

# Angular

## Lab 8

### RxJS and HttpClient

1	LAB SETUP .....	1
2	WORKING WITH JSON .....	2
3	ACCESSING JSON FROM A PUBLIC REST API .....	3
4	GENERATING A NEW ANGULAR PROJECT .....	5
5	CREATING AND CONSUMING OBSERVABLES .....	6
6	USING HTTPCLIENT TO SEND HTTP GET REQUESTS .....	7
7	IMPLEMENTING A LOCAL FAKE API SERVICE USING JSON SERVER .....	8
8	ACCESSING A LOCAL API SERVICE FROM AN ANGULAR APP .....	10
9	USING OTHER REST API METHODS (POST, PUT, DELETE) .....	11
10	USING THE ASYNC PIPE.....	12

## 1 Lab setup

Make sure you have the following items installed

- Latest version of Node and NPM (note: labs are tested with Node v16.13 and NPM v8.1 but should work with later versions with no or minimal changes)
- Latest version of TypeScript (note: labs are tested with TypeScript v4.5.4 but should work with later versions with no or minimal changes)
- Latest version of Angular (note: labs are tested with Angular v13.1 but should work with later versions with no or minimal changes)
- Visual Studio Code (or a suitable alternative IDE for Angular/TypeScript)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`.

- The `changes` subfolder holds the source code and other related files for the lab. The creation of the Angular app will proceed in a step-wise fashion in order to clearly demonstrate the various features of the framework. The various project source files in this folder are numbered in a way to reflect this gradual construction. For e.g. `xxx.v1.html`, `xxx.v2.html`, etc represents successive changes that are to be made to the `xxxx.html`

- The `final` subfolder holds the complete Angular project. If there was some issue in completing the step-wise construction of the app using the files from the `changes` subfolder, you can use this to run the final working version.

The project folder in `final` is however missing the crucial `node_modules` subfolder which contains all the dependencies (JavaScript libraries) that your Angular app needs in order to be built properly. These dependencies are specified in the `package.json` file in the root project folder. They are omitted in the project commit to GitHub because the number and size of the files are prohibitively large.

In order to run the app via the Angular development server (`ng serve`), you will need to create this subfolder containing the required dependencies. There are two ways to accomplish this:

- a) Copy the project root folder to a suitable location on your local drive and run `npm install` in a shell prompt in the root project folder. NPM will then create the `node_modules` subfolder and download the required dependencies as specified in `package.json` to populate it. This process can take quite a while to complete (depending on your broadband connection speed)
- b) Alternatively, you can reuse the dependencies of an existing complete project as most of the projects for our workshop use the same dependencies. In this case, copy the `node_modules` subfolder from any existing project, and you should be able to immediately start the app. If you encounter any errors, this probably means that there are some missing dependencies or Angular version mismatch, in which case you will then have to resort to `npm install`.

The lab instructions here assume you are using the Chrome browser. If you are using a different browser (Edge, Firefox), do a Google search to find the equivalent functionality.

## 2 Working with JSON

The basics of JSON syntax is available at:

[https://www.w3schools.com/js/js\\_json\\_syntax.asp](https://www.w3schools.com/js/js_json_syntax.asp)

Some examples of valid JSON:

<https://www.javatpoint.com/json-example>

<https://json.org/example.html>

Some online sites that perform JSON validation:

<https://jsonlint.com/>

<https://jsonformatter.org/>

You can copy and paste the valid JSON into these sites to check out how they work. Experiment with introducing deliberate errors in the JSON formatting in these validators and see how they flag these errors. The second site (<https://jsonformatter.org/>) also provides options for converting the JSON to its equivalent representation in other related formats such as XML and YAML, as well

### 3 Accessing JSON from a public REST API

There are a few sites on the Web that provide fake REST APIs for purposes of testing and prototyping. A well-known one is:

