Angular
Lab 8
RxJS and HttpClient

1 Lab setup

The setup here is identical to that in Lab 1

2 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 stylesheet format (CSS) and enabling Server side rendering (No)

Navigate into the root project folder from the terminal with:

```
cd rxjshttpdemo
```

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.

3 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 demonstrate how the setTimeout and setInterval JavaScript methods work with arrow functions

3.1 Working with asynchronous stream of data

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

`app.component-v1-2.ts`

`app.component-v1-2.html`

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 (using setInterval) and from which we can later unsubscribe from (using setTimeout).

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

3.2 Using observables to pass data in a component hierarchy

Generate the following new components by typing in a command prompt of the root folder of the current project:

```
ng generate component child

ng generate component grandChild

ng generate component greatGrandChild
```

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

```
app.component-v1-3.ts

app.component-v1-3.html

child.component-v1-3.ts

child.component-v1-3.html

grand-child.component-v1-3.ts

grand-child.component-v1-3.html

great-grand-child.component-v1-3.ts

grea-grand-child.component-v1-3.html
```

Enter values for the great grand child string and parent string and verify that they are propagated correctly up and down the hierarchy to their final destination components.

If we wish to pass data from a root component to deeply nested component in the component hierarchy (and back again the other direction), the standard way to do it is through the @Input decorator with property binding and @Output decorator with event binding. Performing it this way however results in a lot of boilerplate code (properties, methods, HTML) in all the intermediary components that convey this data between a root component and the final destination component in the component hierarchy. This technique is known as prop drilling (which is a common feature in the React library).

For e.g. here, you can see that the component and template for Child and GrandChild contain properties, methods and HTML that are not directly used by them, but only serve to convey data between the root (AppComponent) and the final destination component (GreatGrandChild).

Prop drilling is considered a bad practice and tends to become cumbersome and error-prone as the application grows larger. To overcome this, we can instead create a shared service that allows a publisher component to publish data to and one (or more) consumer components that subscribe to changes on this data in the service.

First, we generate a new service by typing in a command prompt in the root folder of the project:

```
ng generate service ParentData
```

This creates a singleton service which can then be subsequently utilized by the root component (the producer) and the great grand child component (the consumer)

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

`parent-data.service-v1-4.ts`

`app.component-v1-4.ts`

`app.component-v1-4.html`

`great-grand-child.component-v1-4.ts`

`great-grand-child.component-v1-4.html`

Verify that you can set a new value for the second string in the parent component, and this is subsequently received successfully by the great grand child component

Here, we have a created a singleton service (`providedIn: 'root'`) which contains a single data property that is made into an Observable subject. This allows it emit a stream of values when a publisher publishes new values to it, which is then subsequently made available to all consumers that subscribe to it via the subscribe method.

Thus when the great grand child component subscribes to this property and the parent component publishes to it, the data is transmitted successfully without the need of the @Input and @Output property and event binding in all the intermediary classes. This effectively eliminates the need for all the boilerplate code that we saw earlier.

Reference:

https://medium.com/@reurairin/passing-data-between-components-in-angular-6230619fe0e3

https://stackoverflow.com/questions/59485252/sending-data-to-a-deeply-nested-angular-component

4 Working with JSON

The basics of JSON syntax is available at:

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

5 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:

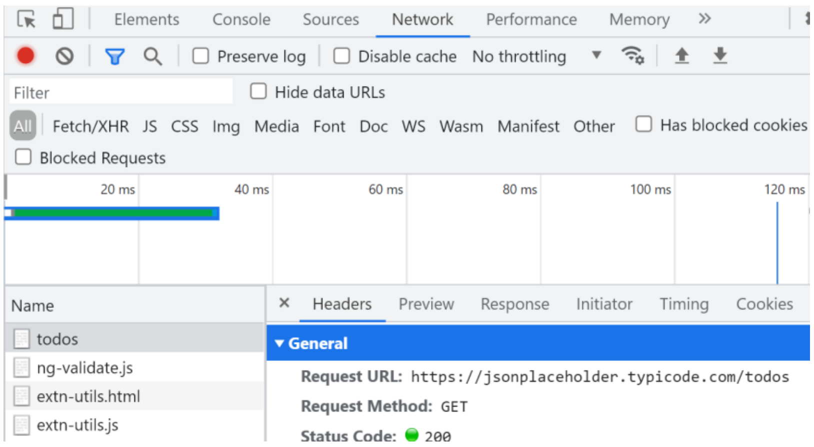
https://jsonplaceholder.typicode.com/

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).



