|  |
| --- |
|  |
| Angular 8 |
|  |
| Angular – HttpClient |

**TechBrain Express**

December 19, 2019

Authored by: Ashu

Angular – HttpClient

# Demonstrate using HttpClient to invoke web service

# What HttpClient?

The [HttpClient](https://angular.io/api/common/http/HttpClient) in @angular/common/[http](https://angular.io/api/common/http) offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers.

Benefits of [HttpClient](https://angular.io/api/common/http/HttpClient) **include**

* Testability features,
* Typed request
* Response objects
* Request and Response interception
* Observable apis,
* Streamlined error handling.

# Angular in-memory-web-api

An in-memory web api for Angular demos and tests that emulates CRUD operations over a RESTy API.

It intercepts Angular Http and HttpClient requests that would otherwise go to the remote server and redirects them to an in-memory data store that you control.

Use Cases:

• Demo apps that need to simulate CRUD data persistence operations without a real server. You won't have to build and start a test server.

• Whip up prototypes and proofs of concept.

• Share examples with the community in a web coding environment such as Plunker or CodePen. Create Angular issues and StackOverflow answers supported by live code.

• Simulate operations against data collections that aren't yet implemented on your dev/test server. You can pass requests thru to the dev/test server for collections that are supported.

• Write unit test apps that read and write data. Avoid the hassle of intercepting multiple http calls and manufacturing sequences of responses. The in-memory data store resets for each test so there is no cross-test data pollution.

• End-to-end tests. If you can toggle the app into test mode using the in-memory web api, you won't disturb the real database. This can be especially useful for CI (continuous integration) builds.

LIMITATIONS

The in-memory-web-api exists primarily to support the Angular documentation. It is not supposed to emulate every possible real world web API and is not intended for production use.

Most importantly, it is always experimental. We will make breaking changes and we won't feel bad about it because this is a development tool, not a production product. We do try to tell you about such changes in the CHANGELOG.md and we fix bugs as fast as we can.

# Observable

Angular uses observables extensively in the event system and the HTTP service.

### Observables are lazy collections of multiple values over time.

**Lazy(NewLetter):** You could think of lazy observables as newsletters. For each subscriber a new newsletter is created. They are then only send to those people, and not to anyone else.

**Multiple Values:** Now if you keep that subscription to the newsletter open, you will get a new one every once and a while. The sender decides when you get it but all you have to do is just wait until it comes straight into your inbox.

## Observable Vs Promises

If you come from the world of promises this is a key difference as promises always return only one value. Another thing is that observables are cancelable. If you don’t want your newsletter anymore, you unsubscribe. With promises this is different, you can’t cancel a promise. If the promise is handed to you, the process that will produce that promise’s resolution is already underway, and you generally don’t have access to prevent that promise’s resolution from executing.

## Push Vs Pull

Push and pull are two different ways that describe how a data producer communicates with the data consumer.

|  |  |
| --- | --- |
| PUSH | PULL |
| When pulling, the data consumer decides when it get’s data from the data producer. The producer is unaware of when data will be delivered to the consumer.  Every javascript function uses the pull. The function is a Producer of data, and the code that calls the function is consuming it by “pulling” out a *single* return value from its call. | When pushing, it works the other way around. The data producer (the creator of the newsletter) decides when the consumer (the subscriber to the newsletter) gets the data.  Promises are the most common way of push in JavaScript today. A promise (the producer) delivers a resolved value to registered callbacks (the consumers), but unlike functions, it is the promise which is in charge of determining precisely when that value is “pushed” to the callbacks.  Observables are a new way of pushing data in JavaScript. An observable is a Producer of multiple values, “pushing” them to subscribers. |

Function which returns observable, you can subscribe in two ways.

