Clase 9 @October 20, 2022

Esta clase veremos

- Corrección del practico 3
- Custom pipes
- Servicios e inyección de dependencia
  - Que son los servicios
  - Como funciona la inyección de dependencia
  - Construcción de un servicio
  - Registro del servicio al modelo
  - Utilización del servicio en los componentes

Corrección practico 3

Comandos importantes para recordar

Instalar angular CLI

```
npm install -g @angular/cli
```

Crear un proyecto

```
ng new nombreProyecto
```

Ejecutar el proyecto

```
ng serve --open
```

Instalar bootstrap

```
npm install bootstrap@3 --save
```

Crear un componente

```
ng generate component nombreComponente
```

Crear un modulo

```
ng generate module nombreModulo
```

Custom Pipes: Filtrado en el listado de tareas

Como vimos la clase anterior, Angular provee un conjunto de Pipes que ya vienen integrados y que sirven para transformar los datos de nuestras bound properties antes de mostrarlos en el template (HTML). Ahora veremos como construir nuestros propios, Pipes personalizados, o *Custom Pipes*.

El código necesario para crearlos seguramente a esta altura ya nos resulte familiar:

```
import { Pipe, PipeTransform } from '@angular/core'; //0) importamos
import { Homework } from '../models/Homework';

//1) Nos creamos nuestra propia clase HomeworksFilterPipe y la decoramos con @Pipe
@Pipe({
  name: 'homeworksFilter'
})
export class HomeworksFilterPipe implements PipeTransform { //2) Implementamos la inte
rfaz PipeTransform

  transform(list: Array<Homework>, arg: string): Array<Homework> { //3) Método de la i
```

```
nterfaz a implementar
        //4) Escribimos el código para filtrar las tareas
        // El primer parametro 'list', es el valor que estamos transformando con el pi
pe (la lista de tareas)
        // El segundo parametro 'arg', es el criterio a utilizar para transfmar el val
or (para filtrar las tareas)
        // Es decir, lo que ingresó el usuario
        // El retorno es la lista de tareas filtrada
  }
}
```

Como podemos ver, tenemos que crear una **clase**, y hacerla que implemente la interfaz **PipeTransform**. Dicha interfaz tiene un método **transform** que es el que será encargado de filtrar las tareas. A su vez decoramos la clase con un `@Pipe` que hace que nuestra clase sea un Pipe. Como notamos, la experiencia a la hora de programar en Angular es bastante consistente, esto es muy similar a cuando creamos componentes. Tambien podemos lanzar el comando **ng generate pipe homeworks-filter** que nos da como resultado esto mismo.

Luego, para usar este CustomPipe en un template, debemos hacer algo así:

```
<tr *ngFor='let homework of homeworks | homeworksFilter:listFilter'> </tr>
```

Siendo:

- HomeworksFilter: el pipe que acabamos de crear.
- listFilter: el string por el cual estaremos filtrando.

Si quisieramos pasar más argumentos además del listFilter, los ponemos separados por `:`.

También nos falta agregar el Pipe a nuestro módulo. Si queremos que el componente pueda usarlo, entonces debemos decirle a nuestro AppModule que registre a dicho Pipe. Siempre que queremos que un Componente use un Pipe, entonces el módulo del componente debe referenciar al Pipe. Lo haremos definiendo al Pipe en el array `declarations` del decorador `ngModule` de nuestro módulo.

Armemos el Pipe!

1) Creamos un archivo para el Pipe

Creamos en la carpeta `app/homeworks-list`, un ``homeworks-filter.pipe.ts`, siguiendo nuestras convenciones de nombre. O lanzamos **ng generate pipe HomeworksFilter**``` y

movemos los archivos a la carpeta.

