# How to Integrate Angular with .NET Core Project

In this article, we are going to show you how to integrate the server side .NET Core project with the client side Angular project. We are going to start with the server side first by creating a layer to fetch data from a database and creating an endpoint that will process the client side request and return fetched data from a database.

After we are finished with the server side of the application, we are going to continue our work on the client part. We are going to create components to process and display the data fetched from the server with a simple interface. For that purpose, we are going to use a service to create a centralized place to handle our HTTP requests.

To create a server side of our project, we are going to use Visual Studio 2017 and for a client part Visual Studio Code editor.

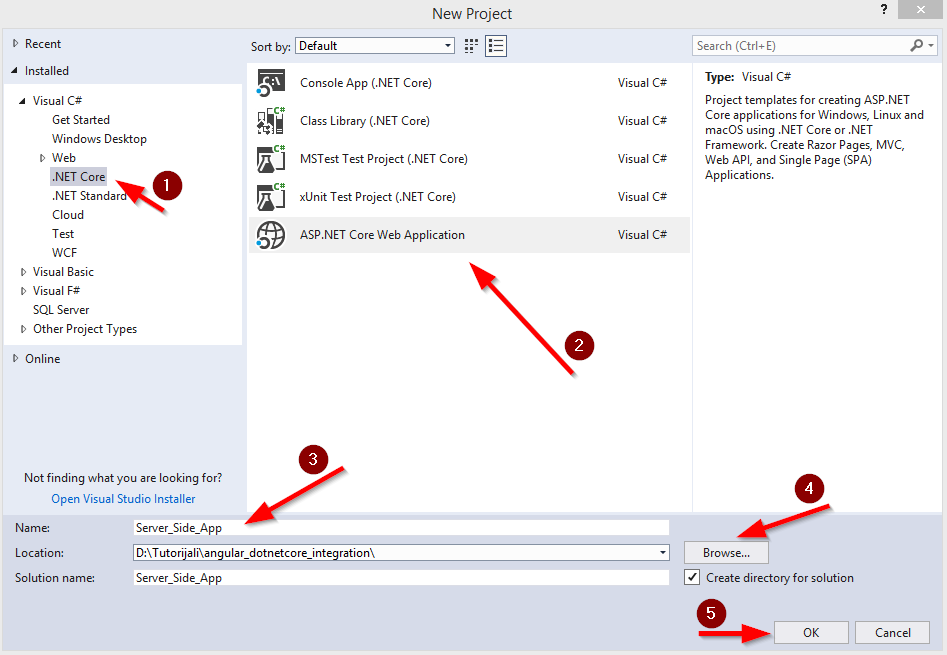
To download the server-side source code check out ……..

For the client side code check out ….

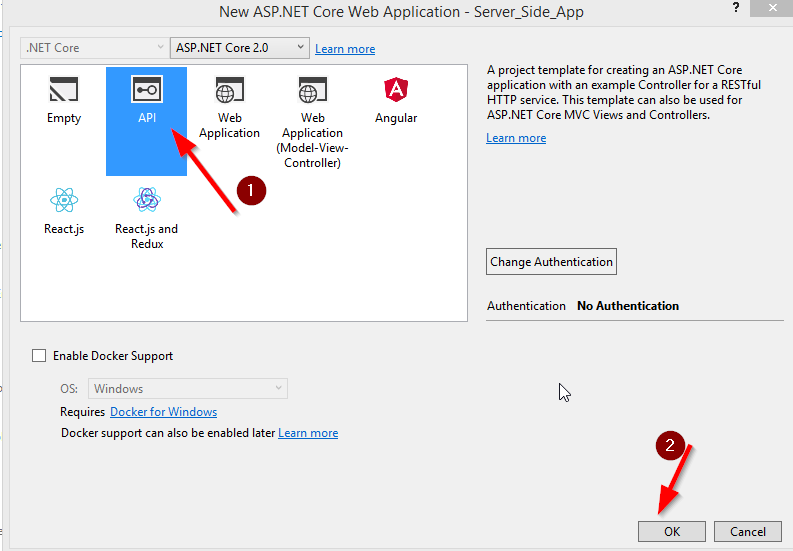
So, let's get to the action.