<https://jsonplaceholder.typicode.com/>

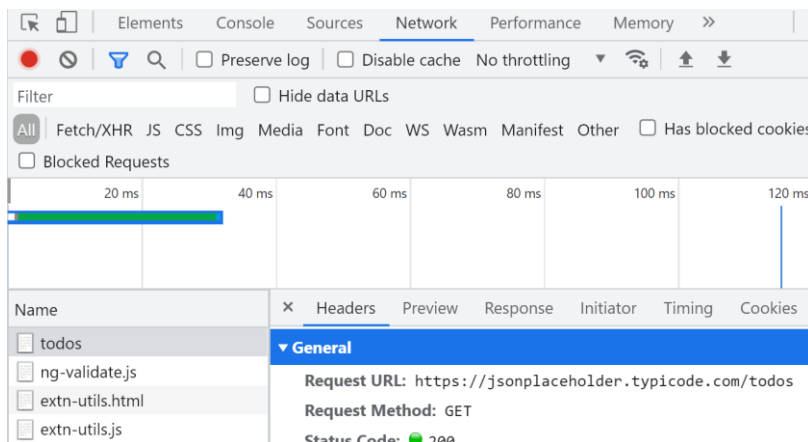
Open a browser tab at this URL and scroll down to the end where you see the subtopics Resources and Routes. Open any of the links highlighted in green in new tab. This sends out a HTTP GET request from the browser to the specified API endpoint whose URL you will be able to see in the address bar. Notice that the complete URL for a given endpoint is the server name (<https://jsonplaceholder.typicode.com>) appended with the path portion (`/posts`) for that endpoint.

You can verify that the returned content is in the form of JSON. Validate the JSON using any one of the online validation sites:

<https://jsonlint.com/>

<https://jsonformatter.org/>

If you wish to inspect the Headers in the request and response messages, and Response in detail, switch to the Network view tab in the Developer Tools and refresh the browser to reload the URL (For Chrome: More Tools -> Developer Tools to access this, a similar tab view exists for other major browser Developer Tool environments).



The Preview view shows the returned JSON formatted in a more structured form.

For each of the API endpoints for the 6 common resources, you can append a number to the end preceded by a forward slash to retrieve a single item from the collection of resources, for e.g.

<https://jsonplaceholder.typicode.com/posts/5>

<https://jsonplaceholder.typicode.com/comments/2>

<https://jsonplaceholder.typicode.com/photos/6>

This number will typically, for the majority of REST APIs, represent a unique ID for the resource in question. Make sure that the number that you append at the end does not exceed the number listed for that particular resource (e.g. 100 posts, 500 comments, etc).

You can also use query parameters (the portion after the ? at the end of the URL path) in your request to drill down to a single item or subset of items from the collection of resources available based on an id for the item, for e.g:

<https://jsonplaceholder.typicode.com/comments?postId=5>

<https://jsonplaceholder.typicode.com/comments?postId=2&id=7>

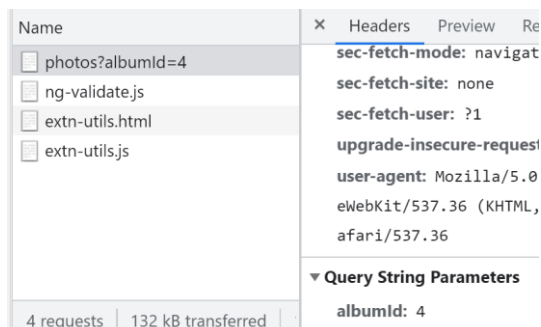
<https://jsonplaceholder.typicode.com/comments?id=10>

<https://jsonplaceholder.typicode.com/posts?id=5>

<https://jsonplaceholder.typicode.com/posts?userId=3&id=23>

Be careful with case as the path components of the URL after the domain name (jsonplaceholder.typicode.com) are case-sensitive: so `postId=5` is not the same as `postId=5`

If you check the Network view in the Developer Tools, you will see the query string parameters shown at the bottom, for e.g.



In order to test out the other REST HTTP methods such as POST, PUT, etc, you will need to write and execute JavaScript code that uses an appropriate library (for e.g. the Fetch API or XHR object) in order to send out these methods. This is because the browser by default can only send out a GET request to a specified URL. Go to this link:

<https://jsonplaceholder.typicode.com/guide/>

Switch to the Console view in the Developer Tools and copy one of the JavaScript snippets and paste it into the Console and press enter. You should be able to see the returned response in the Console view. In the Network view, you will also be able to see the request details (headers and content). Select to filter based on Fetch/XHR to be able to locate the relevant request more quickly.