2) Agregamos la lógica del Pipe:

```
import { Pipe, PipeTransform } from '@angular/core';
import { Homework } from '../models/Homework';

@Pipe({
  name: 'homeworksFilter'
})
export class HomeworksFilterPipe implements PipeTransform {

  transform(list: Array<Homework>, arg: string): Array<Homework> {
    return list.filter(
      x => x.description.toLocaleLowerCase()
        .includes(arg.toLocaleLowerCase())
    );
  }
}
```

3) Agregamos el filtrado en el template y sus estilos

Vamos a `homeworks-list.component.html` y donde usamos `*ngFor`, agregamos el filtrado tal cual lo vimos arriba:

```
<tr *ngFor="let aHomework of homeworks | homeworksFilter : listFilter">
```

4) Agregamos el Pipe a nuestro AppModule

Vamos a `app.module.ts` y agregamos el pipe:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { HomeworksListComponent } from './homeworks-list/homeworks-list.component';
import { HomeworksFilterPipe } from './homeworks-list/homeworks-filter.pipe';
import { HomeworksService } from './services/homeworks.service';

@NgModule({
  declarations: [
    AppComponent,
    HomeworksListComponent,
    HomeworksFilterPipe
  ],
  imports: [
    FormsModule,
```

```
    BrowserModule
  ],
  providers: [
    HomeworksService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora ya podremos filtrar en nuestra lista, corremos la aplicación y probamos

Servicios e Inyección de Dependencias

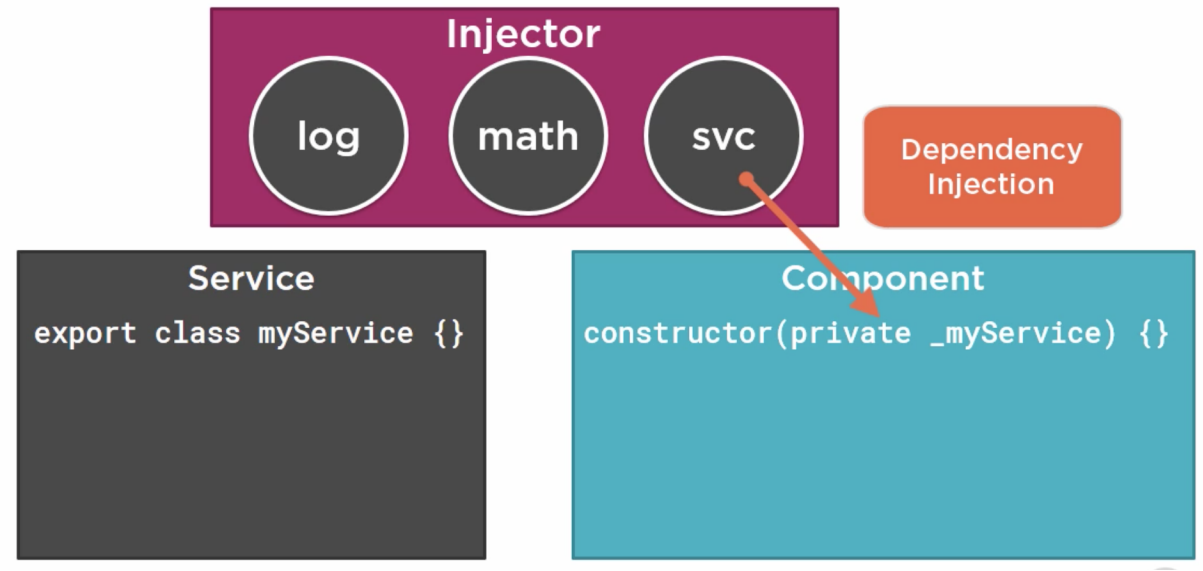
Los componentes nos permiten definir lógica y HTML para una cierta pantalla/vista en particular. Sin embargo, ¿Qué hacemos con aquella lógica que no está asociada a una vista en concreto?, o ¿Qué hacemos si queremos reusar lógica común a varios componentes (por ejemplo la lógica de conexión contra una API, lógica de manejo de la sesión/autenticación)?

