**Chat Application Documentation:**

## I am writing about writing a document version of Angular that practically killed Microsoft on the client side. Technologies like ASP.Net, Web Forms, replaced by a JavaScript framework that isn't exactly Microsoft. However, since the second version of Angular, Microsoft and Google have worked together to create Angular 10. About the architecture:

You will build an Angular 10 client that consumes a RESTful Web API Core 2 service.

The client side:

* Angular 10
* Angular CLI
* Angular Material

The server side:

* .NET C# Web API Core 2
* Injection dependencies
* [JWT authentication](https://www.toptal.com/angular/angular-6-jwt-authentication)
* Entity framework code first
* SQL Server

**What Do You Need :**

Let’s start by choosing the IDE. Of course, this is just my preference, and you can use the one you feel more comfortable with. In my case, I will use Visual Studio Code and Visual Studio 2017.Why two different IDEs? Since Microsoft created Visual Studio Code for the front end, I cannot stop using this IDE. Anyway, we will also see how to integrate Angular 5 inside the solution project, that will help you if you are the kind of developer who prefers to debug both back end and front with just one F5. About the back end, you can install the latest Visual Studio 2017 version which has a free edition for developers but is very complete: Community.

**So, here the list of things we need to install for this project:**

1. Visual Studio Code
2. Visual Studio 2017 Community (or Any)
3. Node.js v8.10.0
4. Microsoft SQL Server 2017

**The Front End:**

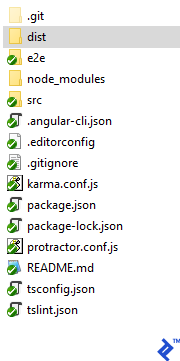
Quick Start

Let the fun begin! The first thing we need to do is install Angular CLI globally, so open the node.js command prompt and run this command: npm install -g @angular/cli   
Okay, now we have our module bundler. This usually installs the module under your user folder. An alias should not be necessary by default, but if you need it you can execute the next line:

**alias ng="<UserFolder>/.npm/lib/node\_modules/angular-cli/bin/ng".** The next step is to create the new project. I will call it angular5-app. First, we navigate to the folder under which we want to create the site, and then:  
ng new angular5-app

**First Build:** While you can test your new website just running ng serve --open, I do recommend testing the site from your favorite web service. Why? Well, some issues can happen only in production, and building the site with ng build is the closest way to approach this environment. Then we can open the folder angular5-app with Visual Studio Code and run ng build on the terminal bash:  
 

A new folder called dist will be created and we can serve it using IIS or whichever web server you prefer. Then you can type the URL in the browser, and…done!



My have components where we will create for each Angular component the css, ts, spec, and html files. We will also create a config folder to keep the site configuration, directives will have all our custom directives, helpers will house common code like the authentication manager, layout will contain the main components like body, head, and side panels, models keeps what will match with the back-end view models, and finally services will have the code for all the calls to the back end.

Outside the app folder we will keep the folders created by default, like assets and environments, and also the root files.

src folder is structured as follows: Inside the app folder we

**Creating the Configuration File:**

Let’s create a config.ts file inside our config folder and call the class AppConfig. This is where we can set all the values we will use in different places in our code; for instance, the URL of the API. Note that the class implements a get property which receives, as a parameter, a key/value structure and a simple method to get access to the same value. This way, it will be easy to get the values just calling this.config.setting['PathAPI'] from the classes that inherit from it.

**Angular Material**

Before starting the layout, let’s set up the UI component framework. Of course, you can use others like Bootstrap, but if you like the styling of Material, I do recommend it because it’s also supported by Google.

To install it, we just need to run the next three commands, which we can execute on the Visual Studio Code terminal:

npm install --save @angular/material @angular/cdk

npm install --save @angular/animations

npm install --save hammerjs

The second command is because some Material components depend on Angular Animations. I also recommend reading [the official page](https://angular.io/guide/animations) to understand which browsers are supported and what a polyfill is.