|  |  |
| --- | --- |
| Manner 1 (Async) | Manner 2 (Subscribe) |
| We subscribe to an observable in our template using the async pipe. | Use the actual subscribe() method. |
| The benefit of this is that Angular deals with your subscription during the lifecycle of a component. Angular will automatically subscribe and unsubscribe for you. Don’t forget to import the “CommonModule” into your module, as the async | The benefits is it can be handy if you would first like to do something with the data before displaying it.   The downside is that you have to manage the subscription yourself. |
| The [AsyncPipe](https://angular.io/api/common/AsyncPipe) subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes. |  |
|  |  |
| EG:  constructor(private httpClient: HttpClient){}  <button (click)="get\_product\_Aysnc()">List Products(async )</button> <table border=2 >  <tr \*ngFor="let product of productsObservable | async">  <td> {{product.id }} </td>  <td>{{ product.name }}</td>  <td>{{ product.cost }}</td>  <td>{{ product.quantity }}</td>  </tr>  </table>  // Demo of Asynch - import { Observable } from 'rxjs';  private productsObservable : Observable<any> ;  get\_product\_Aysnc(){  this.productsObservable = this.httpClient.get(this.baseUrl + '/products');  } | EG:  <button (click)=" get\_product\_Subscribe()">List Products(Get)</button>  <table border=2 >  <tr \*ngFor="let product of products">  <td> {{ product.id }} </td>  <td>{{ product.name }}</td>  <td>{{ product.cost }}</td>  <td>{{ product.quantity }}</td>  </tr>  </table>  constructor(private httpClient: HttpClient){}  private products : Product[] = [];  baseUrl:string = "http://localhost:3000";    get\_product\_Subscribe(){  this.httpClient.get(this.baseUrl + '/products').subscribe((res : any[])=>{  console.log(res);  this.products = res;  });  } |
|  |  |
| There are three functions available to send data to the subscribers of the observable. | |
| next- sends any value such as Numbers, Arrays or objects to it’s subscribers. During observable execution there can be an infinite calls to the observer.next(), | |
| error: sends a Javascript error or exception. It is called, the execution stops and no more data will be delivered to the subscribers. | |
| complete: Does not send any value. It is called, the execution stops and no more data will be delivered to the subscribers. | |
| unscubscribe() : When you subscribe to an observable, you get back a subscription, which represents the ongoing execution | |

Case Study: To learn -

## How to generate a fake and complete working REST API,

## How to create Angular services,

## How to subscribe to Observables,

## How to use the async pipe in templates to iterate over Observable data.

Step 0a: src/app/product.ts

export interface Product {

id: number;

name: string;

cost: number;

quantity: number;

locationId: number;

familyId: number;

}

src/app/family.ts

export interface Family {

id: number;

name: string;

}

src/app/location.ts

export interface Location {

id: number;

name: string;

constructor() { }

}

src/app/transaction.ts

export interface Transaction {

id: number;

cost: number;

productId: number;

quantity: number;

}

## Step 1: Create a new Project

F:\AngularOct19\HttpClientDemo>ng new HttpClientDemo

## Step 2: Setting up HttpClient

src/app/app.module.ts

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

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

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

## Step 3: Setting up a Fake REST API

> $ npm install -g json-server

{

"products": [

{

"id": 1,

"name": "Product001",

"cost": 10.0,

"quantity": 1000,

"locationId" : 1,

"familyId" : 1

},

{

"id": 2,

"name": "Product002",

"cost": 20.0,

"quantity": 2000,

"locationId" : 1,

"familyId" : 2

},

{

"id": 3,

"name": "Product003",

"cost": 30.0,

"quantity": 3000,

"locationId" : 3,

"familyId" : 2

},

{

"id": 4,

"name": "Product004",

"cost": 40.0,

"quantity": 4000,

"locationId" : 2,

"familyId" : 3

}

],

"locations":[

{

"id": 1,

"name": "Location001"

},

{

"id": 2,

"name": "Location002"

},

{

"id": 3,

"name": "Location003"

}

],

"families":[

{

"id": 1,

"name": "FM001"

},

{

"id": 2,

"name": "FM002"

},

{

"id": 3,

"name": "FM003"

}

],

"transactions":[

{

"id": 1,

"cost":11,

"quantity":10,

"productId":1

},

{

"id": 2,

"cost":12,

"quantity":100,

"productId":2

},

{

"id": 3,

"cost":15,

"quantity":101,

"productId":3

}

]

}

$ json-server --watch db.json

## Step 4: Use Subscribe to get Product Data

import { Component } from '@angular/core';

import { Product } from './model/products';

import { Family } from './model/Family';

import { Transaction } from './model/Transaction';

import { Observable } from 'rxjs';

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'HttpClientDemo';

constructor(private httpClient: HttpClient){}

private products : Product[] = [];

private families : Family[] = [];

private locations : Location[] = [];

private transactions : Transaction[] = [];

baseUrl:string = "http://localhost:3000";

get\_product\_Subscribe(){

this.httpClient.get(this.baseUrl + '/products').subscribe((res : any[])=>{

console.log(res);

this.products = res;

});