Para lograr eso, construiremos **servicios**. Y a su vez, usaremos **inyección de dependencias** para poder meter/inyectar esos servicios en dichos componentes.

Definiendo servicios, son simplemente clases con un fin en particular. Los usamos para aquellas features que son independientes de un componente en concreto, para reusar lógica o datos a través de componentes o para encapsular interacciones externas. Al cambiar esta responsabilidades y llevarlas a los servicios, nuestro código es más fácil de testear, debuggear y mantener.

Angular trae un `Injector` *built-in*, que nos permitirá registrar nuestros servicios en nuestros componentes, y que estos sean Singleton. Este Injector funciona en base a un contenedor de inyección de dependencias, donde una vez estos se registran, se mantiene una única instancia de cada uno.

Supongamos tenemos 3 servicios: svc, log y math. Una vez un componente utilice uno de dichos servicios en su constructor, el Angular Injector le provee la instancia del mismo al componente.



Construyamos un servicio

Para armar nuestro servicio precisamos:

- Crear la clase del servicio.
- Definir la metadata con un @, es decir un decorador.
- Importar lo que precisamos. ¿Familiar? Son los mismos pasos que hemos seguido para construir nuestros componentes y nuestros custom pipes

1) Creamos nuestro servicio

Vamos a `app/services` y creamos un nuevo archivo: `homeworks.service.ts`. O lanzamos `ng generate service Homeworks` y lo movemos a la carpeta

Luego, le pegamos el siguiente código:

```
import { Injectable } from '@angular/core';
import { Homework } from '../models/Homework';
import { Exercise } from '../models/Exercise';

@Injectable()
export class HomeworksService {

  constructor() { }

  getHomeworks():Array<Homework> {
    return [
      new Homework('1', 'Una tarea', 0, new Date(), [
        new Exercise('1', 'Un problema', 1),
        new Exercise('2', 'otro problema', 10)
      ]),
      new Homework('2', 'Otra tarea', 0, new Date(), [])
    ];
```

```
  }
}
```

2) Registramos nuestro servicio a través de un provider

Para registrar nuestro servicio en nuestro componente, debemos registrar un Provider. Un provider es simplemente código que puede crear o retornar un servicio, **típicamente es la clase del servicio mismo**. Esto lo lograremos a través de definirlo en el componente, o como metadata en el Angular Module (AppModule).

- Si lo registramos en un componente, podemos inyectar el servicio en el componente y en todos sus hijos.
- Si lo registramos en el módulo de la aplicación, lo podemos inyectar en toda la aplicación.

En este caso, lo registraremos en el Root Component (`AppModule`). Por ello, vamos a `app.module.ts` y reemplazamos todo el código para dejarlo así:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { HomeworksListComponent } from './homeworks-list/homeworks-list.component';
import { HomeworksFilterPipe } from './homeworks-list/homeworks-filter.pipe';
import { HomeworksService } from './services/homeworks.service'; //importamos el servi
cio

@NgModule({
  declarations: [
    AppComponent,
    HomeworksListComponent,
    HomeworksFilterPipe
  ],
  imports: [
    FormsModule,
    BrowserModule
  ],
  providers: [
    HomeworksService //registramos el servicio para que este disponible en toda nuestr
a app
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3) Inyectamos el servicio en nuestro HomeworksListComponent

La inyección la logramos a través del constructor de la clase, para ello hacemos en `homeworks-list.component.ts`:

Primero el import:

```
import { HomeworksService } from '../services/homeworks.service';
```

Y luego definimos el constructor que inyecta el servicio a la clase:

```
constructor(private _serviceHomeworks:HomeworksService) {
   // esta forma de escribir el parametro en el constructor lo que hace es:
   // 1) declara un parametro de tipo HomeworksService en el constructor
   // 2) declara un atributo de clase privado llamado _serviceHomeworks
   // 3) asigna el valor del parámetro al atributo de la clase
}
```