In a real functional REST API, requests such as PUT, POST, PATCH should modify or add to the content of the resources stored on the server side. Then, subsequent GET requests should be able to retrieve this modified or newly added content. However, for this fake API, no such modification is performed

(due to the heavy traffic load from multiple users testing out this API), so subsequent GET requests that you issue will still show identical content.

Scroll down to the bottom of the page to see the nested routes that are available:

For e.g.

`/posts/1/comments`

would mean to retrieve all the comments related to the post with the id of 1

`/users/1/photos`

would mean to retrieve all the photos related to the user with the id of 1

This nesting relationship between resources would be reflective of the Entity Relationship Diagram in the relational table that is used to store the records.

If you return to the main page:

<https://jsonplaceholder.typicode.com/>

you will see that there is fixed number on the 6 common resources (for e.g. 100 posts, 500 comments and so on). If you attempt to access a resource ID beyond the range for the specified resource, you are effectively attempting to access a non-existent resource. Try this now by typing this URL into a browser tab:

<https://jsonplaceholder.typicode.com/posts/2000>

In the Network view of the Developer Tools, you should an entry with status 404 outlined in red. Responses outlined in red indicate either a client-side error (code 4xx) or server-side error (code 5xx). In this case, there is nothing contained in the response body.

You can repeat this with other resources, e.g.

<https://jsonplaceholder.typicode.com/comments/1000>

In fact, any kind of URL which is not matched by the server-side API code to a resource will result in a 404 error, for e.g.

<https://jsonplaceholder.typicode.com/asdfasdf/2323>

## 4 Generating a new Angular project

In either a separate command prompt (or shell terminal) or in an embedded terminal in Visual Studio Code, create a new Angular project with:

```
ng new rxjshttpdemo
```

Press enter to accept the default values for all the question prompts that follow regarding routing and stylesheet.

Navigate into the root project folder from the terminal with:

```
cd rxjshttdemo
```

Build the app and serve it via Angular's live development server by typing:

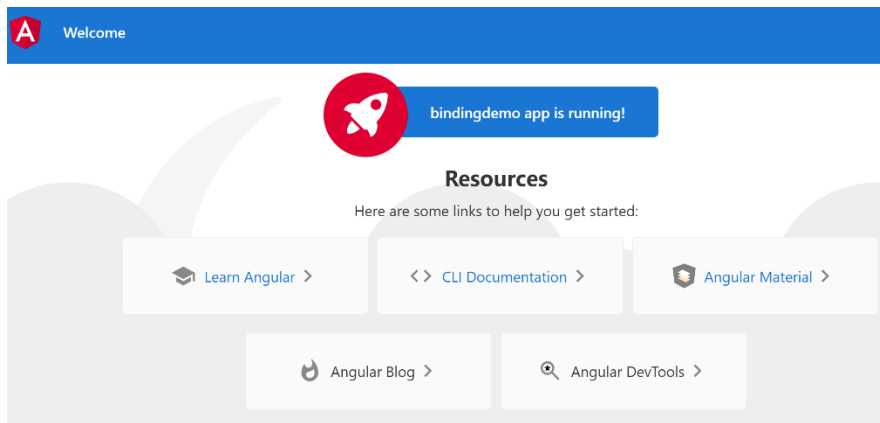
```
ng serve
```

The final output line from this should indicate that the compilation process was successful and identify the port that the live development server is currently serving up the Angular app dynamically from (by default this will be port 4200)

Open a browser tab at:

<http://localhost:4200/>

You should be able to see the default landing page for all new autogenerated Angular projects.



To stop the development server, type Ctrl+C in the terminal window that it is running in. You can restart again anytime by typing `ng serve` in the project root folder.