## Project Preparation

Let’s start by creating a new ASP.NET Core Web Application:

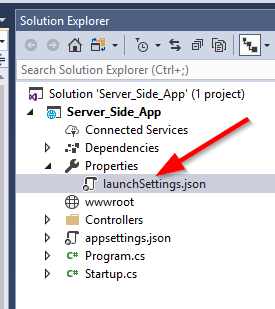


After we click on the OK button, we are going to choose the Web API project:



After several seconds, the project is prepared for us.

Now, let’s open the launchSettings.json file to modify the endpoint url address and to disable our browser to start as soon as the application starts:



{

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:5000/",

"sslPort": 0

}

},

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": false,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"Server\_Side\_App": {

"commandName": "Project",

"launchBrowser": false,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

},

"applicationUrl": "http://localhost:5000/"

}

}

}

Because the server side and the client side projects will run on different domains we need to enable CORS on the server side project. To do that we need to open the Startup.cs class and to modify the ConfigureServices and Configure methods:

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(options =>

{

options.AddPolicy("CorsPolicy",

builder => builder.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

});

services.AddMvc();

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseCors("CorsPolicy");

app.UseMvc();

}

To learn more about CORS configuration and the project configuration overall, you can read the [.NET Core Project Configuration article.](https://code-maze.com/net-core-web-development-part2/)

## Configuring Entity Framework Core

We are going to use EF Core library to enable access to a database from our server application. It was installed automatically during the application creation process.

So, let’s create a new folder Models and a new class Movie inside it:

[Table("Movie")]

public class Movie

{

[Key]

public Guid Id { get; set; }

public string Name { get; set; }

public string Genre { get; set; }

public string Director { get; set; }

}

This class represents our Movie table inside a database.

Now, we need to create a context class MovieContext.cs in our project, which will be a middleware for a database communication:

public class MovieContext : DbContext

{

public MovieContext(DbContextOptions options)

: base(options)

{

}

public DbSet<Movie> Movies { get; set; }

}

After that, we need to provide a data base connection string inside the appsettings.json file:

{

"Logging": {

"IncludeScopes": false,

"Debug": {

"LogLevel": {

"Default": "Warning"

}

},

"Console": {

"LogLevel": {

"Default": "Warning"

}

}

},

"ConnectionStrings": {

"sqlConString": "Server=.;Database=CodeMaze;Trusted\_Connection=True;"

}

}

Finally, we need to register our context class in the Startup.cs class:

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(options =>

{

options.AddPolicy("CorsPolicy",

builder => builder.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

});

services.AddDbContext<MovieContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("sqlConString")));

services.AddMvc();

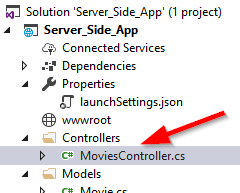
}

Excellent.

We have finished with configuring EF Core in our application. We didn’t dive any deeper than we should into the registration logic, but if you like to read in more detail about it you can read this article [.NET Core 2.0, Angular and MySQL. Repository Pattern](https://code-maze.com/net-core-web-development-part4/).

## Creating Controller and the Get Action

Now is the time to create our action that will represent an endpoint for the client request. In the SolutionExplorer we can find the Controllers folder and one class inside it. We are going to rename that class to MoviesController and to remove all the code except the first Get action:



[Route("api/[controller]")]

public class MoviesController : Controller

{

// GET api/values

[HttpGet]

public IEnumerable<string> Get()

{

return new string[] { "value1", "value2" };

}

}

Let’s modify our MoviesController class to fetch data from the data base by using our previousely created context class:

[Route("api/[controller]")]

public class MoviesController : Controller

{

private readonly MovieContext \_context;

public MoviesController(MovieContext context)

{

\_context = context;

}

// GET api/values

[HttpGet]

public IActionResult Get()

{

try

{

var movies = \_context.Movies.ToList();

return Ok(movies);

}

catch (Exception ex)

{

return StatusCode(500, $"Something went wrong: {ex.Message}");