}

## Step 5: Update app.component.html

<button (click)=" get\_product\_Subscribe()">List Products(Get)</button>

<table border=2 >

<tr \*ngFor="let product of products">

<td> {{ product.id }} </td>

<td>{{ product.name }}</td>

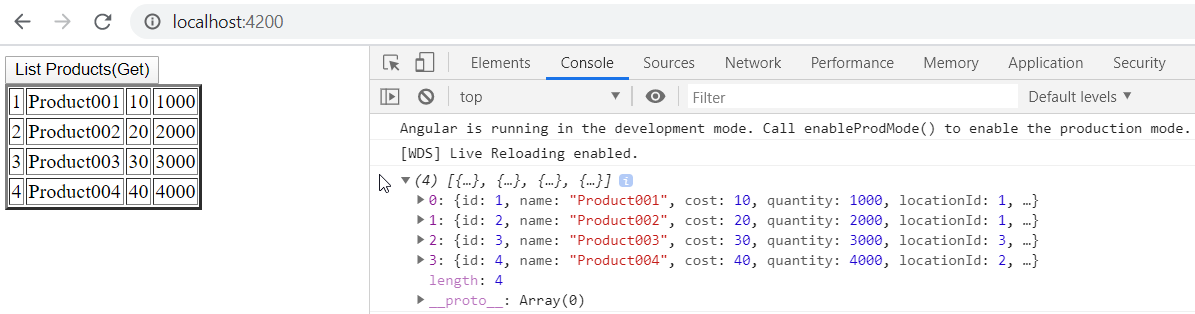
<td>{{ product.cost }}</td>

<td>{{ product.quantity }}</td>

</tr>

</table>

## Step 6: Verify



## Step 7: Use Asyn|Pipe to get data

import { Component } from '@angular/core';

import { Product } from './model/products';

import { Family } from './model/Family';

import { Transaction } from './model/Transaction';

import { Observable } from 'rxjs';

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'HttpClientDemo';

constructor(private httpClient: HttpClient){}

private products : Product[] = [];

private families : Family[] = [];

private locations : Location[] = [];

private transactions : Transaction[] = [];

baseUrl:string = "http://localhost:3000";

// Demo of Asynch - import { Observable } from 'rxjs';

private productsObservable : Observable<any> ;

get\_product\_Aysnc(){

this.productsObservable = this.httpClient.get(this.baseUrl + '/products');

}

## Step 8: Update app.component.html

<button (click)="get\_product\_Aysnc()">List Products(async )</button>

<table border=2 >

<tr \*ngFor="let product of productsObservable | async">

<td> {{product.id }} </td>

<td>{{ product.name }}</td>

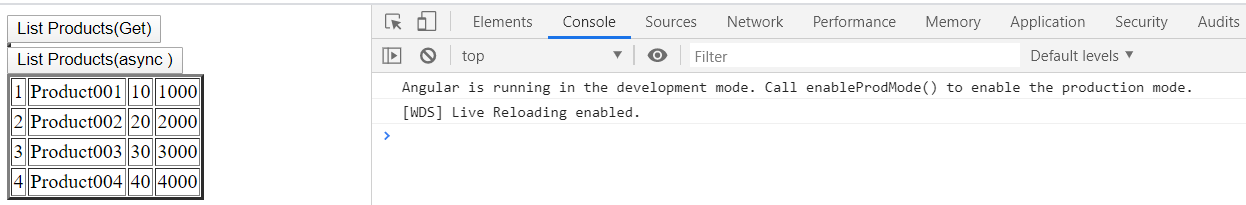
<td>{{ product.cost }}</td>

<td>{{ product.quantity }}</td>

</tr>

</table>

## Step 9: Verify



<https://www.metaltoad.com/blog/angular-2-using-http-service-write-data-api>

## Step 8 : Using Data Service

>Ng g s Data

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

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

@Injectable({

providedIn: 'root'

})

export class DataService {

baseUrl:string = "http://localhost:3000";

constructor(private httpClient : HttpClient) {}

get\_products(){ return this.httpClient.get(this.baseUrl + '/products');}

get\_families(){ return this.httpClient.get(this.baseUrl + '/families');}

get\_locations(){ return this.httpClient.get(this.baseUrl + '/locations');}

get\_transactions(){ return this.httpClient.get(this.baseUrl + '/transactions');}

}

## Step 9: Update app.component.ts

/---------------------------------Demo using Service

private families : Family[] = [];

private locations : Location[] = [];

constructor(private dataService: DataService,private httpClient: HttpClient){

this.dataService.get\_products();

this.dataService.get\_families().subscribe((res : any[])=>{this.families = res; });

this.dataService.get\_locations().subscribe((res : any[])=>{

console.log(res);

this.locations = res;

}

); }//constructor

private transactions : Transaction[] = [];

getTransact()

{ this.dataService.get\_transactions().subscribe(

(res : any[])=>{ console.log(res);

this.transactions = res;

}

); }

## Step 10: Update app.component.html

<button (click)="getTransact()">List Transactions(Calling Data Service </button>

<table border=2 >

<tr \*ngFor="let tr of transactions">

<td> {{tr.id }} </td>

<td>{{ tr.name }}</td>

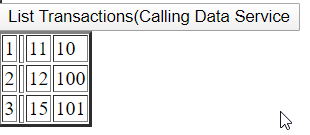
<td>{{ tr.cost }}</td>

<td>{{ tr.quantity }}</td>

</tr>

</table>

## Step 11: Verify



<https://pusher.com/tutorials/error-handling-angular-part-1>