Práctica Angular – Parte 5

Tic-Tac-Toe

Objetivos

- Ampliar nuestra base teórica para así poder dotar de mayor funcionalidad la aplicación del juego de las tres en rayas.
- Reforzar el concepto de enrutamiento en una aplicación SPA Angular, así como los elementos del framework que lo hacen posible: módulos, objetos, etc.
- Analizar el funcionamiento del servicio HTTP de Angular para la comunicación con terceros.

Introducción

Antes de nada vamos a recapitular lo que hemos visto de Angular hasta el momento:

- Gestión de componentes: Tanto componentes individuales como las interacciones necesarias para que dos o más componentes colaboren para casos de uso más complejos (Interpolación, bindings, @Input(), @Output(), servicios).
- Gestión de módulos: Para tener el código bien organizado vimos el funcionamiento de los módulos.
- Gestión de servicios: Vimos la importancia de los servicios para compartir información entre componentes y para manejar el estado de la aplicación (muy importantes los Observables para esto último). También vimos que cuando un componente requiere un servicio, Angular lo inyecta en el componente mediante el mecanismo de inyección de dependencias. También comentamos que generalmente los servicios siguen el patrón Singleton (una única instancia del servicio es compartida por todos los componentes).
- Integración de elementos en un proyecto: Para asimilar todo lo anterior realizamos la aplicación de las tres en rayas.

Ahora, vamos a aprender el funcionamiento en Angular del mecanismo del enrutamiento para permitir la navegación a través de una aplicación SPA (Single Page Application) y a realizar

conexiones a servicios del lado del *backend* para así poder recuperar información del servidor o para enviársela.

Conceptos fundamentales sobre enrutamiento

El enrutamiento permite la navegación del usuario a través de la aplicación. La navegación se produce en tres casos:

- Cuando el usuario escribe una URL en la barra de direcciones del navegador.
- Al hacer clic sobre un enlace de una página.
- Al pulsar sobre los botones 'atrás' o 'adelante' del navegador.

Gracias al enrutamiento podemos manejar estas interacciones del usuario y controlar qué vista se le debe presentar al usuario en todo momento.

Configurar y usar rutas

Lo primero es asegurarnos que en el fichero **src/index.html** está definida la etiqueta **<base>** y que el atributo **href** apunta a la raíz de la aplicación web:

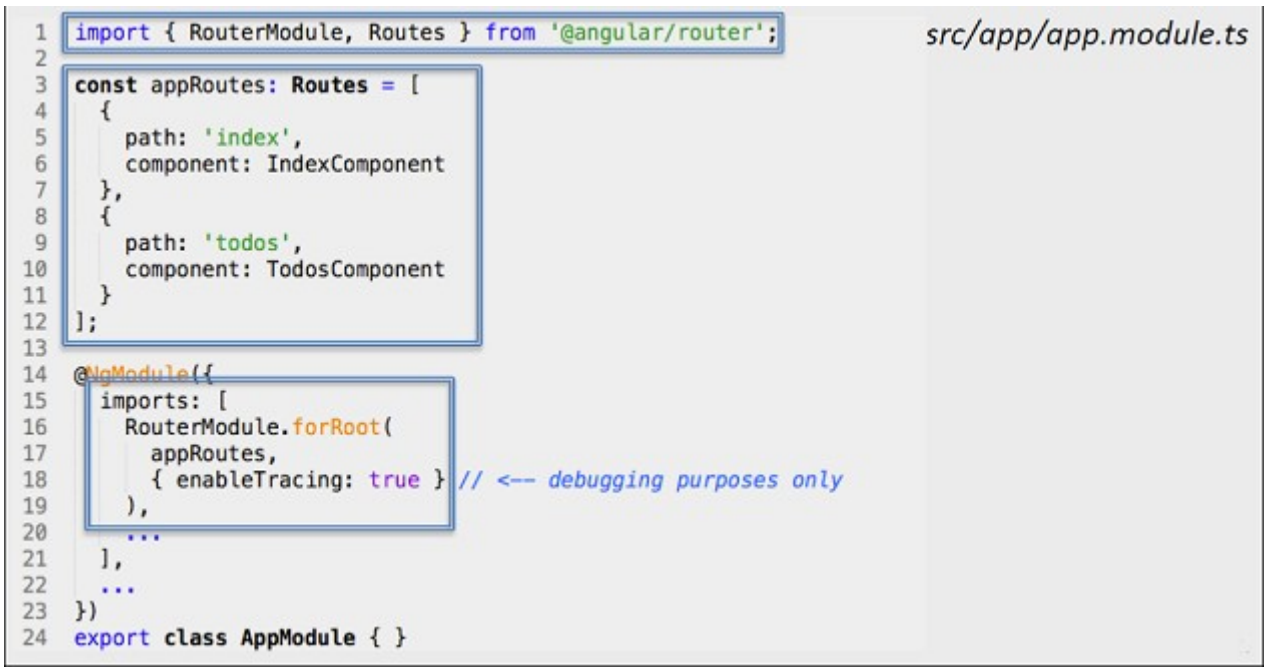
```
...
<head>
   <base href="/">
...
```

