Http Requests

# **App & Backend Setup**

# **Sending Requests (Example: POST Request)**

# **Adjusting Request Headers**

# **Sending GET Requests**

# **Sending a PUT Request**

# **Transform Responses Easily with Observable Operators (map())**

# **Using the Returned Data**

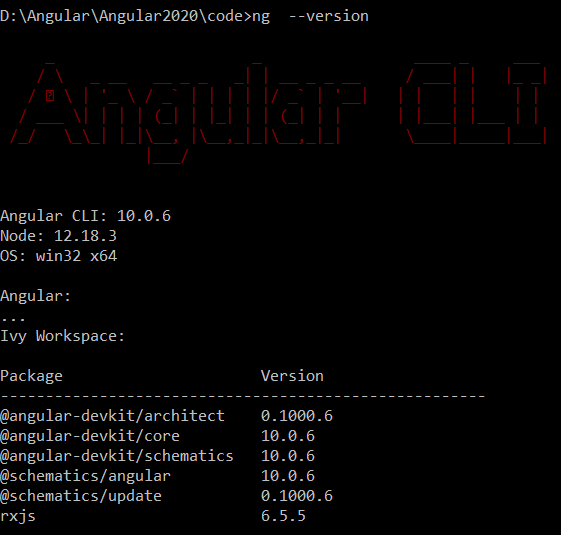
# **Catching Http Errors**

# **Using the "async" Pipe with Http Requests**

# Reference: <https://www.sitepoint.com/update-angular-projects/>

# What is Rest ?

***Representational State Transfer****(****REST****) is a software architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, termed RESTful web services, provide interoperability between computer systems on the Internet. RESTful web services allow the requesting systems to access and manipulate textual representations of web resourcesby using a uniform and predefined set of stateless operations. Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations*

Reference : <https://blog.angular-university.io/angular-http/> 

# 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. |

|  |
| --- |
| 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 |

# Project - CustomerHttpClientDemo

Location : D:\Angular\Angular2020\code

## Create FrontEnd project “CustomerHttpClientDemo”

* ng new CustomerHttpClientDemo

## Setup Fake Json Server

<https://scotch.io/tutorials/json-server-as-a-fake-rest-api-in-frontend-development>

|  |  |  |
| --- | --- | --- |
| 1 | Install json server | npm install -g json-server |
| 2 | Check json-server version | json-server -v |
| 3 | Create db.json in project folder | {  "Customers": [  {  "id": 1,  "name": "Robert",  "gender": "Male",  "age": 50  },  {  "id": 2,  "name": "Ronaldo",  "gender": "Male",  "age": 30  },  {  "id": 4,  "name": "Ginni",  "gender": "Female",  "age": 25  },  {  "id": 3,  "name": "Martin",  "gender": "Male",  "age": 45  }  ]  } |
| 4 | Run Fake Json server | json-server --watch db.json |
| 5 | Test | http://localhost:3000/Customers |

Lab: Create a Project and call and call getCustomer() from service which calls http url. If button clicked call getCustomer()

Summary:

1. Create Project
2. Create Service
3. Call method in required component in from service.
   1. Subscribe
   2. Async Pipe

|  |  |  |
| --- | --- | --- |
| 0. | Create Project | * ng new CustomerHttpClientDemo |
| 1 | Create src/app/customer/customer.ts | export class Customer{  id: number;  name: string;  gender: string;  age: number;  constructor(){ }  } |
| 2. | Create Service | D:\Angular\Angular2020\code\CustomerHttpClientDemo>**ng g s Customer** |
| 3. | Add getCustomer() to call GET method | import { Injectable } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Observable } from 'rxjs';  @Injectable({  providedIn: 'root'  })  export class CustomerService {  constructor(private \_httpService: HttpClient) { }  getCustomers(): Observable<any>{ //asynchronous  return this.\_httpService.get("http://localhost:3000/Customers");  } |
| 3. | Update component.list.component.ts   1. Inject CustomerService 2. Define getCustomer() to call service method 3. Call service method and subscribe it | import { Component, OnInit } from '@angular/core';  import {Customer} from '../customer/customer';  import {CustomerService} from '../customer.service';  @Component({  selector: 'app-customer-list',  templateUrl: './customer-list.component.html',  styleUrls: ['./customer-list.component.css']  })  export class CustomerListComponent implements OnInit {  public customers:Customer[];  isVisible=false;  constructor(private \_customerService:CustomerService) { }  ngOnInit(){}  getCustomers()  {  this.isVisible=true;  console.log("Inside getCustomers():::::");  /\*WAY 1: return this.\_customerService.getCustomers()  .subscribe((x) => this.customers = x,  (error) =>{console.log(error);  this.statusMessage = "Problem with service. Please try again later!";\*/  //Way 2:  this.\_customerService.getCustomers()  .subscribe((res : any[])=>{console.log(res);  this.customers= res; //Will return array of customers  });  }  statusMessage:string;  } |
| 4 | Create Button on customerlist page to call getCustomer() | <p>customer-list works!</p>  <button (click)="getCustomers()">Get Customer</button>  <ng-template [ngIf]="isVisible">  <h2> Customer List </h2>  <table class = "table table-striped table-bordered">  <tr style = "background: rgb(168, 147, 154);">  <th>ID</th>  <th>Name</th>  <th>Gender</th>  <th>Age</th>  </tr>  <tr \*ngFor = "let cust of customers">  <td>{{cust.id}}</td>  <td>{{cust.name}}</td>  <td>{{cust.gender}}</td>  <td>{{cust.age}}</td>    </tr>  <tr \*ngIf = "customers && customers.length == 0">  <td colspan = "3">No records found!</td>  </tr>  </table>  </ng-template> |
|  | Test | Using Way 2: |

# Demonstration of Async Pipe

|  |  |  |
| --- | --- | --- |
|  |  | Refer above project |
|  | Customer.service.ts | import { Injectable } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Observable } from 'rxjs';  @Injectable({  providedIn: 'root'  })  export class CustomerService {  constructor(private \_httpService: HttpClient) { }  getCustomers(): Observable<any>{ //asynchronous  return this.\_httpService.get("http://localhost:3000/Customers");  }  } |
|  | Customer-list.component.ts | import { Component, OnInit } from '@angular/core';  import {Customer} from '../customer/customer';  import {CustomerService} from '../customer.service';  import { Observable } from 'rxjs';  @Component({  selector: 'app-customer-list',  templateUrl: './customer-list.component.html',  styleUrls: ['./customer-list.component.css']  })  export class CustomerListComponent implements OnInit {  constructor(private \_customerService:CustomerService) { }  ngOnInit(){}  //Demo of Async Pipe  public customerObservable : Observable<any> ;  getCustomerUsingAsyncPipe()  { this.customerObservable = this.\_customerService.getCustomers();  }  } |
|  | Customer-list.component.html | <button (click)=" getCustomerUsingAsyncPipe()">Get Customer Using async</button>  <table border=2 >  <tr \*ngFor = "let cust1 of customerObservable | async">  <td>{{cust1.id}}</td>  <td>{{cust1.name}}</td>  <td>{{cust1.gender}}</td>  <td>{{cust1.age}}</td>    </tr>  </table> |
|  |  |  |