The third command is because some Material components rely on HammerJS for gestures. Now we can proceed to import the component modules we want to use in our app.module.ts file:

**import** {MatButtonModule, MatCheckboxModule} **from** '@angular/material';

**import** {MatInputModule} **from** '@angular/material/input';

**import** {MatFormFieldModule} **from** '@angular/material/form-field';

**import** {MatSidenavModule} **from** '@angular/material/sidenav';

*// ...*

**@NgModule**({

imports: [

BrowserModule,

BrowserAnimationsModule,

MatButtonModule,

MatCheckboxModule,

MatInputModule,

MatFormFieldModule,

MatSidenavModule,

AppRoutingModule,

HttpClientModule

],

Next step is to change the style.css file, adding the kind of theme you want to use:

**@import** "~@angular/material/prebuilt-themes/deeppurple-amber.css";

Now import HammerJS by adding this line in the main.ts file:

**import** 'hammerjs';

And finally all we’re missing is to add the Material icons to index.html, inside the head section:

<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

The idea is to open/hide the menu by clicking on some button on the header. Angular Responsive will do the rest of the work for us. To do this we will create a layout folder and put inside it the app.component files created by default. But we will also create the same files for each section of the layout like you can see in the next image. Then, app.component will be the body, head.component the header, and left-panel.component the menu.

Now let’s change app.component.html as follows:

<div \*ngIf="authentication">

<app-head></app-head>

<button type="button" mat-button (click)="drawer.toggle()">

Menu

</button>

<mat-drawer-container class="example-container" autosize>

<mat-drawer #drawer class="example-sidenav" mode="side">

<app-left-panel></app-left-panel>

</mat-drawer>

<div>

<router-outlet></router-outlet>

</div>

</mat-drawer-container>

</div>

<div \*ngIf="!authentication"><app-login></app-login></div>

Basically we will have an authentication property in the component which will allow us to remove the header and the menu if the user is not logged in, and instead, show a simple login page.

The head.component.html looks like this:

<h1>{{title}}</h1>

<button mat-button [routerLink]=" ['./logout'] ">Logout!</button>

Just a button to log the user out—we will come back to this again later. As for left-panel.component.html, for now just change the HTML to:

<nav>

<a routerLink="/dashboard">Dashboard</a>

<a routerLink="/users">Users</a>

</nav>

We’ve kept it simple: So far it’s just two links to navigate through two different pages. (We will also return to this later.)

Now, this is what the head and the left-side component TypeScript files look like:

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

**@Component**({

selector: 'app-head',

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

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

})

**export** **class** HeadComponent {

title = 'Angular 5 Seed';

}

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

**@Component**({

selector: 'app-left-panel',

templateUrl: './left-panel.component.html',

styleUrls: ['./left-panel.component.css']

})

**export** **class** LeftPanelComponent {

title = 'Angular 5 Seed';

}

But what about the TypeScript code for app.component? We will leave a little mystery here and pause it for a while, and come back to this after implementing authentication.

**Authentication:**

I had the class AuthGuard implemented to set the routing configuration? Every time we navigate to a different page we will use this class to verify if the user is authenticated with a token. If not, we’ll redirect automatically to the login page. The file for this is canActivateAuthGuard.ts—create it inside the helpers folder and have it look like this:

**import** { CanActivate, Router } **from** '@angular/router';

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

**import** { Observable } **from** 'rxjs/Observable';

**import** { Helpers } **from** './helpers';

**import** { ActivatedRouteSnapshot, RouterStateSnapshot } **from** '@angular/router';

**@Injectable**()

**export** **class** AuthGuard **implements** CanActivate {

**constructor**(**private** router: Router, **private** helper: Helpers) {}

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | boolean {

**if** (!**this**.helper.isAuthenticated()) {

**this**.router.navigate(['/login']);

**return** false;

}

**return** true;

}

}

So every time I change the page the method canActivate will be called, which will check if the user is authenticated, and if not, we use our Router instance to redirect to the login page. But what is this new method on the Helper class? Under the helpers folder let’s create a file helpers.ts. Here we need to manage localStorage, where we will store the token we get from the back end.

