Creating Angular 8 web app and setting up the project

Angular Setup

Install Node.js

Before we begin, we need to make sure our development environment includes Node.js® and an npm package manager. Lets go to https://nodejs.org/en/ and install Node.js. You can check if node and npm have been properly installed by opening terminal/console window (Administrator mode) and typing following commands

```
node -v

npm -v
```

Install TypeScript globally

Since Angular uses TypeScript lets install it globally using your command Prompt

```
npm install -g typescript
```

Creating Angular app with Angular CLI and Visual Studio Code Setup

We typically use Angular CLI to create projects, generate all the required application code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment. Lets install Angular CLI globally, to install the CLI using npm, open a terminal/console window (as Admin) and enter the following command

```
npm install -g @angular/cli
```

Now lets create our Angular application.

We are going to use Visual Studio Code for all our Angular Development as its lightweight and has very many great extensions that's going to help developer productivity. But you can use any of your favorite IDE. Open Visual Studio Code and open its built-in terminal, shortcut is **Ctrl+~** Navigate to the src folder that we created in the terminal window and type the following command

```
ng new MovieShopWeb
```

If it asks for Add Routing then select Yes, and for styles select scss option (more on that later). Then it will some time to install all the required files. Once installed type cd MovieShopWeb in the terminal window so

that we are in the exact location. Now lets enter following command to make sure our Angular app is properly installed

```
cd MovieShopWeb
ng serve --open
```

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files. The --open (or just -o) option automatically opens your browser to http://localhost:4200/

In the browser you should see the following greeting message, which means we have successfully created our Angular App.

>  alt text

Understanding the Angular app - Walkthrough

Before we dig deep into Angular files and folder structure lets have some basic understanding of webpack **webpack** - It's a popular module bundler, a tool for bundling application source code in convenient chunks and for loading that code from a server into a browser. When we say bundle in JavaScript world you can think of it as something that has all the HTML, CSS and JavaScript that is required by your application.

> Webpack scans your application source code looking for import statements then it build dependency graph and emitting one or more modules. With various plugins and rules you can have Webpack preprocess and minify non JS files such as TypeScript, SAAS files etc.

Now when you create the Angular app with Angular CLI it creates lots of files. The one file that's interests us is angular.json. Webpack is very powerful and can be configured as much as you want. At the same time learning and configuring webpack is time consuming process and as a developer you don't want to spend most of your time configuring your application. So what happened is Angular team kind of embedded webpack into Angular CLI so that you as Angular developer don't need to worry abut it. You can look for webpack references inside the in node_modules -> @angular-devkit -> in our Angular App. So basically Angular CLI calls webpack (its kind of like a proxy) and then your code bundling happens. So then main.ts will be transpiled into JavaScript.

How Angular builds ? The Angular CLI calls Webpack, when Webpack hits a .ts file it passes it off to TypeScript Compiler which has a output transformer which compiles Angular templates So build sequence is:

Angular CLI => Webpack => TypeScript Compiler => TypeScript Compiler calls the Angular compiler in compile time.

Now lets take a look at angular.json file, here we have main, index which refer to **main.ts** file and **index.html** file. Remember index.html file is the Single html file that we talk about Single Page Applications.

Now lets go to main.ts, here you can see that we have import statement for **AppModule**, every Angular app requires at least one AppModule, If you look at platformBrowserDynamic() , it means we are about to boot Angular in a browser environment, that's because Angular can be used in server environments also.
The bootstrapModule() function helps bootstrap our root module taking in the root module as its argument .

AppModule is our root module which is the entry module for our application, this can actually be any of the modules in our application but by convention AppModule is used as the root module.

In our AppModule, we then need to specify the component that will serve as the entry point component for our application. This happens in our app.module.ts file where we import the entry component (conventionally AppComponent) and supply it as the only item in our bootstrap array inside the NgModule configuration object

Understanding of NgModule

Now if you look at AppModule class it has a decorator called NgModule (you can think of decorator as similar to attributes in C#), so basically AppModule is just a normal TypeScript class that becomes a Angular module with **@NgModule** decorator

There are four parts in our @NgModule that interests us

- **declarations** array : Basically declarations array tells Angular which components belong to that module and as you create more and more components, we need to add them over here. But remember as we use Angular CLI for most of our development we don't need to add them here manually, CLI automatically all newly created Components here. Not only can components declarations reside in this array but also any directives and pipes also go here.

```
declarations: [
YourComponent,
YourPipe,
YourDirective
],
```

- **imports** array : The purpose of imports array is to tell Angular what other NgModules that our application is going to use.Right now we have two modules in our application **BrowserModule** and **AppRoutingModule** . BrowserModule which was by default imported since we are going to use browser to run our app, similarly AppRoutingModule is imported as we are going to use Routing features in our Application. Similarly if you want to use any third-party modules or other Angular modules such as **FormsModule** (When you want to build template driven forms) or **HttpClientModule** (when you want to make HTTP requests to server) you need to import those modules.