## 5 Creating and consuming observables

The source code for this lab can be found in the `changes` subfolder of `RxJS-HTTP-Demo`

Modify / add the following files in `src/app` with the latest update from `changes`:

```
app.component-v1.html
```

```
app.component-v1.ts
```

Here, we create two synchronous streams of data to demonstrate how we can subscribe to an observable in order to handle the stream of data returned from it.

Modify / add the following files in `src/app` with the latest update from `changes`:

```
app.component-v1-2.ts
```

Here, we create an asynchronous stream of data to demonstrate how we can subscribe to an observable to get a stream of values that arrive over time and from which we can later unsubscribe from.

References:

<https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/#observable-from-an-array-2>

<https://angular.io/guide/rx-library#the-rxjs-library>

## 6 Using HttpClient to send HTTP GET requests

Here, we use the online public REST API to test out HTTP GET request issues from HttpClient:

<https://jsonplaceholder.typicode.com/>

Modify the following files in `src/app` with the latest update from changes:

`app.component-v2.ts`

`app.component-v2.html`

`app.component-v2.css`

`app.module-v2.ts`

`fakeAPI.service-v2.ts`

`post-v2.ts`

`photo-v2.ts`

General steps involved in using HttpClient for interacting with a backend service via HTTP method calls:

- i. Import `HttpClientModule` into the module containing the components that will use `HttpClient` (typically the root `AppModule`)
- ii. Inject the `HttpClient` as a dependency via the constructor of a service class
- iii. Provide methods in this service class which will call the appropriate methods on `HttpClient` (`get`, `post`, `put`, etc) to issue the corresponding HTTP requests to a service at the specified URL.
- iv. By default, `HttpClient` assumes the HTTP response returned from the service contains JSON within its body and will parse it accordingly as such. The calls to `HttpClient` method (`get`, `post`, `put`, etc) will return an `Observable` which will emit the parsed JSON content from the response when it returns.
- v. We can explicitly type the `Observable` to the structure of this JSON content using either a standard TypeScript class or interface.
- vi. The service class methods can incorporate additional logic to work on the response returned (for e.g. error handling) or return it directly to the component using it.

- vii. This service class is injected as a dependency of a component that will use its methods to perform the various HTTP method calls indirectly
- viii. In the component, we can call the methods of this service class and then subscribe to the Observables returned from them. This will send off the corresponding HTTP requests to the service at the specified URL. Remember that HTTP requests will NOT be sent until the subscribe method is called on an Observable, because the Observable will only start emitting values (values in this case will be the returned HTTP responses) when its subscribe method is called - this is the operation mode for cold observables.

References:

<https://angular.io/guide/>

<https://www.tektutorialshub.com/angular/angular-httpclient>

## 7 Implementing a local fake API service using JSON Server

We can use the JSON Server package to implement a fake API service for purposes of testing HTTP calls from our Angular app. This is usually done while the app is being developed in tandem with a backend service that it will eventually access, but whose custom business logic has not been completed yet. The project website can be found at: <https://github.com/typicode/json-server>.

There may be occasions involving extensive testing where you need to generate large amounts of random data that will be returned from JSON server. For situations like this, we can use Faker.js ( <https://fakerjs.dev/guide/#overview> )

Create a new empty directory (for e.g. `localhostserver`). Open a new command prompt in this directory. Ensure that you still have your existing command prompt with the Angular live development server still running (`ng serve`).

Navigate into this empty directory and install JSON server globally as a NPM package via:

```
npm install -g json-server
```

Next, install Faker.js locally in this directory with:

```
npm install faker@5.5.3
```

NOTE: There is a bug with the latest version of faker.js, so we will use an earlier version instead.

Copy the following files into the empty directory from `changes`:

```
heroes.json
```

```
generateFakeUsers.js
```

In the command prompt in this directory, start up the JSON server to serve the content from this file via a REST API in response to any HTTP requests that it receives:



```
json-server --watch heroes.json
```

In a new browser tab, navigate to:

<http://localhost:3000/>

You should be able to see the landing page for JSON server.