Request URL: https://jsonplaceholder.typicode.com/todos
Request Method: GET
Status Code: 200

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

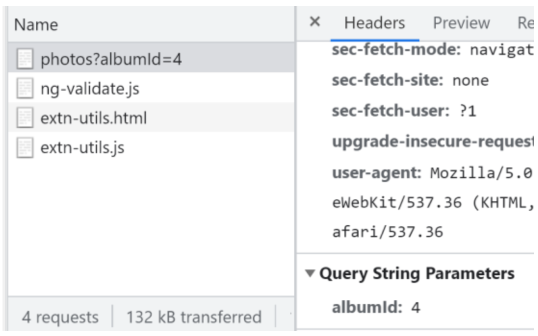
https://jsonplaceholder.typicode.com/comments?id=10

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.



```
sec-fetch-mode: navigat
sec-fetch-site: none
sec-fetch-user: ?1
upgrade-insecure-request
user-agent: Mozilla/5.0
eWebKit/537.36 (KHTML,
afari/537.36
▼ Query String Parameters
albumId: 4
```

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

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.config-v2.ts`

Create the new files `post.ts` and `photo.ts` and `fakeAPI.service.ts` with the latest update from `changes`:

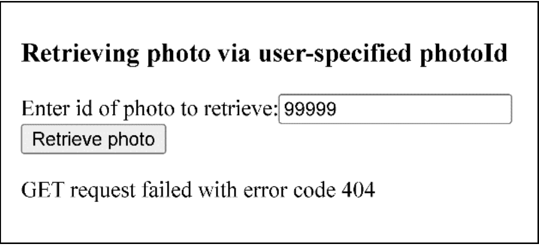
`post-v2.ts`

`photo-v2.ts`

`fakeAPI.service-v2.ts`

Check the Network tab of the Chrome Developer tools to observe the outgoing HTTP GET requests and their URLs, their headers and the content of the responses.

Attempting retrieving a photo with a non-existent ID, for e.g. 999999999 and notice that the arrow function for handling error responses in the subscribe method on getSinglePhoto will now execute:

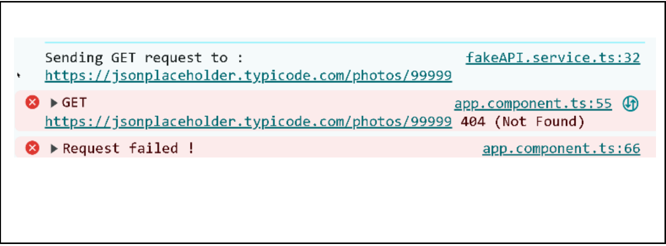


Retrieving photo via user-specified photoId

Enter id of photo to retrieve: 99999

Retrieve photo

GET request failed with error code 404



```
Sending GET request to :                                    fakeAPI.service.ts:32
https://jsonplaceholder.typicode.com/photos/99999
▶ GET                                                       app.component.ts:55
https://jsonplaceholder.typicode.com/photos/99999 404 (Not Found)
▶ Request failed !                                          app.component.ts:66
```

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

i. We need to include the provideHttpClient invocation in config.ts (this is referenced together with AppComponent in main.ts for bootstrapping the application). This allows us to use HttpClient in fakeAPI.service. For Angular 15 and earlier, 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 (FakeAPIService)
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 being 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. `localserver`). 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&_order=desc

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

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

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`:

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

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

Create the following files `hero.ts` and `localAPI.service.ts` in `src/app` with the latest update from `changes`:

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

```
hero-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
```

You can attempt to retrieve the heroes based on the various approaches available.

Check the Network tab of the Chrome Developer tools to observe the outgoing HTTP GET requests and their URLs, their headers and the content of the responses.

