# .NET Core with SignalR and Angular – Real-Time Charts

SingalR is a library which helps us provide a real-time web functionality to our applications. This means that our server can push data to any connected client as soon as that data is provided, in a real time.

In this article, we are going to show you how to use SignalR with .NET Core and Angular through the practical example. We are going to simulate a real-time data flow by using a Timer in .NET Core and use that real-time data to change states of our Angular charts in real time as well. For this example, we are going to use only one-way communication (from the server to the client), but we will add an additional feature to the example, to show the two-way communication as well (client-server-client).

So, without further ado, let’s get started.

If you want to download a finished project, you can do that here ...

This article is divided into the following sections:

* Creating Projects and Basic Configuration
* SignalR Installation, Hub and Configuration
* Timer Implementation with DataManager and ChartController
* Angular Chart and SignalR Listener
* Sending Data via SignalR from the Client to the Server and Back

## Creating Projects and Basic Configuration

First thing first.

Let’s create both the .NET Core and Angular projects. We are going to name them RealTimeCharts\_Server and RealTimeCharts\_Client respectively. For the .NET Core project, we are going to choose 2.1 version Web API empty project and for the Angular side, we are creating Angular version 7 with no routings created and CSS for the styles.

As soon as projects are created, we are going to switch to the server side project and set up some basic configuration. To do that, let’s open the launchSettings.json file and modify it accordingly:

{

"$schema": "http://json.schemastore.org/launchsettings.json",

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:60967",

"sslPort": 44342

}

},

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": false,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"RealTimeCharts\_Server": {

"commandName": "Project",

"launchBrowser": false,

"applicationUrl": "https://localhost:5001;http://localhost:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

}

Our server-side project will run on the localhost:5001 and the client side will run on the localhost:4200, so in order to establish communication between those two, we need to enable CORS. Let’s open the Startup.cs class and modify it:

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(options =>

{

options.AddPolicy("CorsPolicy",

builder => builder.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseCors("CorsPolicy");

app.UseMvc();

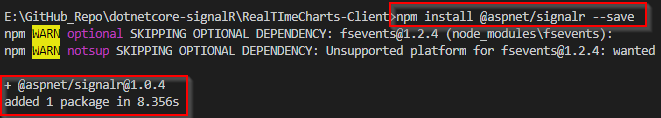
}

That is it regarding the configuration. Let’s move on to the next part.

## SignalR Installation, Hub and Configuration

The SignalR library is already installed in the .NET Core project as a part of the Microsoft.AspNetCore.App package, but we need to install it for the client side. To do that, we are going to open the Angular project in Visual Studio Code and type the following command in the terminal window:

npm install @aspnet/signalr –save



That is it for now for the client side. Let’s switch back to the server side project and create a new folder Models. In that folder, we are going to create a new class ChartModel and modify it:

public class ChartModel

{

public List<int> Data { get; set; }

public string Label { get; set; }

public ChartModel()

{

Data = new List<int>();

}

}

This form of data is expected by the Angular Charts library (which is yet to be installed), thus the model properties Data and Label.

Having the model prepared, we are going to continue by creating a new folder HubConfig and inside a new class ChartHub:

public class ChartHub: Hub

{

}

As we can notice, our ChartHub class must derive from the Hub class, which is a base class for the SignalR hub. But why do we need this ChartHub?

Well, a Hub is a high-level pipeline that allows communication between client and server to call each others methods directly. So basically, a Hub is a communication foundation between client and server while using SignalR.

Right now our ChartHub class is empty because we don’t need any methods inside it, yet.

To complete the SignalR configuration, let’s modify the Startup.cs class again:

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(options =>

{

options.AddPolicy("CorsPolicy",

builder => builder.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

});

services.AddSignalR();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseCors("CorsPolicy");

app.UseSignalR(routes =>

{

routes.MapHub<ChartHub>("/chart");

});

app.UseMvc();

}

In the Configure method, we are adding SignalR to the IService collection. And in a ConfigureServices method, we are adding SignalR to the request pipeline by pointing to our ChartHub with the provided /chart path.