**Note**

|  |
| --- |
| Regarding localStorage, you can also use cookies or sessionStorage, and the decision will depend on the behavior we want to implement. As the name suggests, sessionStorage is only available for the duration of the browser session, and is deleted when the tab or window is closed; it does, however, survive page reloads. If the data you are storing needs to be available on an ongoing basis, then localStorage is preferable to sessionStorage. Cookies are primarily for reading server-side, whereas localStorage can only be read client-side. So the question is, in your app, who needs this data---the client or the server? |

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

**import** { Observable } **from** 'rxjs';

**import** { Subject } **from** 'rxjs/Subject';

**@Injectable**()

**export** **class** Helpers {

**private** authenticationChanged = **new** Subject<boolean>();

**constructor**() {

}

**public** isAuthenticated():boolean {

**return** (!(window.localStorage['token'] === undefined ||

window.localStorage['token'] === null ||

window.localStorage['token'] === 'null' ||

window.localStorage['token'] === 'undefined' ||

window.localStorage['token'] === ''));

}

**public** isAuthenticationChanged():any {

**return** **this**.authenticationChanged.asObservable();

}

**public** getToken():any {

**if**( window.localStorage['token'] === undefined ||

window.localStorage['token'] === null ||

window.localStorage['token'] === 'null' ||

window.localStorage['token'] === 'undefined' ||

window.localStorage['token'] === '') {

**return** '';

}

**let** obj = JSON.parse(window.localStorage['token']);

**return** obj.token;

}

**public** setToken(data:any):void {

**this**.setStorageToken(JSON.stringify(data));

}

**public** failToken():void {

**this**.setStorageToken(undefined);

}

**public** logout():void {

**this**.setStorageToken(undefined);

}

**private** setStorageToken(value: any):void {

window.localStorage['token'] = value;

**this**.authenticationChanged.next(**this**.isAuthenticated());

}

}

Is our authentication code making sense now? We’ll come back to the Subject class later, but right now let’s circle back for a minute to the routing configuration. Take a look at this line:

{ path: 'logout', component: LogoutComponent},

This is our component to log out of the site, and it’s just a simple class to clean out the localStorage. Let’s create it under the components/login folder with the name of logout.component.ts:

**import** { Component, OnInit } **from** '@angular/core';

**import** { Router } **from** '@angular/router';

**import** { Helpers } **from** '../../helpers/helpers';

**@Component**({

selector: 'app-logout',

template:'<ng-content></ng-content>'

})

**export** **class** LogoutComponent **implements** OnInit {

**constructor**(**private** router: Router, **private** helpers: Helpers) { }

ngOnInit() {

**this**.helpers.logout();

**this**.router.navigate(['/login']);

}

}

So every time we go to the URL /logout, the localStorage will be removed and the site will redirect to the login page. Finally, let’s create login.component.ts like this:

**import** { Component, OnInit } **from** '@angular/core';

**import** { Router } **from** '@angular/router';

**import** { TokenService } **from** '../../services/token.service';

**import** { Helpers } **from** '../../helpers/helpers';

**@Component**({

selector: 'app-login',

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

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

})

**export** **class** LoginComponent **implements** OnInit {

**constructor**(**private** helpers: Helpers, **private** router: Router, **private** tokenService: TokenService) { }

ngOnInit() {

}

login(): void {

**let** authValues = {"Username":"pablo", "Password":"secret"};

**this**.tokenService.auth(authValues).subscribe(token => {

**this**.helpers.setToken(token);

**this**.router.navigate(['/dashboard']);

});

}

}

As you can see, for the moment we’ve hard-coded our credentials here. Note that here we are calling a service class; we will create these services classes to get access to our back end in the next section.