You can retrieve the entire content of the json file that you specified in the watch parameter with:

<http://localhost:3000/heroes>

The following endpoints are automatically created by the JSON server:

```
GET    /heroes
GET    /heroes/{id}
POST   /heroes
PUT    /heroes/{id}
PATCH /heroes/{id}
DELETE /heroes/{id}
```

For e.g., you can retrieve a specific resource based on its id directly by from the browser (which sends a GET request by default) with:

<http://localhost:3000/heroes/3>

You can also use query string parameters to further perform operations on the results returned. For e.g. to match on specific properties within the array of objects:

<http://localhost:3000/heroes?firstName=Bruce>

<http://localhost:3000/heroes?firstName=Bruce&lastName=Banner>

To obtain a sorted order of objects (either ascending or descending) based on the values of a specific property:

[http://localhost:3000/heroes?\\_sort=age](http://localhost:3000/heroes?_sort=age)

[http://localhost:3000/heroes?\\_sort=age&\\_order=desc](http://localhost:3000/heroes?_sort=age&_order=desc)

To get objects whose specific property values are within a specific range:

[http://localhost:3000/heroes?age\\_gte=30&age\\_lte=50](http://localhost:3000/heroes?age_gte=30&age_lte=50)

To do a full text search to locate an object whereby any of its properties have a specified value:

<http://localhost:3000/heroes?q=high>

<http://localhost:3000/heroes?q=Diana>

A list of all the possible options that can be utilized via query string parameters are documented at the project website:

<https://github.com/typicode/json-server#routes>

Stop the running JSON server in the command prompt of the new directory by typing Ctrl-C.

We will now use the Faker library to generate large amount of random JSON content. We already have a program for this purpose, which we can run from the command line with:

```
node generateFakeUsers.js
```

This will generate a new file with sample JSON content: `users.json`

You can restart JSON server to expose the content in this file to access through a REST API:

```
json-server --watch users.json
```

You can now check access to this API in a similar way to the previous example. Some sample endpoints you could try from inside a new browser tab:

<http://localhost:3000/users>

<http://localhost:3000/users?sort=age>

[http://localhost:3000/users?age\\_gte=30&age\\_lte=50](http://localhost:3000/users?age_gte=30&age_lte=50)

References:

<https://medium.com/codingthesmartway-com-blog/create-a-rest-api-with-json-server-36da8680136d>

## 8 Accessing a local API service from an Angular app

Modify the following files in `src/app` with the latest update from changes:

```
hero-v3.ts
```

```
app.component-v3.ts
```

```
app.component-v3.html
```

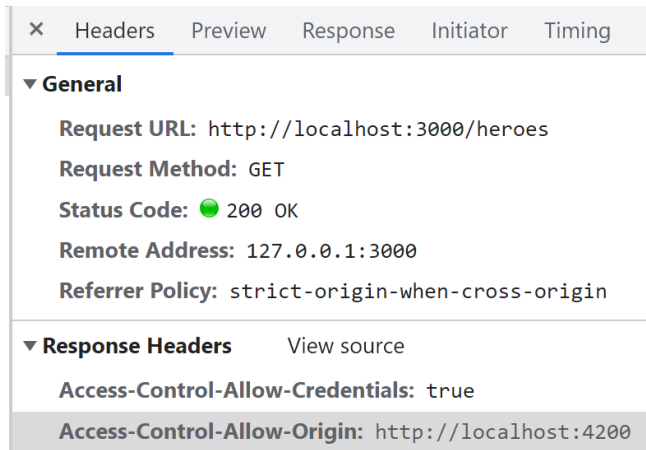
```
localAPI.service-v3.ts
```

Stop the running JSON server in the command prompt of the new directory by typing Ctrl-C.

You can restart the JSON server to expose the content of `heroes.json` through a REST API:

```
json-server --watch heroes.json
```

Note that the JSON server already incorporates the necessary headers for CORS access.



If you are creating your own backend service for testing from scratch using Node.js/Express.js (or any other backend framework for that matter), you must ensure to set the correct value for the `Access-Control-Allow-Origin` header to the port that the Angular app is being served from (in this case, the default value of 4200) or to `*`.

