Consultas Asíncronas

Material: 2.3.2 Cómo programar consultas asíncronas
Asignatura: PGY4121

Docente Diseñador	*Ian Cárdenas Castillo*	Revisor metodológico	*Manuela Jimenez*

Introducción

En el presente documento ustedes podrán observar el proceso para conectar su aplicación ionic Angular a un APIRest para el consumo de servicios y programar consultas asíncronas. El ejemplo está basado en Ionic 5 y Angular 9 utilizando un APIRest pública para pruebas JSONPlaceholder - Fake online REST API for developers.

Pasos Instalación Servicio API

Importar HttpClientModule

Para comenzar en nuestro proceso debemos agregar HttpClientModule en nuestro archivo app.module.ts que se encuentra a nivel de la carpeta “app” (src -> app) ya que utilizaremos en el servicio api la dependencia HttpClient

```
...

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

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

Generar Service

Luego importar el HttpClientModule en nuestro app.module.ts debemos generar el servicio mediante comando CLI.

```
Ionic generate service nombre_servicio
```

El comando CLI generará un servicio a nivel de la carpeta “app” e importamos las dependencias **HttpClient**, **HttpHeaders**, **HttpErrorResponde** desde la librería @angular/common/http, **retry** y **catchError** desde rxjs/operators y finalmente **Observable** desde rxjs.

Debemos generar una variable que sirva de objeto **Json** para las opciones de la solicitud, en el cual agregaremos una clave llamada “**headers**” y un valor que será un objeto **HttpHeaders** con sus opciones.

Las cabeceras iniciales, son **Content-Type** donde indicaremos que tipo de mensaje vamos a transaccionar en este caso es json y se lo indicamos mediante el siguiente valor **‘application/json’** , la segunda cabecera nos permitirá realizar solicitudes a los servicios con menos dificultades mediante **Access-Control-Allow-Origin** y con su valor **‘*’.**

Finalmente se establecerá un url base que sería la dirección donde consultaremos los servicios.

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

import { HttpClient, HttpHeaders, HttpErrorResponse } from
'@angular/common/http';
import { retry, catchError } from 'rxjs/operators';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class ApiService {
  httpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json',
      'Access-Control-Allow-Origin' :'*'
    })
  }

  // Se establece la base url del API a consumir
  apiURL = 'https://jsonplaceholder.typicode.com';

  // Se declara la variable http de tipo HttpClient
  constructor(private http:HttpClient) { }
```

Consultando al servicio

Ya teniendo el servicio generado podemos empezar a programar las consultas hacia el servicio APIRest mediante sus **end-point**, que son aquellas rutas que nos proveerá de servicios específicos.

Como ya observaron en la presentación “2.3.1 Conectándonos a una APIRest” tenemos a disposición funciónes como GET, POST, PUT y DELETE que nos servirá para darle funcionalidad y vida a nuestra aplicación.

Consultando mediante GET

El función **GET** es uno de los más utilizados en las soluciones Web ya que cuando buscamos por google, accedemos a Instagram, cargamos nuestros correos, vemos nuestro perfil, estamos consultando información a un Servicio mediante GET.

Con esta función podemos **conseguir un conjunto de información como información específica y que considera un objeto**.

GET: Conjunto de información sin filtrar

Llamaremos un c**onjunto de información a una colección de objetos** que nos retorne el servicio consultado, en este caso sin ningún tipo de filtro.

```
getPosts():Observable<any>{
    return this.http.get(this.apiURL+'/posts/').pipe(
      retry(3)
    );
  }
```

En el ejemplo podemos observar que se realiza una petición GET a una ruta ‘posts’, esta nos retorna una lista de post sin ningún tipo de condición.

GET: Conjunto de información con filtro

Llamaremos un conjunto de información a una colección de objetos que nos retorne el servicio consultado, en este caso con filtros.

```
getPost(id):Observable<any>{
    return this.http.get(this.apiURL+'/posts/'+id).pipe(
      retry(3)
    );
  }
```

En caso que queramos filtrar la lista (si es que el end-point lo permite) debemos adjuntar en la url el criterio de filtrado id, si el end-point así lo filtra o un string si es por algún término.

GET: Obtener un Objeto

En ocasiones queremos **obtener sólo un objeto** y para estos motivos también se tiene que usar el función GET

```
getPost(id):Observable<any>{
    return this.http.get(this.apiURL+'/posts/'+id).pipe(
      retry(3)
    );
  }
```

Si esa url retornara solo un objeto, le debemos pasar el id del objeto, esto es muy utilizado cuando contamos con el id del objeto a consultar y el end-point retorna más información al respecto pero se requiere el id.