}

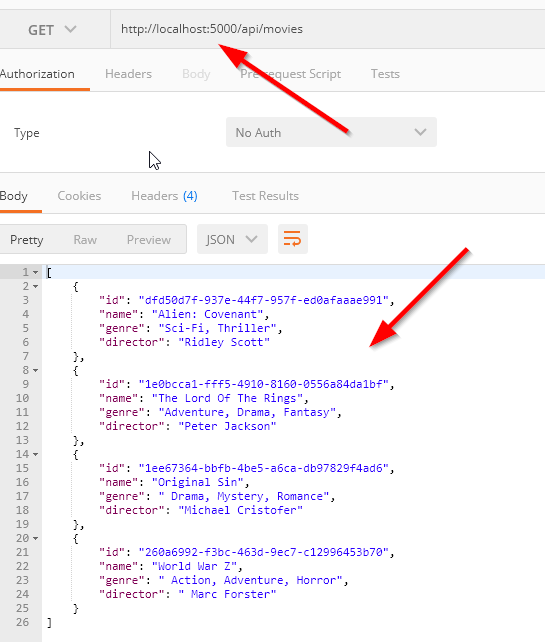
}

}

Awesome.

# With this code in place, we have finished with the server side. Of course, we didn’t dive deep inside the action logic due to the sake of simplicity, but to read in more detail about all this stuff you can read [.NET Core 2.0, Angular and MySQL. Get Requests](https://code-maze.com/net-core-web-development-part5/).

Now we can test our Web API by sending a request from Postman or any other tool you like:



Now we are ready to move on the Angular part.

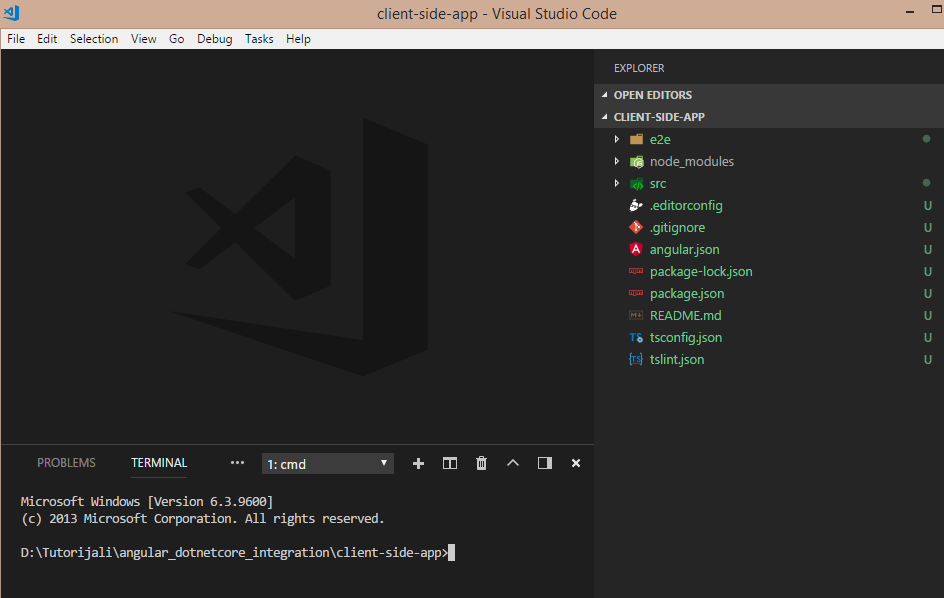
## Creating an Angular Project

We are going to use Angular CLI to help us with the project creation and the creation of our components in the project. To learn more about Angular CLI, you can read [Angular CLI Installation and Starting a New Project.](https://code-maze.com/net-core-web-development-part7/#installationAngularCLI)

Let’s open the command prompt window, navigate to the location we want our application in and type the command:

ng new client-side-app

After some time, our project is ready. Let’s open it inside the Visual Studio Code editor and start working on it:



## Modifying the App Component

Let’s modify the app.component.html file first:

<div class="container">

<div class="content">

<h1 class="headerText">Welcome to the movies presentation!!!</h1>

<p>

<span>Click this button to see the movie list:</span>

<button type="button" name="show" (click)="getMovies()">Show Movies</button>

</p>

</div>

</div>

Then, the app.component.css file as well:

.container{

width: 100%;

}

.content{

width: 70%;

margin: 0 auto;

border:1px solid gray;

padding: 10px;

box-shadow: 1px 1px 1px gray;

}

.headerText{

color: #2795ca;

text-align: center;