- **providers** array : The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading

- **bootstrap** array: This is where we tell Angular which component should be loaded in to browser when application launches. Technically you can put more than one Component here, most of the time we will be having only one component. In our case **AppComponent** is the bootstrapped component.

Note: since Angular 6, services don’t need to be registered in a module anymore. The use of “providers” in a NgModule is now limited to overriding existing services.

Configuring the Routing - Basic Setup

If you remember when we are creating our new Angular application the CLI asked whether we want router or not and we said yes, thats because in our application we usually have different areas where we want to divide them based on certain rules, like protecting certain pages from un-authorized users, maintain the state of the application, define clear and readable URL patterns etc.

In our application we can divide our app in to three areas of access

- First is unrestricted access, i.e anyone can access those pages without authentication, basically publicly available. Like searching for movies, browsing movie details etc.
- Second is access to pages where only Authenticated users can see, like only users that have been authenticated can buy movies.
- Third is restricting access of certain pages to only Admin users, for example in our application only Admin can change movie details, add movies etc.

The Angular router module along with directive helps us in dynamically display the components based on the url. If you look at the **app.component.html** we have which essentially acts like a placeholder where based on the URL that particular component will be displayed.

Our application has **app-routing.module.ts** which was created when we created new app using Angular CLI. Even if you said no for router when you were creating the angular app you can create a separate routing module using Angular CLI later in development. For example

```
ng generate module app-routing --module app --flat
<!-- will create NgModule that can use routing -->
```

Here we have module called **AppRoutingModule** that is exported and used in our main **AppModule**. The router will have no routes until we configure it, we can configure our routes in Routes[] array which in turn used at **RouterModule.forRoot()** method

For example lets take a look at following routes configuration

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'genres', component: GenreComponent },
  { path: 'genre/:id', component: MovieListComponent },
  { path: '**', component: PageNotFoundComponent }
];
```

The empty path in the first route represents the default path for the application, so when we browse to **localhost:4200** then our *HomeComponent* will be rendered.

In the second path we have **'genres'** as path and *GenreComponent* as component, that means **localhost:4200/genres** will render the GenreComponent.

The third one we have **'genre/:id/'** that means **localhost:4200/genre/2** url will map 2 to id , and it will render *MovieListComponent* in which we get the id from URL and call the API ..so on

Finally we have ** which is wildcard router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration, this is where we can use or 404 *PageNotFoundComponent*

Also , the order in which routes are configured is important as the router will use first-match wins strategy, therefore we need to have more specific routes first and then more generic later. The wildcard route comes last because it matches every URL and should be selected only if no other routes are matched first

WE will look into more Routing features as we go along.

Lets create a HomeComponent so that when our application loads HomeComponent will be rendered Also go ahead and create new component called GenresComponent using CLI so that we can show list of genres.

```
  ng g c Home
  ng g c genres
  <!-- here g means shorthand for generate and c means component and then finally
genres is the component name -->
```

By default CLI creates new folder when ever you create a new component, but you can use **--flat** to create new files at the current top level of the project.

```
//  add the following routes to our routes array in AppRoutingModule
      const routes: Routes = [
        { path: '', component: HomeComponent },
        { path: 'genres', component: GenresComponent },

      ];

