While building Angular applications, we might need the application to communicate with back-end services to fetch or persist data. This is done using **HttpClientModule.**

When we make calls to an external server, we want users to continue to be able to interact with the page. That is, we don’t want our page to freeze until the HTTP request returns from the external server. So, all HTTP requests are asynchronous.

HttpClient from @angular/common/http is used to communicate with backend services.

Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable APIs, and streamlined error handling.

We need to import **HttpClientModule** from @angular/common/http in the module class to make HTTP service available to the entire module. Import HttpClient service class into a component’s constructor. We can make use of HTTP methods like get, post, put and delete.

The HTTP get, post, put and delete methods will automatically convert the received JSON data from the back-end server, to any desired type.

**Note**: The JSON object array data in the file should have the same key names of object literals as mentioned in the CustomModel class imported from './custom-model'.

JSON is the default response type for  HttpClient.

The following statement is used to fetch data from a server.

1. this.http.get(url)

http.get by default returns an observable.

Let us learn about observables and how to use them for service creation.

**What is an Observable?**

An Observable is a 'collection that arrives over time'.

Observable produces data which a subscriber can subscribe to and then use it. It is like a getting a newsletter where a publishing house produces letters and customers subscribe to it and then read it. In code, we can get data from an observable using subscribe() method.

An observable object can be used to represent asynchronous requests.

The newsletter publisher decides when the customers will get their next newsletter. All the newsletters may not be delivered at same time gap. Customers need to wait till the next newsletter arrives.

The **performance**of an Angular application increases if the requests and responses are sent asynchronously. The *asynchronous* communication is facilitated in Angular with the help of observables.

In Angular, observables are handled using a **third party library**called **RxJs.**

**RxJS**

RxJS is a reactive streams library used to work with asynchronous streams of data.

**Why RxJS Observables?**

Angular team has recommended Observables for asynchronous calls because of the following reasons:

1. Promises emit a single value whereas observables (streams) emit many values
2. Observables can be cancellable where Promises are not cancellable. If any HTTP response is not required, observables allow us to cancel the subscription whereas promises execute either success or failure callback even if we don’t need the result
3. Observables support functional operators such as map, filter, reduce, etc.,

Let us see how to create and use an observable in Angular.

Highlights:

* Importing observables
* Handling asynchronous calls using observables

**Demo Steps:**

1. Modify **app.component.ts**file as shown.

1. import { Component } from '@angular/core';
2. import { Observable } from 'rxjs';
3. @Component({
4. selector: 'app-root',
5. styleUrls: ['./app.component.css'],
6. templateUrl: './app.component.html'
7. })
8. export class AppComponent {
9. data!: Observable<number>;
10. myArray: number[] = [];
11. errors!: boolean;
12. finished!: boolean;
13. fetchData() {
14. this.data = new Observable(observer => {
15. setTimeout(() => { observer.next(11); }, 1000),
16. setTimeout(() => { observer.next(12); }, 2000),
17. setTimeout(() => { observer.complete(); }, 3000)
18. });
19. let sub = this.data.subscribe((value) => this.myArray.push(value),
20. error => this.errors = true,
21. () => this.finished = true);
22. }
23. }

**Line 2:** imports Observable class from rxjs

**Line 11:** data is of type Observable which holds numeric values

**Line 16:** fetchData() is invoked on click of a button

**Line 17:** A new Observable is created and stored in the variable data

**Line 18-20:** next() method of Observable sends the given data through the stream. With a delay of 1, 2 and 3 seconds, a stream of numeric values will be sent. The complete() method completes the Observable stream i.e., closes the stream.

**Line 23:** Observable has another method called subscribe which listens to the data coming through the stream. The subscribe() method has three parameters.

* First parameter is a success callback which will be invoked upon receiving successful data from the stream.
* Second parameter is a error callback which will be invoked when observable returns an error
* Third parameter is a complete callback which will be invoked upon successful streaming of values from Observable i.e., once complete() is invoked.

Here, upon successful response, the data is pushed to the local array called myArray, if any error occurs, a Boolean value true is stored in errors variable and upon complete() will assign a Boolean value true in finished variable.

2. Add the below code in **app.component.html**file

1. <b> Using Observables!</b>
2. <h6 style="margin-bottom: 0">VALUES:</h6>
3. <div \*ngFor="let value of myArray"> {{ value }}</div>
4. <div style="margin-bottom: 0">ERRORS: {{ errors }}</div>
5. <div style="margin-bottom: 0">FINISHED: {{ finished }}</div>
6. <button style="margin-top: 2rem;" (click)="fetchData()">Fetch Data</button>

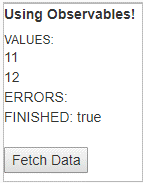
**Line 4:** ngFor loop is iterated on myArray which will display the values on the page

**Line 6:** {{ errors }} will render the value of errors property if any

**Line 8:** Displays finished property value when complete() method of Observable is executed

**Line 10:** Button click event is bound with fetchData() method which is invoked and creates an observable with a stream of numeric values

**Output**:



**Note:**We can convert any data into an observable using the **of()** method. For example,

1. import {Observable, of} from 'rxjs';
3. myObservable:Observable<number> = of(1, 2, 3);
4. myObservable2:Observable<number[]>=of([1,2,3],[4,5,6],[7,8,9]);
5. display(){
6. this.myObservable.subscribe(
7. data=>console.log(data)
8. )
9. this.myObservable2.subscribe(
10. data=>console.log(data)
11. )
12. }

Let us use the same example used for custom services.

Add HttpClientModule to the **app.module.ts** to make use of HttpClient class.

1. ...
2. import { HttpClientModule } from '@angular/common/http';
3. ...
4. @NgModule({
5. imports: [BrowserModule, HttpClientModule],
6. ...
7. })
8. export class AppModule { }

**Line 2:** imports HttpClientModule from @angular/common/http module

**Line 6:**Includes HttpClientModule class to make use of Http calls

# Services and HttpClient

We will be using HttpClient inside our services. Since our services will now have their own dependency, we need to add ***@Injectable()*** annotation on top of the service class.

We can also use Angular cli instead to create a service for us, using the command:

1. ng generate service Book

This will generate a class which looks like below:

1. @Injectable({
2. providedIn: 'root',
3. })
4. export class BookService {
5. constructor() { }
6. }

Here, @Injectable indicates that the service may have its own dependencies. The **providedIn: 'root'** indicates that this should be available to all the components of the module. This is optional if we included it in **providers** array of app.module.ts

# Error handling

What happens if the request fails on the server side, or if a poor network connection prevents it from even reaching the server?

There are two types of errors that can occur. The server might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.

Or something could go wrong on the client-side such as network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript ErrorEvent objects.

HttpClient captures both kinds of errors in its HttpErrorResponse and we can inspect that response to find out what really happened.

We need to do error inspection, interpretation, and resolution in service, not in the component.

Let us learn how to implement this with the help of a demo.