}

p{

text-align: center;

}

span{

font-size: 18px;

margin-right: 15px;

}

button{

background-color: #2795ca;

color:white;

height: 35px;

border: 1px solid #2795ca;

border-radius: 3px;

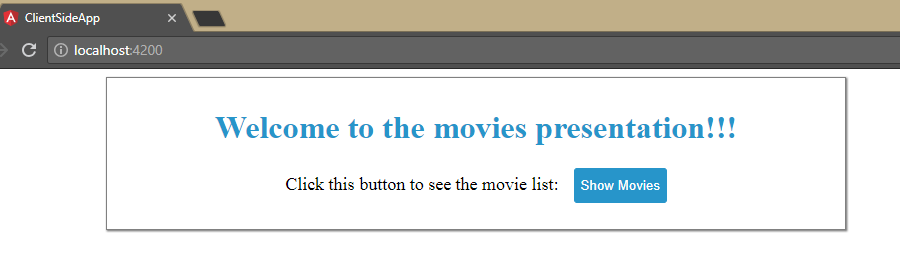
}

button:hover{

cursor: pointer;

}

If we start our application now by typing ng serve in the console window of Visual Studio Code, we will see this result:



When we click on the Show Movies button we should be able to see all of our movie entities on the page. So, let’s start working on that feature.

## Using Services and HttpClientModule

To send a request to our endpoint, we are going to introduce the HttpClientModule in our app. So we need to import it inside the app.module.ts file and to place it in the imports array:

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

import { NgModule } from '@angular/core';

import { HttpClientModule} from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

After the module modification, we are going to create a new folder \_interfaces and to create one interface file movie.model.ts inside that folder:

export interface Movie{

id: string,

name: string,

genre: string,

director: string

}

Then, we are going to create a new service file http-service by using the Angular CLI command:



That looks great.

So, let’s modify that service file:

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

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

@Injectable({

providedIn: 'root'

})

export class HttpService {

constructor(private httpService: HttpClient) { }

public getData = (route: string) =>{

return this.httpService.get(route);

}

}

We have the getData function, which we are going to call once we require a data from a certain endpoint from the server. You can read [here](https://code-maze.com/net-core-web-development-part9/) in more detail about services and HTTP calls.

The getData function will return an Observable as a response, so if we want to fetch that data from a response we need to subscribe on this function. So, let’s do exactly that.

## Fetching and Displaying Data from the Server

Let’s modify the app.component.ts file first:

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

import { HttpService } from './services/http.service';

import { Movie } from './\_interfaces/movie.model';

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

public movies: Movie[];

constructor(private httpService: HttpService){}

public getMovies = () => {

let route: string = 'http://localhost:5000/api/movies';

this.httpService.getData(route)

.subscribe((result) => {

this.movies = result as Movie[];

},

(error) => {

console.error(error);

});

}

}

In order to display our data, we need to modify the app.component.html file by adding this code below the p tag:

<div \*ngIf="movies" class="table-center">

<table>

<thead>

<tr>

<th>Name</th>

<th>Genre</th>

<th>Director</th>

</tr>

</thead>

<tbody>

<tr \*ngFor='let movie of movies'>

<td>{{movie?.name}}</td>

<td>{{movie?.genre}}</td>

<td>{{movie?.director}}</td>

</tr>

</tbody>

</table>

</div>

Finally, let's modify our app.component.css file:

.table-center{

margin: 20px auto;

}

table{

width: 100%;

}

table, th, td{

border: 1px solid #ddd;

border-collapse: collapse;

text-align: center;

}

tr:nth-child(even){

background-color: #f2f2f2;

}

th {

padding-top: 8px;

padding-bottom: 8px;

background-color: #2795ca;

color: white;