Eso el HomeworksService y lo deja disponible para la clase. Ahí mismo podríamos inicializar nuestras homeworks, llamando al `getHomeworks` del servicio, sin embargo, no es proljo mezclar la lógica de construcción del componente (todo lo que es renderización de la vista), con lo que es la lógica de obtención de datos. Para resolver esto usarmoes Hooks particularmente, el `OnInit` que se ejecuta luego de inicializar el componente.

```
ngOnInit(): void {
    this.homeworks = this._serviceHomeworks.getHomeworks();
}
```

Quedando, el código del componente algo así:

```
import { Component, OnInit } from '@angular/core';
import { Homework } from '../models/Homework';
import { Exercise } from '../models/Exercise';
import { HomeworksService } from '../services/homeworks.service';

@Component({
  selector: 'app-homeworks-list',
  templateUrl: './homeworks-list.component.html',
  styleUrls: ['./homeworks-list.component.css']
})
export class HomeworksListComponent implements OnInit {
  pageTitle:string = 'HomeworksList';
  homeworks:Array<Homework>;
  showExercises:boolean = false;
  listFilter:string = "";
```

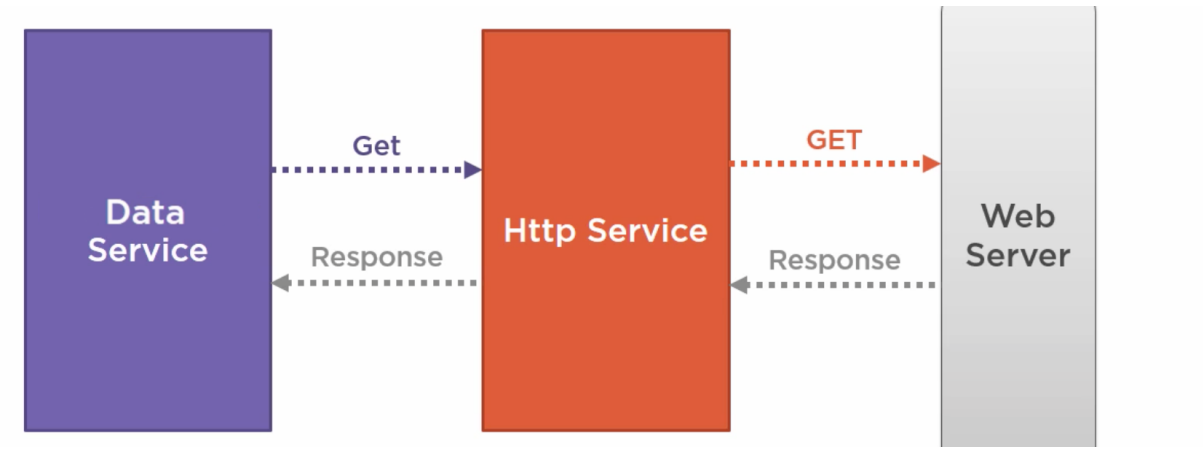
```
  constructor(private _serviceHomeworks:HomeworksService) {

  }

  ngOnInit() {
    this.homeworks = this._serviceHomeworks.getHomeworks();
  }

  toogleExercises() {
    this.showExercises = !this.showExercises;
  }
}
```

HTTP. Observables

Los datos a usar en nuestra aplicación van a estar almacenados en algún lado; en la nube, en un servidor en nuestra misma red, en nuestra pc de escritorio, etc. ¿Cómo hacemos para lograr traer esos datos y ponerlos dentro de nuestras Views?

La mayoría de las aplicaciones hechas en Angular obtienen datos usando HTTP. ¿Cuál es el flujo que se da?

1. La aplicación envía una request a un servidor/web service (HTTP GET ../api/homeworks/2)
2. Ese Web Service obtiene los datos, seguramente accediendo a una base de datos.
3. El Web Service le contesta a la aplicación, con los datos obtenidos, en forma de una HTTP Response.
4. La aplicación procesa entonces los datos (por ej: los muestra en una View).



