie de operadores.

```
export class AuditInterceptorService implements HttpInterceptor {
  constructor() {}

  public intercept(req: HttpRequest<any>, next: HttpHandler){
    const started = Date.now();
    return next.handle(req).pipe(
      filter((event: HttpEvent<any>) => event instanceof HttpResponse),
      tap((resp: HttpResponse<any>) => this.auditEvent(resp, started))
    );
  }

  private auditEvent(resp: HttpResponse<any>, started: number) {
```

```
      const elapsedMs = Date.now() - started;
      const eventMessage = resp.statusText + ' on ' + resp.url;
      const message = eventMessage + ' in ' + elapsedMs + 'ms';
      console.log(message);
    }
  }
```

El operador `filter(any=>bool)` se usa para descartar eventos que no cumplan unos criterios. En mi caso sólo me interesan los eventos de recepción de la petición, y no necesito los intermedios. Uso de nuevo el `tap(callback)` para hacer cosas con los datos sin modificarlos en absoluto. En este caso los envío al método `auditEvent` para que lo saque por consola. Listo: un auditor para todas las llamadas.

Ya tenemos el programa comunicado por *http* con un servidor; aunque por ahora de forma anónima y sin ninguna seguridad. Con el conocimiento actual de los observables, del *httpClient* y de los interceptores ya estamos cerca de resolverlo. Esto lo veremos en los próximos temas.