  //  import the required components path in AppRoutingModule
  import { GenresComponent } from './genres/genres.component';
  import { HomeComponent } from './home/home.component';
```

Lets delete all the HTML markup in *app.component.html* except for the and then use Angular template syntax to render title property in the *app.component.html* {{title}}. The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. This process is called Interpolation {{...}}. With interpolation, you put the property name in the view template, enclosed in double curly braces: {{title}}

Communication between Angular and Rest API with Angular HttpClient

Most of the front-end applications communicate with backend REST API over HTTP protocol ans since we are using Angular here and the http request from Angular (means from browser) will be going to our ASP.NET Core API. Most of the browsers supports two ways of making HTTP requests

- One using **XMLHttpRequest** interface
- Second one is **fetch()** API

So, technically you can use either one of those in your Angular project to make an Http Call. But Angular has **HttpClient** in **@angular/common/http** which offers a much simplified API to make HTTP requests which uses XMLHttpRequest internally.

https://github.com/angular/angular/tree/master/packages/common/http/src

CORS and enabling CORS in ASP.NET Core

When we say Angular is making a Http Request means it uses HttpClient (which uses **XMLHttpRequest**) to call API through browser. But the thing with browsers is that they have security mechanism called ***same-origin policy*** placed that prevent from making request to a different domain than the one where the actual page is on.

lets say our Angular application is on **https://example.com/foo.html** and when it wants to make an HTTP request to either of the following domains it is considered to violate **same-origin policy** therefore browser prevents making the HTTP call

- https://example.net – **Different domain , here we have .net instead of .com**
- https://www.example.com/foo.html **– Different subdomain here we have www**
- http://example.com/foo.html – **Different scheme, this one is using HTTPS**
- https://example.com:9000/foo.html – **Different port**

But there will be scenarios where we want to make HTTP request to other domains, simple example would be when you wanna show some weather information on your website where you need to call some third-party weather API which is obviously in different domain.

This is where **Cross Origin Resource Sharing (CORS)** comes in to picture, basically CORS which is a W3C standard allows server to relax the same-origin policy. So using CORS a server can explicitly allow some cross-origin requests. So when we configure our Server (API) to allow some CORS requests by adding some new special HTTP headers, which will be used by web browsers to determine whether the **XMLHttpRequest** should continue or fail. There are many HTTP headers that can be used when implementing CORS standard but the most important one is **Access-Control-Allow-Origin**.

For example if browser supports CORS then it will set **Origin** header that provides the domain of the site that's making the request Then is server is configured for CORS and allows some domain for making HTTP requests then it sets the **Access-Control-Allow-Origin** header in the response. The value of this header either matches the Origin header from the request or is the wildcard value "*", meaning that any origin is allowed.

If the response doesn't include the **Access-Control-Allow-Origin header**, the cross-origin request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser doesn't make the response available to the client app

In our application we need CORS as our local Angular app is running on http://localhost:4200/ and our API is on https://localhost:44346/ (remember different port and scheme means violation of **same-origin policy**)

Lets test the CORS for our Angular application by making HTTP request using Angular's **HttpClient**.

Before we can start using the HttpClient, we need to import the Angular **HttpClientModule** in the AppModule. this is how our AppModule will look like

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { GenresComponent } from './genres/genres.component';
import { HomeComponent } from './home/home.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent,
    GenresComponent,
    HomeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Lets go to our **GenresComponent** and write following code:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-genres',
  templateUrl: './genres.component.html',
  styleUrls: ['./genres.component.scss']
})
export class GenresComponent implements OnInit {

  genres: any;
  constructor(private http: HttpClient) { }

  ngOnInit() {
    this.http.get('https://localhost:44312/api/genres').subscribe(
      (data) => {
        console.log(data);
        this.genres = data;
      }
    );
  }
```

```
}
```

Now lets examine our code:

Here, first we are importing HttpClient from **@angular/common/http** module then in the constructor we created private variable http of type Http. We are going to use get method to fetch the data from our API. Here we are specifying what is the URL for our HTTP request. In our case its https://localhost:44312/api/genres.

HttpClient in Angular is **Observable** based API which is coming from RxJS library https://rxjs-dev.firebaseapp.com/ (version 6). You can think of Observables as mechanism that provides a way to pass messages between publishers and subscribers in your application. If you are familiar with Promise based async programming then Observables offer significant benefits. The thing with Observables is that it can deliver multiple values over time. In our case our HTTP call returns an Observable. The thing with Observable here is that if we don't subscribe nothing will happen. If we subscribe multiple times to these observables, multiple HTTP requests will be triggered. As mentioned above Observables can return stream of values over time, in our case when HTTP call is successfully then it will emit only one value and then completes. If HTTP request fails then this Observable will emit error so that we can catch and take appropriate action.

In our code when we call get() method, it return an Observable, then we are subscribing to that Observable, HttpClient.get() method parses the JSON server response into the anonymous Object type. HTTP library by default assumes that we have queried a JSON API and it internally parses the HTTP response body as JSON.

> Note: We are using HttpClient directly here in Component which is not best practice, we are going to create services later and refactor our code to make it more maintainable and testable.

Now lets run our app and go to http://localhost:4200/genres and open Chrome Developer tools by pressing F12 and go to Console tab. You should see following message

Access to XMLHttpRequest at https://localhost:44312/api/genres from origin http://localhost:4200 has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

This is exactly the message we are expecting because CORS is not enabled in our API.