Observables

Los Observables nos permiten manejar datos asincrónicos, como los datos que vendrán de nuestro *backend* o de algún *web service*.

Los mismos tratan a los eventos como una colección; podemos pensar a un Observable como un array de elementos que van llegando asincrónicamente a medida que pasa el tiempo.

Los **Observables** se usan incluso dentro de Angular, en su sistema de eventos o en su servicio HTTP (motivo por el cual lo estamos viendo).

A su vez, los **Observables** tienen métodos. La gracia de estos métodos es que, una vez se aplican sobre un **Observable**, estos realizan alguna transformación sobre el **Observable** original, y lo retornan. Estos métodos siempre devuelven un **Observable**, permitiendo concatenar llamadas sobre un **Observable** de forma simple. Ejemplo: *map*, *filter*, *take*, *merge*, etc.

Una vez tengamos nuestros Observables, un método en nuestro código puede **suscribirse a un Observable**, para **recibir notificaciones asincrónicas a medida que nuevos datos vayan llegand**o. Dicho método puede entonces "reaccionar", sobre esos datos. Dicho método a su vez es notificado cuando no hay más datos, o cuando un error ocurre.

Observables vs Promises

Otra forma bastante común de obtener datos a través de HTTP es usando **promises**. Las promises/promesas son objetos de JavaScript que sirven para hacer computación asincrónica, representando un cierto valor que puede estar ahora, en el futuro o nunca. Estas permiten setear manejadores (funciones o callbacks), que ejecuten comportamiento una vez que el valor esté disponible. Las llamadas por HTTP, pueden ser manejadas a través de promesas. Esto permite que métodos asíncronicos devuelvan valores como si fueran sincrónicos: en vez de inmediatamente retornar el valor final, el método asincrónico devuelve una promesa de suministrar el valor en algún momento en el futuro.

Tanto Observables como Promises sirven para lo mismo, pero los Observables permiten hacer más cosas:

- Los Observables permiten **cancelar la suscripción**, mientras que las Promises no. Si el resultado de una request HTTP a un servidor o alguna otra operación costosa que es asincrónica no es más necesaria, el objeto **Suscription** sobre un Observable puede ser cancelado.

- Las Promises manejan un único evento, cuando una operación asincrónica completa o falla. Los Observables son como los Stream (en muchos lenguajes), y permiten pasar cero o más eventos donde el callback será llamado para cada evento.
- En general, se suelen usar Observables porque permiten hacer lo mismo que las Promises y más.
- Los Observables proveen operadores como *map*, *forEach*, *reduce*, similares a un array.

Ejemplo Consumiendo nuestra API

Como ya hemos vistos, los servicios de Angular son una excelente forma de encapsular lógica como la obtención de datos de un web service / backend, para que cualquier otro componente o service que lo precise lo use, a través de inyección de dependencias. En la clase anterior hicimos eso, pero manteniendo una lista hardcodeada de tareas. En su lugar, queremos enviar una solicitud HTTP para obtener las tareas.

Así tenemos ahora las tareas:

```
getHomeworks():Array<Homework> {
  return [
    new Homework('1', 'Una tarea', 0,
    new Date(), 2, [
      new Exercise('1', 'Un problema', 1),
      new Exercise('2', 'otro problema', 10)
    ]),
    new Homework('2', 'Otra tarea', 0,
      new Date(), 4, [])
  ];
}
```

Angular provee un Servicio HTTP que nos permite llevar a cabo esto; donde luego de comunicarnos con el backend, cada vez que este nos responda, la respuesta llegará a nuestro servicio (HomeworksService), en forma de Observable.

1. Registramos el HttpModule en el AppModule

- En `app.module.ts`, importamos el módulo que precisamos para hacer solicitudes Http.