References:

<https://angular.io/guide/http#configuring-http-url-parameters>

<https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/>

<https://www.section.io/engineering-education/how-to-use-cors-in-nodejs-with-express/>

## 9 Using other REST API methods (POST, PUT, DELETE)

Modify the following files in `src/app` with the latest update from changes:

`app.component-v4.ts`

`app.component-v4.html`

`localAPI.service-v4.ts`

`app.module-v4.ts`

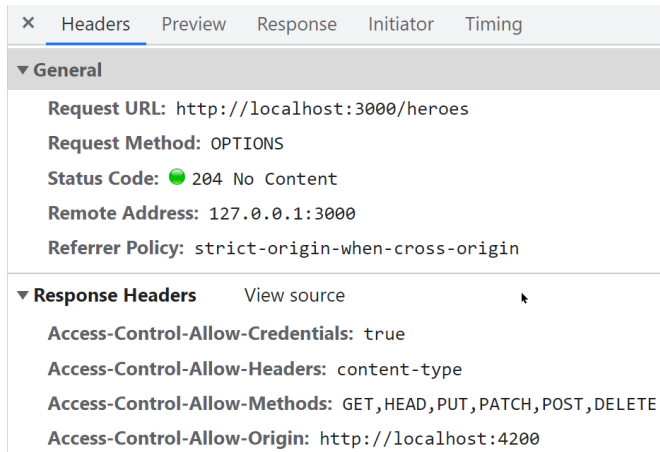
Ensure that the JSON server is running in a command prompt to expose the content of `heroes.json` through a REST API. If it is not, start it up with:

```
json-server --watch heroes.json
```

When working with fake API services like JSON server (and even for real API services), a POST, PUT or PATCH request should include a `Content-Type: application/json` header to use the JSON in the request body.

<https://github.com/typicode/json-server#getting-started>

Notice as well that prior to sending out the POST (or any other request that modifies resources on the backend service), the browser sends out a pre-flight OPTIONS request with related CORS headers, to which the JSON Server service responds with appropriate CORS headers to give the subsequent POST request permission to proceed.



Keep in mind that any backend service that is interacting with the Angular app running in the client-browser must provide support for CORS if the Angular app itself is served from a different origin from than the one that the backend service is running in.

References:

<https://angular.io/guide/http#making-a-post-request>

<https://angular.io/guide/http#making-a-put-request>

<https://angular.io/guide/http#making-a-delete-request>

<https://www.tektutorialshub.com/angular/angular-http-post-example/#http-post-example>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

## 10 Using the Async pipe

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v5.ts`

`app.component-v5.html`

The `async` pipe is used in the template next to an observable property of the component. It is functionally equivalent to an observer that subscribes to that observable and which simply outputs the values emitted from the observable. One of the key reasons for using this pipe is that it automatically unsubscribes from the observable once the component is destroyed, preventing unintended memory leaks

If you use the `async` pipe more than once on an observable that appears multiple times in a template, this is functionally equivalent to subscribing to the Observable more than once. For cold observables (the default state), the observable will emit a completely new stream of values for each new subscription. To avoid this, we can use the `as` keyword to store the `async` pipe and the associated observable property in a template input variable within the specified scope of the template variable

Modify the following files in `src/app` with the latest update from `changes`:

```
app.component-v6.ts
```

```
app.component-v6.html
```

```
localAPI.service-v6.ts
```

Ensure that the JSON server is running in a command prompt to expose the content of `heroes.json` through a REST API. If it is not, start it up with:

```
json-server --watch heroes.json
```

When using the `async` pipe to display contents within a HTTP response directly to a template, there will be limitations in writing code to handle any errors in the response for e.g. a 404 status code. This is because there is no specific callback registered to handle errors occurring in the observable stream such as in the case of the observer object when it is used with the `subscribe` method.

References:

<https://malcoded.com/posts/angular-async-pipe/>

<https://www.tektutorialshub.com/angular/angular-async-pipe/>

[https://www.learnrxjs.io/learn-rxjs/operators/error\\_handling/catch](https://www.learnrxjs.io/learn-rxjs/operators/error_handling/catch)