Finally, we need to go back to the app.component.ts file, the layout of the site. Here, if the user is authenticated, it will show the menu and header sections, but if not, the layout will change to show just our login page.

**export** **class** AppComponent **implements** AfterViewInit {

subscription: Subscription;

authentication: boolean;

**constructor**(**private** helpers: Helpers) {

}

ngAfterViewInit() {

**this**.subscription = **this**.helpers.isAuthenticationChanged().pipe(

startWith(**this**.helpers.isAuthenticated()),

delay(0)).subscribe((value) =>

**this**.authentication = value

);

}

title = 'Angular 5 Seed';

ngOnDestroy() {

**this**.subscription.unsubscribe();

}

}

Remember the Subject class in our helper class? This is an Observable. Observables provide support for passing messages between publishers and subscribers in your application. Every time the authentication token changes, the authentication property will be updated. Reviewing the app.component.html file, it will probably make more sense now:

<div \*ngIf="authentication">

<app-head></app-head>

<button type="button" mat-button (click)="drawer.toggle()">

Menu

</button>

<mat-drawer-container **class**="example-container" autosize>

<mat-drawer #drawer **class**="example-sidenav" mode="side">

<app-left-panel></app-left-panel>

</mat-drawer>

<div>

<router-outlet></router-outlet>

</div>

</mat-drawer-container>

</div>

<div \*ngIf="!authentication"><app-login></app-login></div>

Services

At this point we are navigating to different pages, authenticating our client side, and rendering a very simple layout. But how we can get data from the back end? I strongly recommend doing all back-end access from *service* classes in particular. Our first service will be inside the services folder, called token.service.ts:

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

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

**import** { Observable } **from** 'rxjs/Observable';

**import** { of } **from** 'rxjs/observable/of';

**import** { catchError, map, tap } **from** 'rxjs/operators';

**import** { AppConfig } **from** '../config/config';

**import** { BaseService } **from** './base.service';

**import** { Token } **from** '../models/token';

**import** { Helpers } **from** '../helpers/helpers';

**@Injectable**()

**export** **class** TokenService **extends** BaseService {

**private** pathAPI = **this**.config.setting['PathAPI'];

**public** errorMessage: string;

**constructor**(**private** http: HttpClient, **private** config: AppConfig, helper: Helpers) { **super**(helper); }

auth(data: any): any {

**let** body = JSON.stringify(data);

**return** **this**.getToken(body);

}

**private** getToken (body: any): Observable<any> {

**return** **this**.http.post<any>(**this**.pathAPI + 'token', body, **super**.header()).pipe(

catchError(**super**.handleError)

);

}

}

The first call to the back end is a POST call to the token API. The token API does not need the token string in the header, but what happen if we call another endpoint? As you can see here, TokenService (and service classes in general) inherit from the BaseService class. Let’s take a look at this:

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

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

**import** { Observable } **from** 'rxjs/Observable';

**import** { of } **from** 'rxjs/observable/of';

**import** { catchError, map, tap } **from** 'rxjs/operators';

**import** { Helpers } **from** '../helpers/helpers';

**@Injectable**()

**export** **class** BaseService {

**constructor**(**private** helper: Helpers) { }

**public** extractData(res: Response) {

**let** body = res.json();

**return** body || {};

}

**public** handleError(error: Response | any) {

*// In a real-world app, we might use a remote logging infrastructure*

**let** errMsg: string;

**if** (error **instanceof** Response) {

**const** body = error.json() || '';

**const** err = body || JSON.stringify(body);

errMsg = `${error.status} - ${error.statusText || ''} ${err}`;

} **else** {

errMsg = error.message ? error.message : error.toString();

}

console.error(errMsg);

**return** Observable.throw(errMsg);

}

**public** header() {

**let** header = **new** HttpHeaders({ 'Content-Type': 'application/json' });

**if**(**this**.helper.isAuthenticated()) {

header = header.append('Authorization', 'Bearer ' + **this**.helper.getToken());

}

**return** { headers: header };

}

**public** setToken(data:any) {

**this**.helper.setToken(data);

}

**public** failToken(error: Response | any) {

**this**.helper.failToken();

**return** **this**.handleError(Response);

}

}

So every time we make an HTTP call, we implement the header of the request just using super.header. If the token is in localStorage then it will be appended inside the header, but if not, we will just set the JSON format. Another thing we can see here is what happens if authentication fails.