Lo anterior es el comportamiento predeterminado cuando creamos una aplicación mediante Angular CLI, por tanto, no tendríamos que preocuparnos por esto en un principio (aunque si no está definido de esta manera entonces el enrutamiento no funcionará).

A continuación, hemos de configurar las rutas que queremos gestionar en el módulo raíz de la aplicación (AppModule). Esto implica tres cosas:

- Importar los módulos y objetos necesarios para poder usar enrutamiento.
- Definir el array con las rutas de la aplicación (mapeo entre url y componente a mostrar).
- Añadir el array de rutas a la sección *imports*.

src/app/app.module.ts

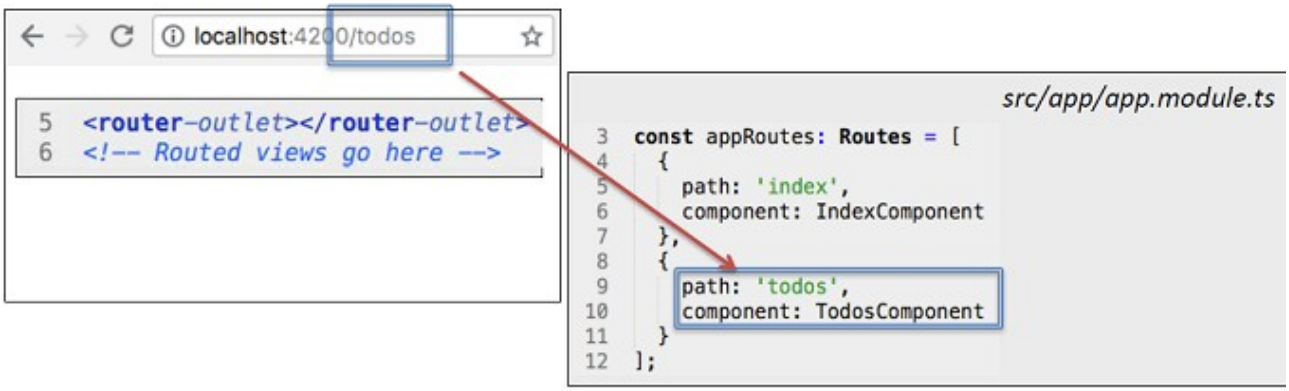


Nota: en la figura anterior mediante “enableTracing” estamos indicando que queremos que se muestre en la consola del navegador información de debug.