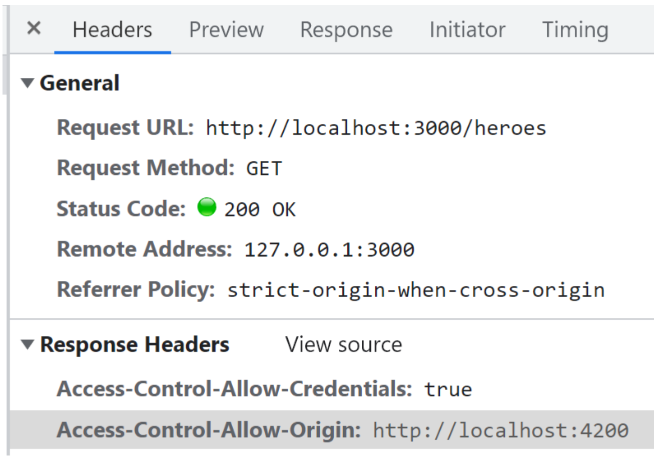
For the implementation of `retrieveAllHeroes` and `getHeroUsingID`, we have included 3 callbacks where the 2nd callback handles various possible error codes in the HTTP response and provides a suitable message to alert the user.

a) To test this out, try retrieving a hero with a non-existent ID while the local JSON server is still running and verify the error message that is displayed.
b) Then stop the local JSON server, and attempt to retrieve a hero with a valid ID and verify the error message that is displayed.

Notice that two different error messages are displayed to the user. There are a few other possible error messages that can be displayed depending on the most common HTTP error codes:
https://www.dotcom-monitor.com/blog/the-10-most-common-http-status-codes/
Of course, the actual number of error messages that you choose to display should be limited since we do not expect a non-technical user to be acquainted with all possible types of errors.

For the 2 other methods (`getHeroesUsingNames` and `getHeroesWithAgeRange`), we have only registered one callback to the subscribe to handle successful emission of values, which means that these methods will not be able to provide suitable messages to alert the user to situations such as the server being down.
Verify this by stopping the local JSON server and attempting to use these 2 approaches to retrieve heroes and notice that no error message is displayed in the template view, although there is an error logged in the console output.

Notice as well that the JSON server already incorporates the necessary headers for CORS access.



▼ **General**

Request URL: `http://localhost:3000/heroes`
Request Method: `GET`
Status Code: ● `200 OK`
Remote Address: `127.0.0.1:3000`
Referrer Policy: `strict-origin-when-cross-origin`

▼ **Response Headers** View source

Access-Control-Allow-Credentials: `true`
Access-Control-Allow-Origin: `http://localhost:4200`

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
```

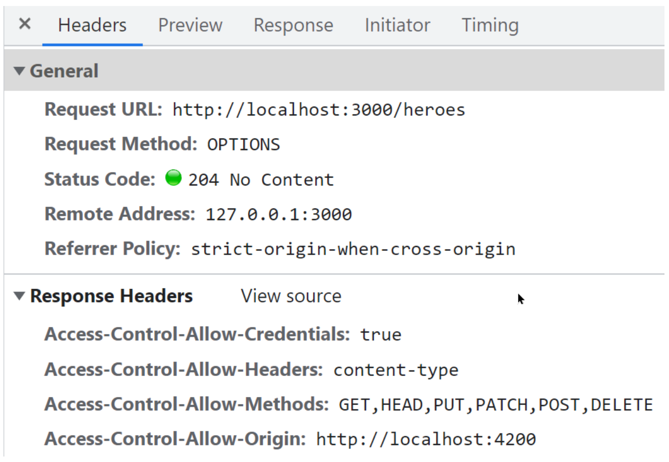
Ensure that the JSON server is running in a command prompt to expose the content of `heroes.json` through a REST API. If it is not, start it up with:

```
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.



General

Request URL: http://localhost:3000/heroes
Request Method: OPTIONS
Status Code: ● 204 No Content
Remote Address: 127.0.0.1:3000
Referrer Policy: strict-origin-when-cross-origin

Response Headers

Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
Access-Control-Allow-Origin: http://localhost:4200

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