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

A su vez necesitamos registrar el provider de ese service, en el Angular Injector. Como en muchos casos, esto ya viene hecho, gracias a que particularmente el módulo **HttpModule** lleva eso a cabo. Por ende, debemos agregarlo al array de imports de nuestro `AppModule`.

```
@NgModule({
  imports:
  [
    BrowserModule,
    FormsModule,
    HttpClientModule, …
```

Recordemos que el array declarations es para declarar componentes, directivas y pipes que pertenecen a nuestro módulo. Mientras que el array imports es para obtener módulos de afuera.

2. Armemos el cuerpo de nuestra llamada

Hagamos que `getHomeworks` devuelva una respuesta de tipo `Observable<Response>`. Siendo Response una clase que contiene información de la respuesta HTTP.

Para ello, en `homeworks.service.ts`, importamos la librería que nos permite trabajar con Observables (Reactive Extensions).

```
import { Observable, throwError } from "rxjs";
```

Es importante notar que las llamadas HTTP son operaciones asincrónicas únicas, por lo que la secuencia Observable contiene sólo un elemento del tipo Response. También precisamos hacer:

```
import { Http, Response } from ‘@angular/http’;
```

Y ahora veamos esto, ¿realmente queremos "observar" Response enteras HTTP? A nosotros simplemente nos interesa obtener tareas, no Responses. No queremos "observar" objetos del tipo Response.

Para cargar el operador map, tenemos que cargarlo usando import:

```
import { map, tap, catchError } from 'rxjs/operators';
```

Esta es una forma bastante inusual de cargar cosas: le dice al Module Loader que cargue una librería, sin particularmente importar nada. Cuando la librería se carga, su código JS se carga, cargándose para esta librería en particular, la función map para que quede disponible.

Para que esto funcione, en la consola deberíamos haber hecho:

```
npm install rxjs --save
```

Finalmente:

- Definimos una constante que tenga la URL de nuestra WebApi (esto en su obligatorio va a cambiar).
- Inyectamos el servicio `Http` en nuestro `HomeworksService`.
- Cambiamos el tipo de `getHomeworks`, para que devuelva `Observable<Response>`.
- Cambiamos el código de `getHomeworks` para que llame al `get` del `_httpService`.

La clase queda algo así:

```
import { Injectable } from "@angular/core";
import { Http, Response, RequestOptions, Headers } from "@angular/http";
import { Observable, throwError } from "rxjs";
import { map, tap, catchError } from 'rxjs/operators';
import { Homework } from '../models/Homework';

@Injectable()
export class HomeworksService {

  private WEB_API_URL : string = 'http://localhost:4015/api/homeworks';

  constructor(private _httpService: Http) {  }

  getHomeworks():Observable<Array<Homework>> {
    const myHeaders = new Headers();
    myHeaders.append('Accept', 'application/json');
    const requestOptions = new RequestOptions({headers: myHeaders});

    return this._httpService.get(this.WEB_API_URL, requestOptions);
  }
}
```

Sin embargo, como ya mencionamos antes, debemos usar la función `map`. Nuestros componentes, como el `HomeworksListComponent`, esperan recibir una lista de tareas, no de respuestas Http `(Response)`. En consecuencia precisamos “traducir” cada

response en un array de tareas. Eso lo hacemos con el operador `map`. Dicho operador lo que nos va a permitir es tomar la Response Http y convertirla en un array de tareas. El argumento que recibe dicha función es una Arrow Function, como ya hemos visto, que son simplemente lambda expressions, que transforma la respuesta en un JSON.

Quedando algo así:

```
getHomeworks():Observable<Array<Homework>> {
    const myHeaders = new HttpHeaders();
    myHeaders.append('Accept', 'application/json');
    let options = {
      headers: myHeaders
    }

    return this._httpService.get(this.WEB_API_URL,options)
      .pipe(
        map((data) => <Array<Homework>> data),
        tap(data => console.log('Los datos que obtuvimos fueron: ' + JSON.stringify(da
ta)))
      );
  }
```