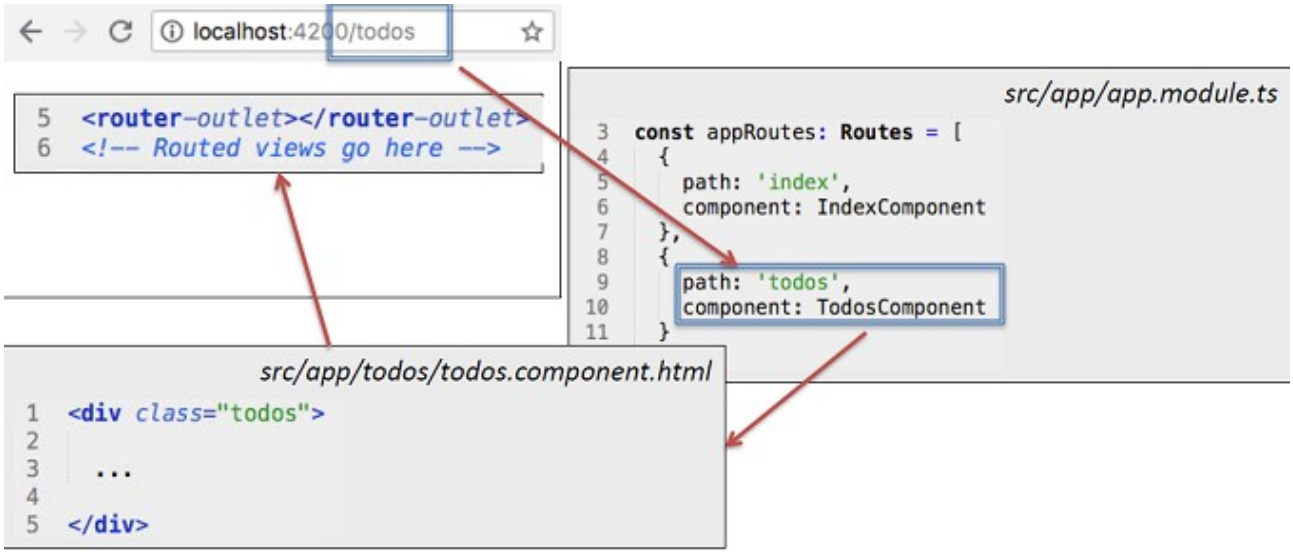
Ejemplo

Veamos un ejemplo en el que tomaremos las rutas mostradas en la figura anterior. El usuario pulsa un enlace para obtener una lista de tareas pendientes de realizar (todos):

Entonces se produce “match” entre la URL y uno de los elementos del array de rutas:



Y por tanto, Angular mostrará la plantilla del componente TodosComponent justo debajo donde tenemos el tag **<router-outlet></router-outlet>**:



Ejemplos de rutas

Podemos configurar rutas para cubrir cualquier caso, por muy complejo que este sea. A continuación se muestran algunos ejemplos:

Ejemplo 1: Se quiere pasar al componente el id de tarea número 31 (notad los ‘:’ antes del identificador).

Navegador (URL): http://localhost:4200/todos/**31**

```
appRoutes: [
      …
      {
            path: ‘todos/:id’, component: TodoDetailComponent
      }
      …
]
```

Ejemplo 2: Se quiere pasar al componente datos adicionales (propiedad “data” de Angular)

Navegador (URL): http://localhost:4200/todos/

```
appRoutes: [
      …
      {
            path: ‘todos’, component: TodosComponent, data: {titulo: ‘Lista de tareas’}
```

```
      }
      …
]
```

Ejemplo 3: Redirecciones. Cuando el usuario navega a "/", Angular redirige a "/index" (la estrategia por defecto de búsqueda, pathMatch, es "prefix" lo que comprueba si la ruta comienza por el path especificado. Mediante el valor "full" decimos que toda la ruta ha de coincidir con el valor especificado).

Navegador (URL): http://localhost:4200/

```
appRoutes: [
      …
      {
            path: '', pathMatch: 'full', redirectTo: '/index'
      }
      …
]
```

Ejemplo 4: Componente por defecto. Toda ruta que no encaje con los paths anteriores provocará que se visualice PageNotFoundComponent. El orden en como establecemos la rutas importa.

Navegador (URL): http://localhost:4200/abc

```
appRoutes: [
      …
      {
            path: '', pathMatch: 'full', redirectTo: '/index'
      },
      {
            path: '**', component: PageNotFoundComponent
      }
]
```

¿Cómo acceder desde un componente a los parámetros o datos de la URL?

Esto es posible gracias a **ActivatedRoute**, un servicio de Angular que podemos utilizar en el constructor de nuestros componentes, ya que Angular lo tiene siempre instanciado y listo para ser inyectado donde se necesite. ActivatedRoute contiene toda la información de la URL.

Por ejemplo, si tenemos definida una ruta como la siguiente, que incluye información sobre el "title" en la propiedad "data":

```
const appRoutes: Routes = [
  {
    path: 'todos/:id',
    component: TodoDetailComponent,
    data: {title: 'List of TODOs'}
  }
];
```

Entonces, en el constructor de TodoDetailComponent podemos definir el objeto ActivatedRoute, con lo que tendremos acceso tanto a la URL, como a los parámetros (id en este caso) y contenido extra (data… *title* en este caso):

```
import { ActivatedRoute } from '@angular/router';
...

export class TodoDetailComponent {
  constructor(route:ActivatedRoute) {
    console.log('Current URL', route.snapshot.url);
    console.log('Data value', route.snapshot.data);
    console.log('Route params', route.snapshot.params);
  }
}
```

El objeto ActivatedRoute tiene una serie de propiedades que son del tipo **Observable**, por lo que podemos suscribirnos a sus cambios, aunque si solamente queremos acceder a los valores en curso lo podemos hacer fácilmente a través de la propiedad **snapshot**, lo cual no requiere suscripción.