}

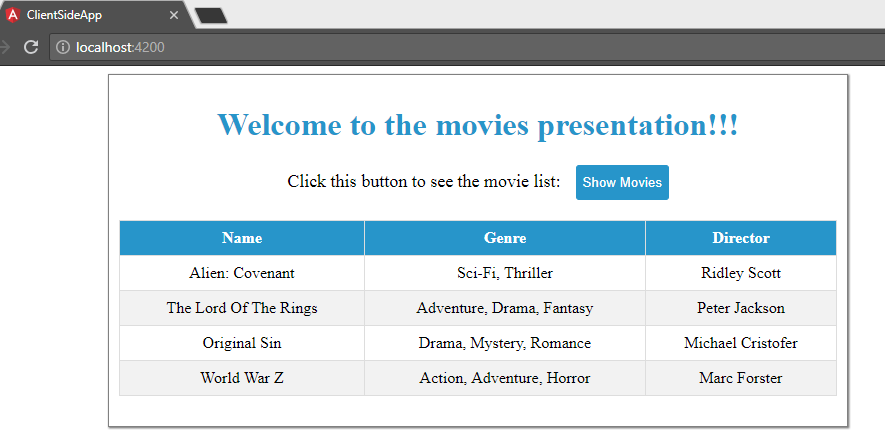
th, td{

padding: 8px;

}

Excellent.

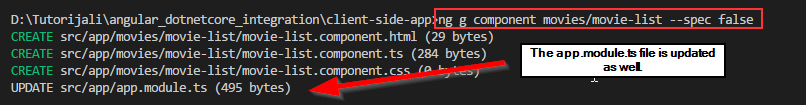
Now if we start our application again and click on the Show Movies button, we will see this result:



All works great, and our application is almost finished. It is always a good solution to extract part of a component into another reusable component. Maybe later on, we would like to reuse our movie list in some other component, therefore, extracting the movie list into its own component is a good solution.

## Extracting the MovieList Component

First, let’s create the MovieList component:



Then, let’s move complete table cod from the app.component.html file to the movie-list.component.ts file:

<table>

<thead>

<tr>

<th>Name</th>

<th>Genre</th>

<th>Director</th>

</tr>

</thead>

<tbody>

<tr \*ngFor='let movie of movies'>

<td>{{movie?.name}}</td>

<td>{{movie?.genre}}</td>

<td>{{movie?.director}}</td>

</tr>

</tbody>

</table>

After that, we are going to move the css code from the app.component.css file to the movie-list.component.css file:

table{

width: 100%;

}

table, th, td{

border: 1px solid #ddd;

border-collapse: collapse;

text-align: center;

}

tr:nth-child(even){

background-color: #f2f2f2;

}

th {

padding-top: 8px;

padding-bottom: 8px;

background-color: #2795ca;

color: white;

}

th, td{

padding: 8px;

}

After the css modification, let’s modify the movie-list.component.ts file by adding the @Input decorator:

import { Movie } from './../../\_interfaces/movie.model';

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

@Component({

selector: 'app-movie-list',

templateUrl: './movie-list.component.html',

styleUrls: ['./movie-list.component.css']

})

export class MovieListComponent implements OnInit {

@Input() public movies: Movie[];

constructor() { }

ngOnInit() {

}

The @Input decorator serve us to accept the data sent from the parent component and it must have the same name as the movies parameter inside the \*ngFor statement:

<tr \*ngFor='let movie of movies'>

Finally, we need to inject this component into the parent component, the app.component.ts file:

div class="container">

<div class="content">

<h1 class="headerText">Welcome to the movies presentation!!!</h1>

<p>

<span>Click this button to see the movie list:</span>

<button type="button" name="show" (click)="getMovies()">Show Movies</button>

</p>

<div \*ngIf="movies" class="table-center">

<app-movie-list [movies]='movies'></app-movie-list>

</div>

</div>

</div>

As we can see, we are using a selector from our movies-list component to inject it into the parent component. Furthermore, to send our data to the @Input movies decorator, we must place its name inside square brackets. Now, as soon as we fetch data into the movies property in the app component, that data will be transferred to the movies-list component via the movies attribute.

If we start our project again and click the Show Movies button, we are going to see the same result, but this time, we have much readable and reusable code.

## Conclusion

In this article, we have learned how to create the Angular project and to integrate it with the .NET Core Web API project.

Furthermore, we have learned:

* How to enable communication between the projects on different domains
* The way to use Entity Framework Core to access our database
* How to create an endpoint for our client requests
* To prepare an Angular application
* How to fetch data from the server by using the HttpClientModule
* The way to display our data by using @Input decorator in a reusable component

That is all. We hope that you have enjoyed reading this article and that you have found many useful pieces of information.