## Timer Implementation with DataManager and ChartController

To simulate a real-time data flow from the server, we are going to implement a Timer class from the System.Threading namespace. Let’s create a new folder TimerFeatures and inside it a new class TimerManager:

public class TimerManager

{

private Timer \_timer;

private AutoResetEvent \_autoResetEvent;

private Action \_action;

public DateTime TimerStarted { get; }

public TimerManager(Action action)

{

\_action = action;

\_autoResetEvent = new AutoResetEvent(false);

\_timer = new Timer(Execute, \_autoResetEvent, 1000, 2000);

TimerStarted = DateTime.Now;

}

public void Execute(object stateInfo)

{

\_action();

if((DateTime.Now - TimerStarted).Seconds > 60)

{

\_timer.Dispose();

}

}

}

We are using an Action delegate to execute passed callback function every two seconds. The timer will make a one-second pause before first execution. Finally, we just create a sixty seconds time slot for execution, to avoid limitless timer loop.

It is very important to have a method which has one object parameter and returns a void result. The Timer class expects that kind of method in its constructor.

After the TimeManager implementation, let's create a new folder DataStorage and inside it a new class DataManager. We are going to use this class to fake our data:

public static class DataManager

{

public static List<ChartModel> GetData()

{

var r = new Random();

return new List<ChartModel>()

{

new ChartModel { Data = new List<int> { r.Next(1, 40) }, Label = "Data1" },

new ChartModel { Data = new List<int> { r.Next(1, 40) }, Label = "Data2" },

new ChartModel { Data = new List<int> { r.Next(1, 40) }, Label = "Data3" },

new ChartModel { Data = new List<int> { r.Next(1, 40) }, Label = "Data4" }

};

}

}

Finally to complete this section, we are going to create a new controller file ChartController inside the Controllers folder:

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

[ApiController]

public class ChartController : ControllerBase

{

private IHubContext<ChartHub> \_hub;

public ChartController(IHubContext<ChartHub> hub)

{

\_hub = hub;

}

public IActionResult Get()

{

var timerManager = new TimerManager(() => \_hub.Clients.All.SendAsync("transferchartdata", DataManager.GetData()));

return Ok(new { Message = "Request Completed" });

}

}

In this controller class, we are using the IHubContext interface to create its instance via dependency injection. By using that instance object, we are able to access and call the hub methods. This is the reason why we don’t have any method in our ChartHub class. We don’t need any yet, because we are providing just one-way communication (server is sending data to the client only), and we can access all the hub methods with IHubContext interface.

Furthermore, in the Get action, we are instantiating the TimerManager class and providing a callback function as a parameter. This callback function will be executed every two seconds.

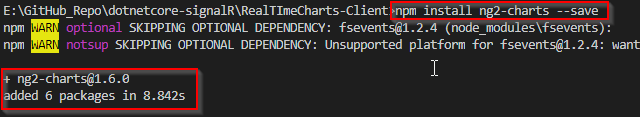
Now, we have to pay attention to the \_hub.Clients.All.SendAsync("transferchartdata", DataManager.GetData()) expression. With it, we are sending generated data to all subscribed clients on the „transferchartdata“ topic. This means that every client if it has a listener on the „transferchartdata“ topic, will receive a data generated by the DataManager class. And that is exactly what we are going to do in the next section.

## Angular Chart and SignalR Listener

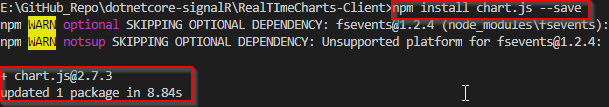
We have currently finished our work on the server side, so let’s switch to the client side.

To use charts in Angular, we are going to install two required libraries:

npm install ng2-charts --save



npm install chart.js --save



As soon as we install required libraries, we are going to modify the angular.json file:

"scripts": [

"./node\_modules/chart.js/dist/Chart.js"

]

And finally, let’s modify the app.module.ts file:

import { ChartsModule } from 'ng2-charts';

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

imports: [

BrowserModule,

ChartsModule,

HttpClientModule

],

Of course, the HttpClientModule is not required for the charts to work, but we are going to send the http request towards our Get action, so we need it.