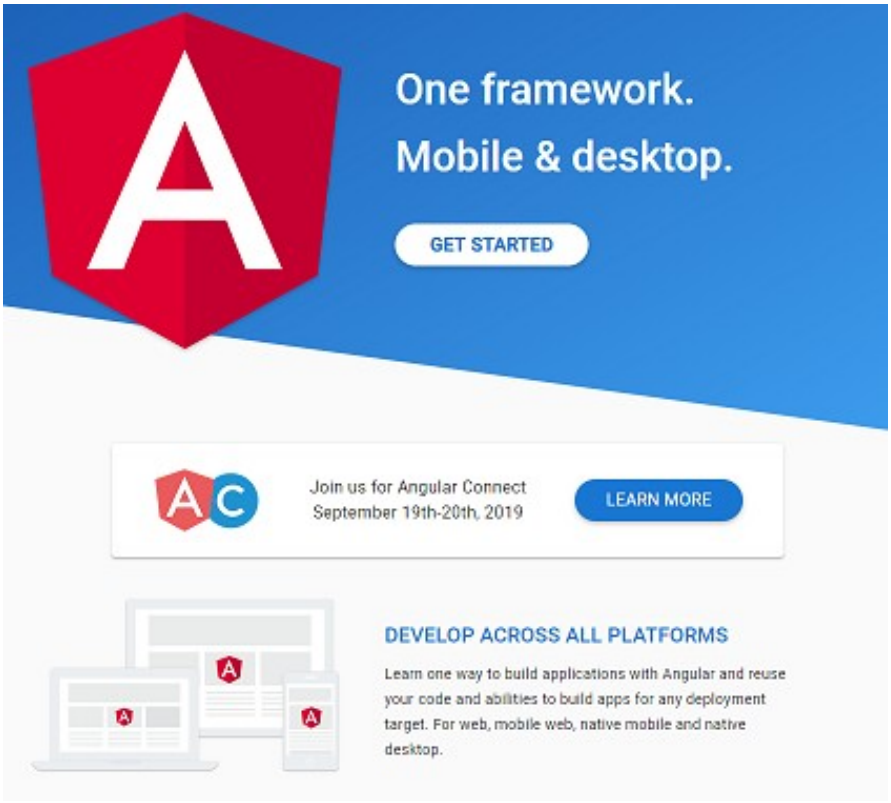
Por otro lado, cuando lo que necesitamos es gestionar enlaces (y no la URL) en una página web, es muy recomendable usar las directivas **routerLink** y **routerLinkActive**.

```
<h1>Todos menu</h1>
<nav>
  <a routerLink="/index" routerLinkActive="active">Home</a>
  <a routerLink="/todos" routerLinkActive="active">TODO's list</a>
</nav>
<router-outlet></router-outlet>
```

Al utilizar routerLinkActive con la etiqueta <a>, Angular añade la clase CSS “active”, consiguiendo que el elemento routerLink se muestre seleccionado al recibir el clic.

Conectar una aplicación Angular a un *backend*

Angular es un framework de *front*, esto es, se ejecuta en navegadores y escritorios, tanto en ordenadores como en dispositivos móviles. En el caso de ejecutarse en un navegador se descargará el contenido del servidor web solamente la primera vez, y las ejecuciones de la aplicación se producirán en el navegador del usuario. Por tanto, en un principio el papel del servidor es servir de repositorio.



No obstante, hay muchas situaciones en las que una aplicación *front* necesita comunicarse con los servicios ofrecidos por una aplicación *backend*. Por ejemplo:

- Obtener información de terceros: servicio meteorológico, noticias, *ERP*, etc.
- Compartir información con otros clientes/aplicaciones.
- Gestionar la persistencia: información relativa al alta de un usuario, modificación del perfil de un usuario, etc.

Normalmente lo interesante de comunicarse con un *backend* es que éste a su vez suele estar conectado a un sistema gestor de bases de datos o similar, con lo que en conjunto nos permite realizar aplicaciones dinámicas donde la información cambia con el paso del tiempo.