Creando objetos mediante POST

Para crear un objeto se utiliza el función POST ya que permite enviar un body al servidor transfiriendo por debajo sin mostrar la información en la url.

```
createPost(post):Observable<any>{
    return
this.http.post(this.apiURL+'/posts',post,this.httpOptions)
    .pipe(
      retry(3)
    );
  }
```

En esta función se puede observar que la función del http cambia a un POST, al igual que el GET, le debo indicar la url del servicio que permite crear el objeto. Como segundo parámetro le paso el body que sería el objeto a entregar y como tercer parámetro se entrega un httpOptions donde le indico que el contenido es un JSON (según la configuración mostrada anteriormente)

Actualizando objetos mediante PUT

Para actualizar un objeto se utiliza el función PUT, que permite enviar por url el id del objeto y un body al servidor

```
updatePost(id,post):Observable<any>{
    return
this.http.put(this.apiURL+'/posts/'+id,post,this.httpOptions).
pipe(retry(3));
  }
```

Al igual que los otros ejemplos podemos ver que la función post se reemplaza por el función put y al igual que los anteriores se indica la url donde está alojado el servicio y en la misma url le indicamos el id del objeto a modificar, para posteriormente en un segundo parámetro le pasamos el body y finalmente como tercer parámetro se le pasa el httpOptions.

Eliminando Objetos mediante DELETE

Para eliminar un objeto se utiliza el función DELETE en el cual permite enviar por url el id del objeto a eliminar.

```
deletePost(id):Observable<any>{
    return
this.http.delete(this.apiURL+'/posts/'+id,this.httpOptions);
  }
```

Se realiza de la misma forma que un GET, pero cambiamos la función a un DELETE donde al igual que las otras funciones pasamos por parámetro la url del servicio que nos permite eliminar el objeto seguido del id a eliminar y como segundo parámetro se envía el httpOptions

Utilizando las consultas

Ya teniendo programados nuestras consultas, tenemos que utilizarlas, para esto desde el componente solicitante tenemos que llamarlas y subscribirnos a su respuesta.

Estas llamadas se realizan de manera asíncronas, esto provoca que cuando se solicite la información (instancia de la función) la aplicación no quedará en espera (ejecución del código) sino que seguirá con la siguiente línea y cuando el llamado tenga una respuesta se actualizará la información mediante la suscripción a la función que es un Observable.

Preparar el Page para el uso del servicio

Antes de llamar a las funciones que interactúan con el servicio debemos acondicionar nuestro Page para su uso, esto se realiza de la siguiente manera.

```
import { Component } from '@angular/core';
import { ApiService } from '../api.service';

@Component({
  ...
})
export class HomePage {
  constructor(private api: ApiService) {
  }
}
```

Se importa el Servicio creado **import { ApiService } from '../api.service'** y se declara en el constructor **private api: ApiService.** Ya teniendo esta preparación podremos hacer uso de las funciones ahí contenidas

Consumo de las funciones

Para consumir las funciones se debe llamar a la variable declarada en el constructor de nuestro Page e instanciar la función a utilizar

```
this.api.getPosts().subscribe((res)=>{
      console.log(res[0]);
    },(error)=>{
      console.log(error);
    });
```

En este ejemplo se observa que desde la variable declarada en el constructor del Page instanciamos el función getPosts() y nos suscribimos a su cambio de estado (que es cuando tenemos respuesta ya sea satisfactoria o no, o no se obtiene respuesta del servidor) mediante la función subscribe.

La función Subscribe recibe obligatoriamente 2 parámetros, que son funciones **function(){}**, en este caso utilizamos lambda que son funciones anónimas **()=>{}.** La primera función que recibe es de Success y la segunda es de Error, cuando obtiene respuesta del servidor entre la familia de respuestas 200 y el error cuando recibe respuesta del servidor de las familias 400 y 500

Si quieren utilizar la función POST:

```
createPost(){
    var post={
      title: 'titulo prueba',
      body: 'algun cuerpo del post',
      userId: 1
    }
    this.api.createPost(post).subscribe((success)=>{
      console.log(success);
    },error=>{
      console.log(error);
    })
```

Se genera o usa según sea el caso un modelo de dato en **Json**, que se pasara por parámetro a la función **createPost()** del api.

Conclusión

Consumir recursos de servicios es relativamente sencillo solo se tiene que saber:

- URL del servicio
- Métodos aceptados
- Información Solicitada
- Casos de Retorno (cuando retorna 200 o 400)

Se recomienda que antes de programar realicen pruebas con Postman para entender y estudiar el servicio a consumir.

Éxito en sus desarrollos y recuerden siempre consultar documentación oficial, ver videos tutoriales o revisar los foros. En su quehacer profesional pregúntenle al encargado del servicio que le suministre la información necesaria para poder hacer un consumo correcto de los servicios.