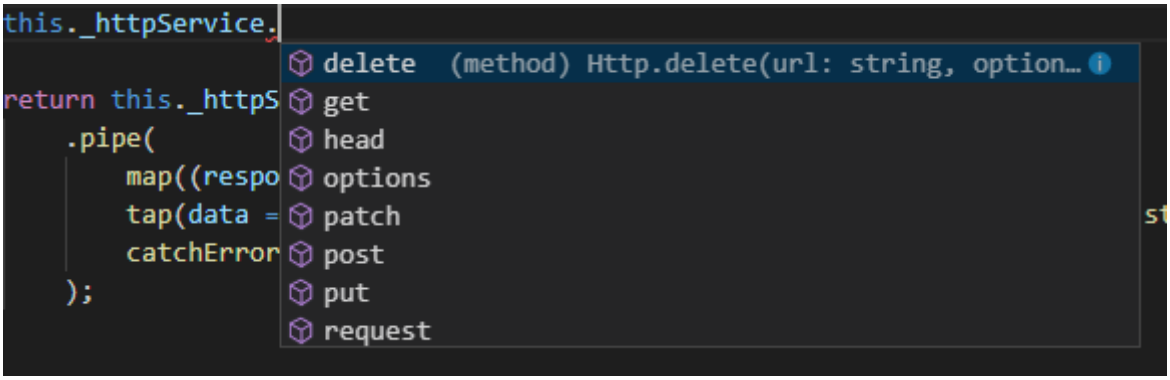
Ahora simplemente agregamos otro operador, para manejar errores. Esto es importante ya que muchísimas cosas pueden darse al querer comunicarse con un servicio de backend, desde una conexión perdida, una request inválida, etc. En consecuencia, agreguemos manejo de excepciones.

Nos queda algo así:

```
import { Injectable } from "@angular/core";
import { HttpClient, HttpResponse, HttpRequest, HttpHeaders } from "@angular/common/ht
tp";
import { Observable, throwError } from "rxjs";
import { map, tap, catchError } from 'rxjs/operators';
import { Homework } from '../models/Homework';

@Injectable()
export class HomeworksService {

  private WEB_API_URL : string = 'http://localhost:4015/api/homeworks';

  constructor(private _httpService: HttpClient) {  }

  getHomeworks():Observable<Array<Homework>> {
    const myHeaders = new HttpHeaders();
    myHeaders.append('Accept', 'application/json');
    let options = {
      headers: myHeaders
    }
```

```
    return this._httpService.get(this.WEB_API_URL,options)
      .pipe(
        map((data) => <Array<Homework>> data),
        tap(data => console.log('Los datos que obtuvimos fueron: ' + JSON.stringify(da
ta))),
        catchError(this.handleError)
      );
  }

  private handleError(error: Response) {
    console.error(error);
    return throwError(error.json()|| 'Server error');
  }
}
```

Es interesante ver como también el servicio `Http` nos permite realizar llamadas usando cualquier verbo Http:



Finalmente, lo que hacemos es que nuestro componente `HomeworksListComponent` se suscriba al resultado de la llamada (a los observables). Esto lo hacemos a través del método `suscribe`. Y cómo los Observables manejan múltiples valores a lo largo del tiempo, la función es llamada para cada valor que el Observable emite. En algunos casos queremos saber cuando el observable se completa, por lo que también podemos tener una función de completado (tercer argumento que es opcional, se ejecuta cuando se completa).

A su vez cuando queramos podemos cancelar la suscripción cuando queramos, con el objeto que nos devolvió al suscribirnos.

La idea es que nuestro componente sea notificado cada vez que un Observable emite un elemento, haciendo alguna acción particular para cada caso (OK, y Error).

Vayamos al `HomeworksListComponent` y en el `OnInit`:

```
ngOnInit() {
  this._serviceHomeworks.getHomeworks().subscribe(
      ((data : Array<Homework>) => this.result(data)),
      ((error : any) => console.log(error))
  )
}

private result(data: Array<Homework>):void {
  this.homeworks = data;
  console.log(this.homeworks);
}
```