En las aplicaciones *front* la conexión con el *backend* normalmente se realizan mediante el protocolo HTTP/HTTPS. Los navegadores implementan la interfaz *XmlHttpRequest*, lo cual permite utilizar

la tecnología AJAX (Asynchronous Javascript And XML) y así poder llevar a cabo aplicaciones *SPA* (Single Page Application).

Angular dispone del objeto *HttpClient* que no es más que un *wrapper* de la interfaz *XmlHttpRequest* para hacer su uso más sencillo (expone un API más simple y cómodo). El objeto *HttpClient* lo tenemos disponible en @angular/common/http

Configurar la aplicación para usar servicios de backend

Lo primero es añadir el módulo HttpClientModule a la sección de los "imports" del módulo raíz de la aplicación (AppModule):

app/app.module.ts

```
1  ... // other imports
2  import { HttpClientModule } from '@angular/common/http';
3
4  @NgModule({
5    declarations: [
6      AppComponent
7    ],
8    imports: [
9      BrowserModule,
10     HttpClientModule
11     )
12   ],
13   providers: [],
14   bootstrap: [AppComponent]
15 })
16 export class AppModule { }
```

Una vez realizado lo anterior ya podemos inyectar el objeto HttpClient en nuestra aplicación. No obstante, en lugar de usar el objeto HttpClient directamente allá donde se necesite, se considera una buena práctica encapsularlo en un servicio propio para gestionar ahí toda la lógica HTTP, es decir, usar los métodos que ofrece el objeto HttpClient dentro de nuestro servicio. De esta manera conseguimos separar la presentación de la información del acceso a los datos. La siguiente figura muestra precisamente esto: un servicio llamado MyHttpService en el que Angular inyecta en su constructor el objeto HttpClient:

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

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

¿Qué métodos ofrece el objeto HttpClient?

Los métodos habituales para realizar un CRUD: GET, POST, PUT, DELETE, etc. Veamos ejemplos con los principales usos de estos métodos:

get(): Es el método adecuado para recuperar información de un servicio externo.

- Se le pasa la URL a la que se debe acceder
- Devuelve un objeto Observable

```
export class MyhttpService {

  constructor(private http: HttpClient) {}

  let backendUrl = 'http://myApi.com/myResource'

  getDataFromService () {
    return this.http.get(backendUrl);
  }
}
```

Notad que hemos encapsulado el método get() en un método propio llamado getDataFromService(). Este método getDataFromService() devolverá un Observable, ocultando la lógica de acceso HTTP.

Desde el componente donde necesitemos los datos tenemos que inyectar en el constructor nuestro servicio MyHttpService y suscribirnos al Observable que retorna el método getDataFromService(). De esta manera conseguimos que los componentes que necesiten gestionar datos con el *backend* sean agnósticos del protocolo HTTP, ya que toda esa lógica queda centralizada en nuestro servicio MyHttpService:

my.component.ts

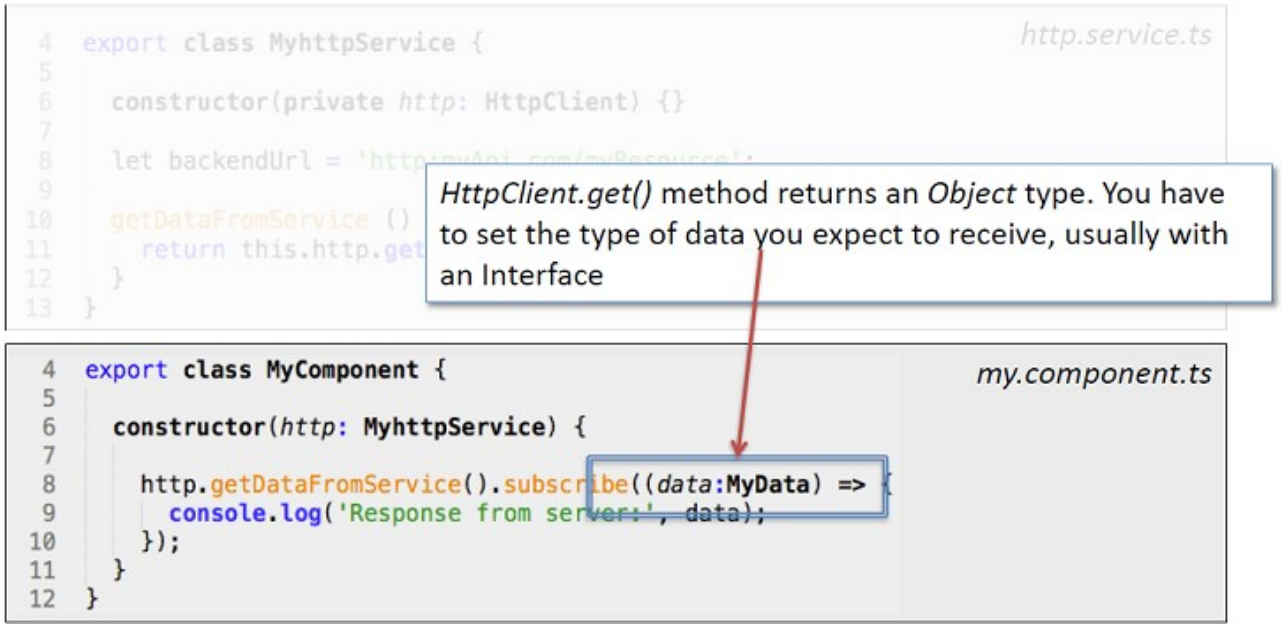
```
export class MyComponent {

  constructor(http: MyhttpService) {

    http.getDataFromService().subscribe((data:MyData) => {
      console.log('Response from server:', data);
    });
  }
}
```

Un detalle a tener en cuenta es que el tipo de dato que nos va a devolver el servicio si no indicamos nada va a ser “object”. Por tanto, normalmente nos interesará hacer un “cast” al tipo de datos deseado, que normalmente es una interfaz (MyData en este ejemplo).

La siguiente figura ilustra este hecho:



Con el código del ejemplo anterior lo que conseguimos es recibir el “body” del objeto “response”. Si esto no es suficiente para nosotros y necesitamos acceder tanto a la cabecera como al “body” de la respuesta, entonces hemos de añadir un parámetro en la llamada al método get():

```
export class MyhttpService {

  constructor(private http: HttpClient) {}

  let backendUrl = 'http://myApi.com/myResource'

  getDataFromService () {
    return this.http.get(backendUrl, { observe: 'response' });
  }
}
```

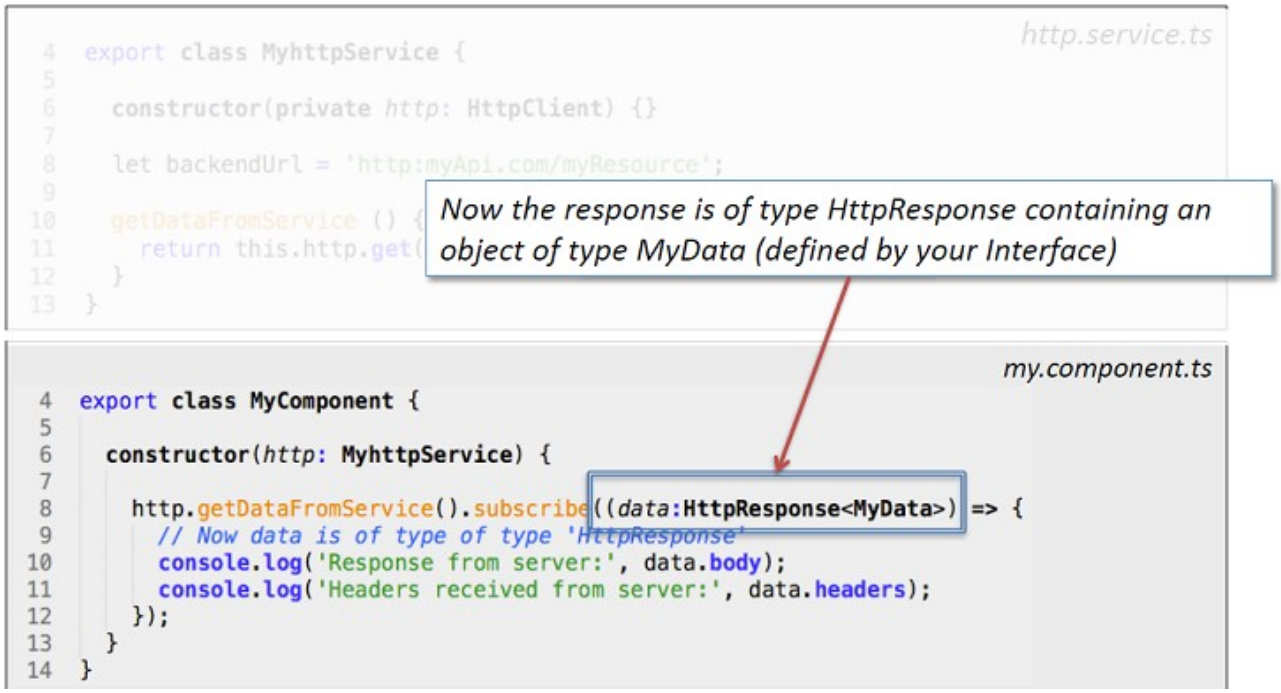
Y en el componente donde se inyecte nuestro servicio, dado que ahora se retorna un objeto HttpResponse en lugar de “object”, hemos de hacer lo siguiente:

```
export class MyComponent {

  constructor(http: MyhttpService) {

    http.getDataFromService().subscribe((data:HttpResponse<MyData>) => {
      // Now data is of type of type 'HttpResponse'
      console.log('Response from server:', data.body);
      console.log('Headers received from server:', data.headers);
    });
  }
}
```