The login component will call the service class and the service class will call the back end. Once we have the token, the helper class will manage the token, and now we are ready to get the list of users from our database. To get data from the database, first be sure we match the model classes with the back-end view models in our response.

In user.ts:

**export** **class** User {

id: number;

name: string;

}

And we can create now the user.service.ts file:

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

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

**import** { Observable } **from** 'rxjs/Observable';

**import** { of } **from** 'rxjs/observable/of';

**import** { catchError, map, tap } **from** 'rxjs/operators';

**import** { BaseService } **from** './base.service';

**import** { User } **from** '../models/user';

**import** { AppConfig } **from** '../config/config';

**import** { Helpers } **from** '../helpers/helpers';

**@Injectable**()

**export** **class** UserService **extends** BaseService {

**private** pathAPI = **this**.config.setting['PathAPI'];

**constructor**(**private** http: HttpClient, **private** config: AppConfig, helper: Helpers) { **super**(helper); }

*/\*\* GET heroes from the server \*/*

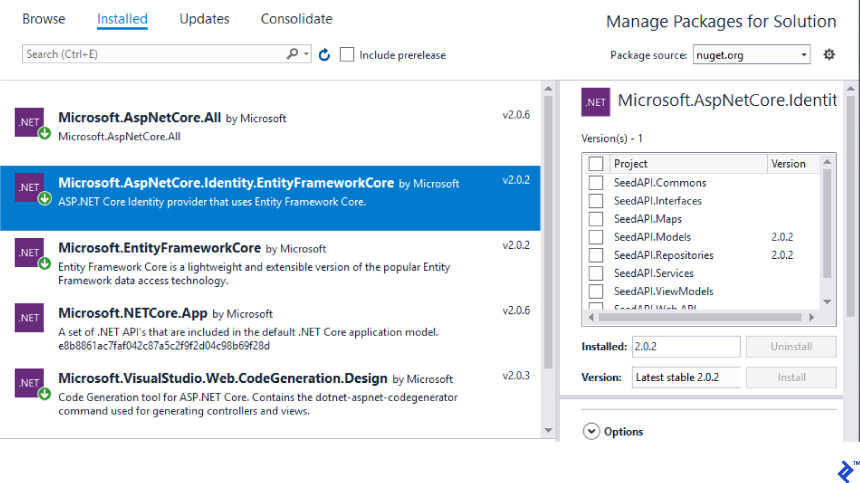
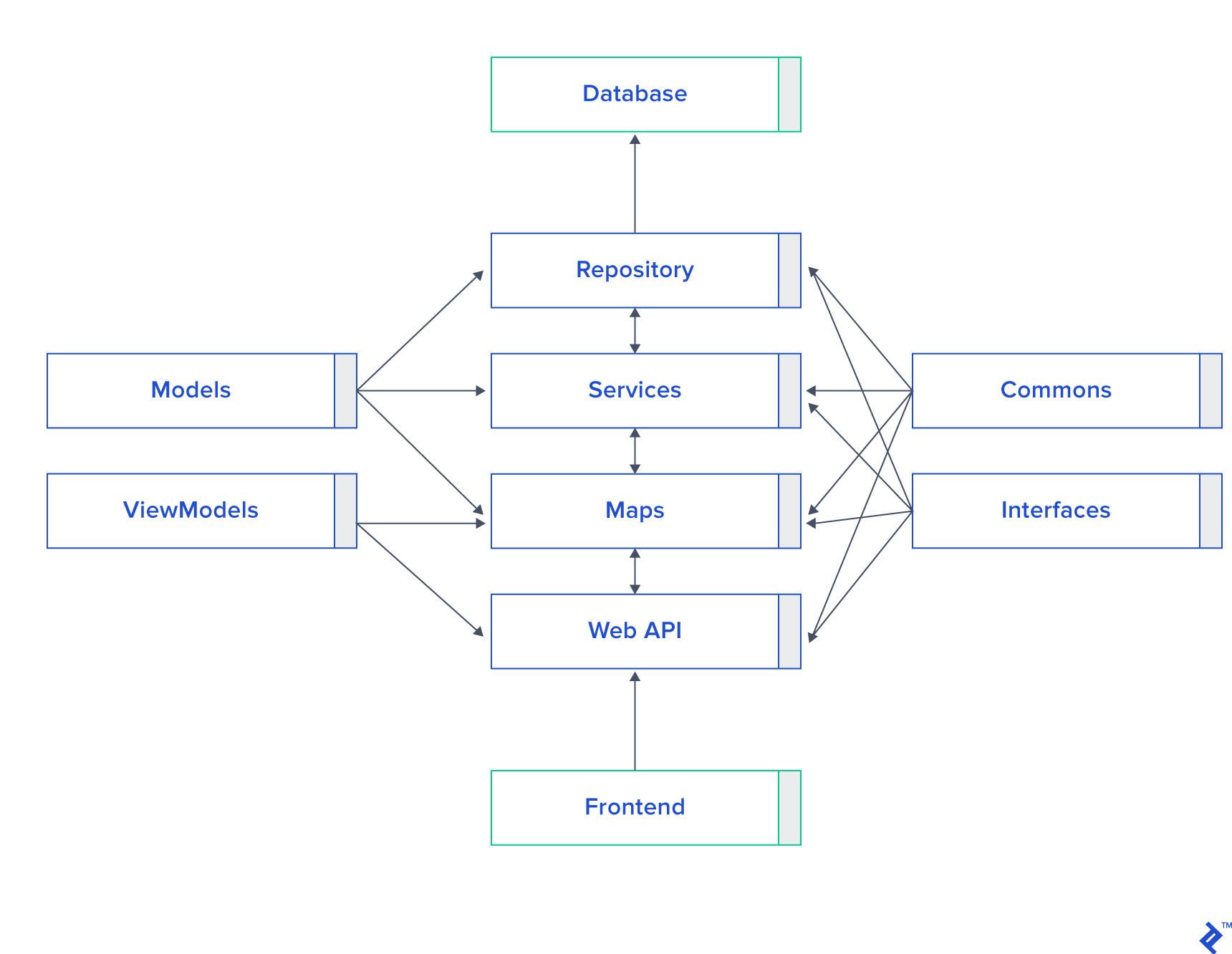
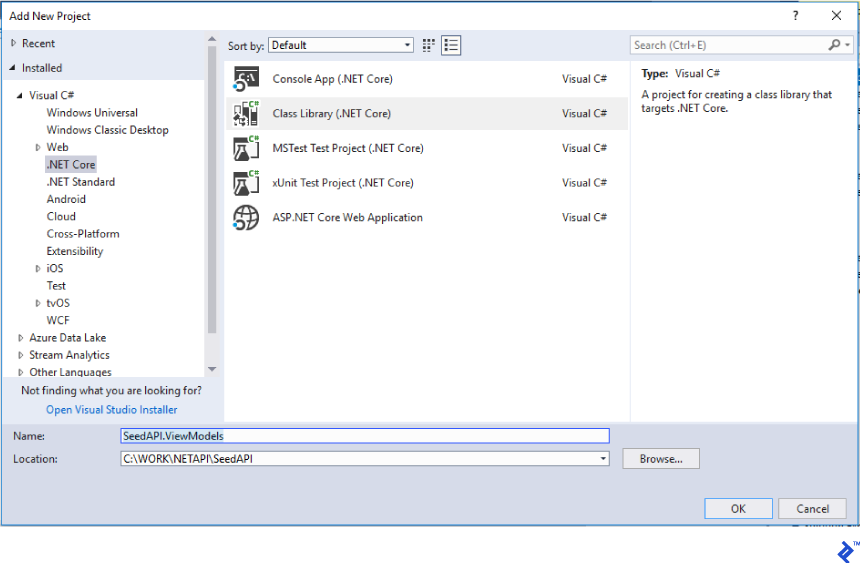
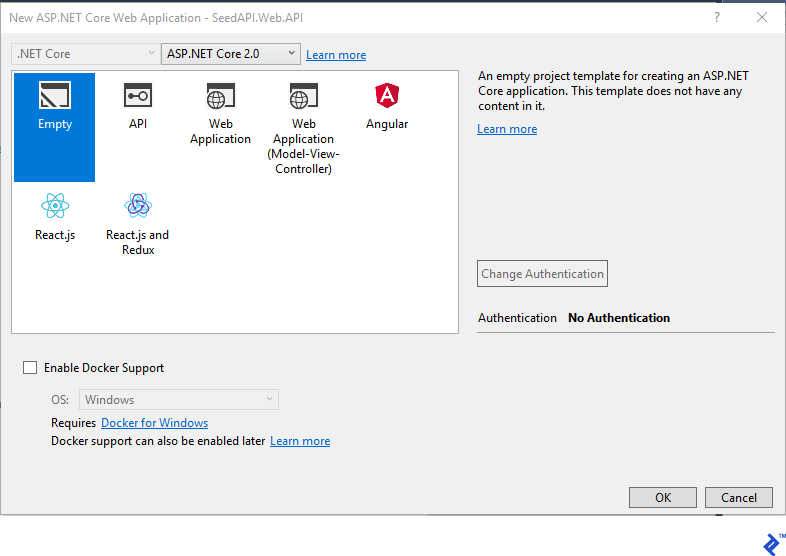
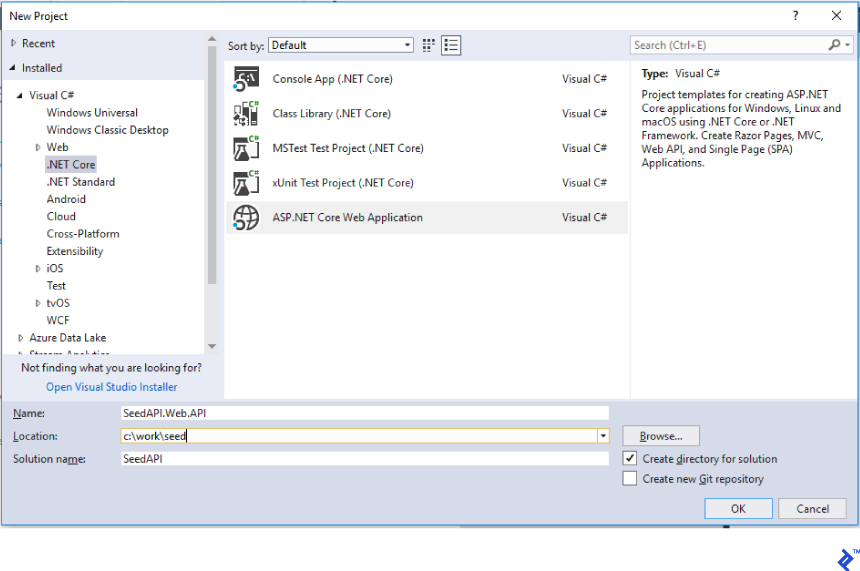
getUsers (): Observable<User[]> {

**return** **this**.http.get(**this**.pathAPI + 'user', **super**.header()).pipe(

catchError(**super**.handleError));

}

**The Back End:**

****