Enabling CORS in our ASP.NET Core API

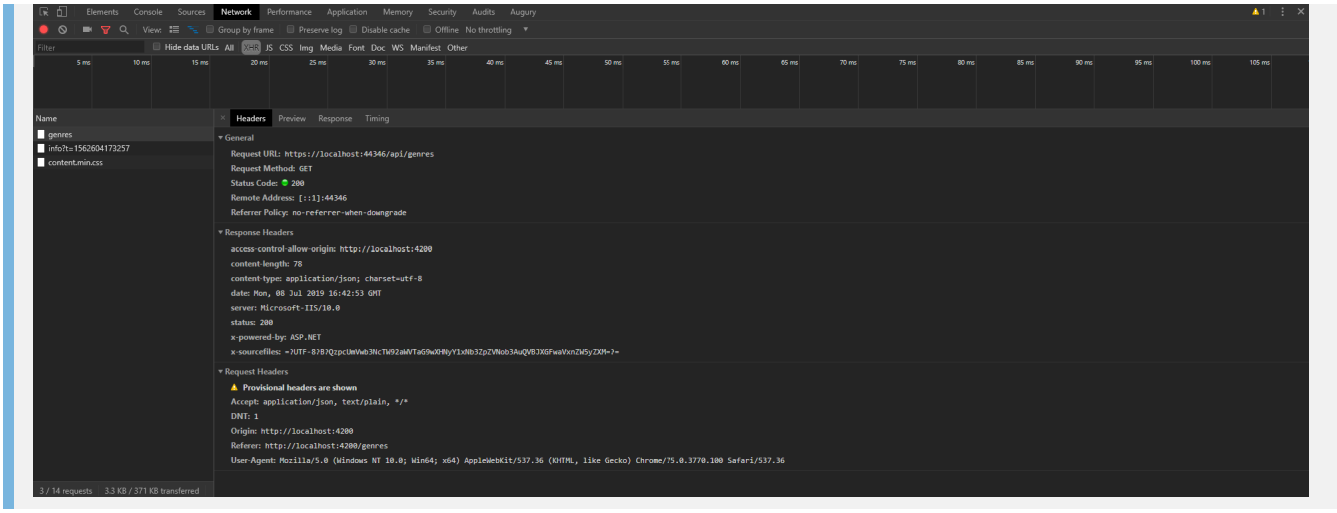
There are multiple ways that we can enable CORS in or API. For our scenario we need to enable CORS for our localhost and allow it for all HTTP methods (GET, POST, PUT and DELETE).

Now, lets go to the Startup.cs in or API and write the following code

```
        // Sets the policy name to "_myAllowSpecificOrigins". The policy name is
arbitrary.
```

```
        private readonly string _myAllowSpecificOrigins =
"_myAllowSpecificOrigins";

// the below code needs to be written in our ConfigureServices method
//  The AddCors method call adds CORS services to the app's service container. The
AddCors method takes lambda expression takes a CorsPolicyBuilder object
        services.AddCors(options =>
            {
                options.AddPolicy(_myAllowSpecificOrigins,
                                  builder =>
                                  {

builder.WithOrigins("http://localhost:4200").AllowAnyHeader().AllowAnyMethod();
                                  });
            });

// UseCors extension method, which enables CORS in Configure(IApplicationBuilder
app, IHostingEnvironment env) method

     app.UseCors(_myAllowSpecificOrigins);
```

Now lets run pur API and refresh the Angular App at http://localhost:4200/genres, you should see the 3 genres in the console window of the Chrome Dev tools. Also if you look at the Network Tab and click the genres item and inspect the Headers tab in the Request Headers you should see the **Origin :** http://localhost:4200 and in Response Headers we should see **access-control-allow-origin:** http://localhost:4200



Using *ngFor Directive for displaying some test data

In Angular you can think of **Directives** as probably most important bit that we use in building our application. Usually we say that Components are building blocks of any Angular application, but really internally they are nothing but directives with Views. So you can think of Directives as some piece of functionality that are used to extend the power of HTML by teaching it new tricks. Angular as many built-in directives that we can use (*ngFor, *ngIf etc) and at the same time we can build our own custom directives to power up our application.

In Angular there are three kinds of Directives

- Components—directives with a template
- Structural directives—change the DOM layout by adding and removing DOM elements, for example **NgFor** and **NgIf**
- Attribute directives—change the appearance or behavior of an element, component, or another directive

NgFor is a repeater directive—a way to present a list of items. You can use ngFor typically to display lists, table rows etc in your template. Lets use ngFor to display list of Genres in our genres template.

```
<ul>
  <li *ngFor="let genre of genres">{{genre.name}}</li>
</ul>
```

You should see our 3 genres in our view.

Wrapping Up