Notad cómo se ha de indicar el *cast* de la información obtenida:



Gestión de errores

Por la naturaleza de las llamadas asíncronas siempre necesitaremos controlar posibles errores que se puedan producir en la comunicación entre cliente y servidor. Recordemos que estamos accediendo a un servicio externo, el cual puede que esté disponible o puede que no lo esté, o sencillamente que estemos en una ubicación con nulo o limitado acceso a Internet.

Para controlar cuando algo ha ido mal, HttpClient dispone de la propiedad “error”, cuyo tipo es HttpErrorResponse. Notad cómo la siguiente figura incluye el control de errores:

```
export class MyComponent {

  constructor(http: MyhttpService) {

    http.getDataFromService().subscribe((data:HttpResponse<MyData>) => {
      // Now data is of type of type 'HttpResponse'
      console.log('Response from server:', data.body);
      console.log('Headers received from server:', data.headers);
    }, error => {
      // error is of type HttpErrorResponse
      console.log('Error code: ', error.status);
      console.log('Error text: ', error.statusText);
    });
  }
}
```

Resto de métodos CRUD

Cómo se ha comentado anteriormente, el objeto HttpClient ofrece otros métodos además de get(). La siguiente figura muestra un ejemplo de como encapsular los métodos post() y delete():

http.service.ts

```
2  import { HttpHeaders } from '@angular/common/http';
3
4  export class MyhttpService {
5
6    constructor(private http: HttpClient) {}
7
8    let backendUrl = 'http://myApi.com/myResource'
9
10   let httpOptions = {
11     headers: new HttpHeaders({
12       'Content-Type':  'application/json',
13       'Authorization': 'my-auth-token'
14     })
15   };
16
17   postDataToService (data: MyData) {
18     return this.http.post(backendUrl, data, httpOptions);
19   }
20
21   deleteDataFromService (id: number) {
22     let url = backendUrl + '/' + id;    // DELETE 'http://myApi.com/myResource/32'
23     return this.http.delete(url, data, httpOptions);
24   }
25 }
```

Notad de la figura anterior que además de pasar los datos (data) también se pasa un objeto con las cabeceras, lo cual nos permite en este caso detallar el formato de los datos enviados, así como cierta información relativa a la autorización.

Otra cosa a tener en cuenta es que es muy importante que el componente se suscriba al Observable, aunque no se espere ni se necesite ningún valor de retorno de la operación solicitada.

http.service.ts

```
21   deleteDataFromService (id: number) {
22     let url = backendUrl + '/' + id;    // DELETE 'http://myApi.com/myResource/32'
23     return this.http.delete(url, data, httpOptions);
24   }
25 }
```

my.component.ts

```
21 http.deleteDataFromService('32').subscribe();
```

Si solamente se llama a deleteDataFromService() - sin llamar a suscribe() - no sucederá nada!

Conclusión

Hemos visto muchos conceptos y todos ellos de gran importancia para el desarrollo en Angular. En la siguiente parte de esta práctica comenzamos a aplicar lo que aquí hemos visto.