**Exercise: SignalR**

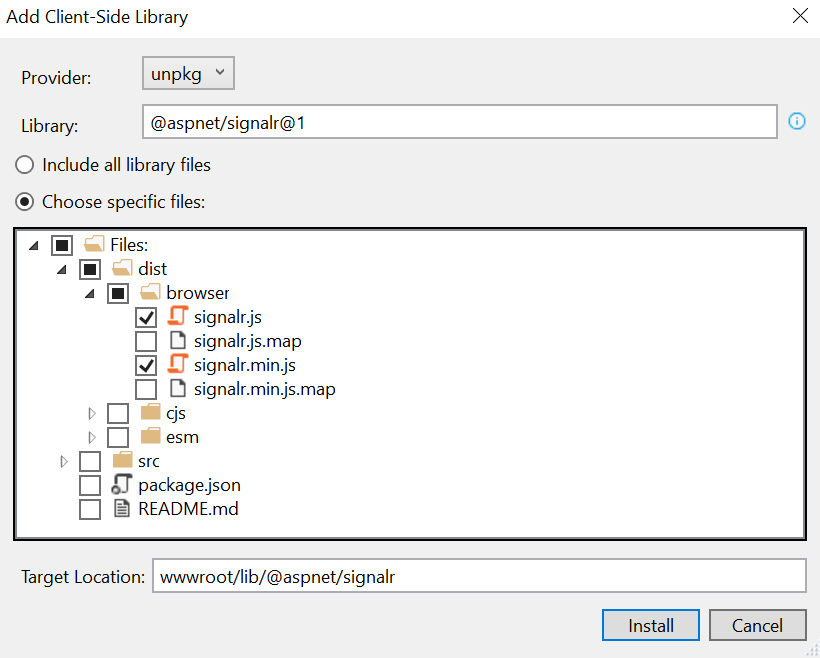
Problems for exercises and homework for the [“ASP.NET Core” course @ SoftUni](https://softuni.bg/trainings/2354/aspnet-core-july-2019).

## Chat

Let’s create a simple **Chat** application in **ASP.NET Core**. The app will let users type a username and send a message.

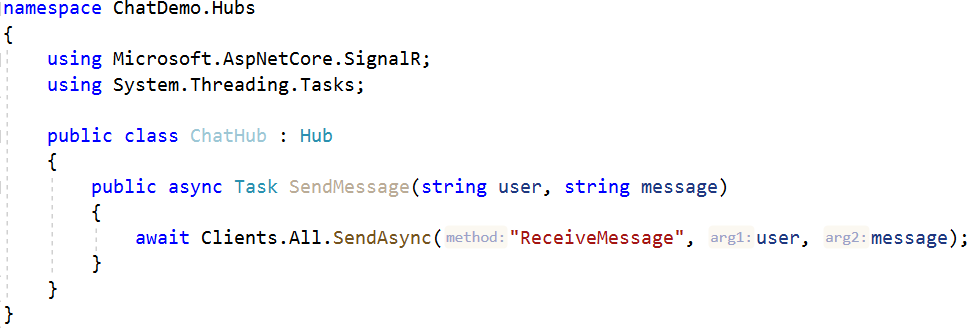
## Create Project and include SignalR

Create new Solution and add new **ASP.NET Core Web Application**. In **Solution Explorer**, right-click on the project, **Add -> Client-Side Library**.



## Create Hub class

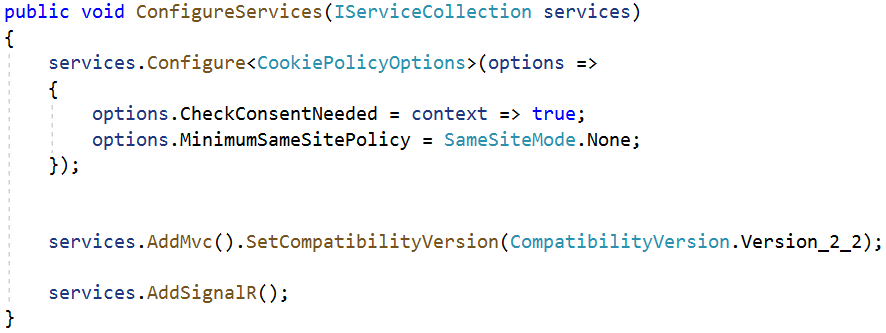
Create a **Hubs** folder in the project and create a **ChatHub.cs** class file.



The **ChatHub** class inherits from the **SignalR Hub** class. The **Hub** class manages connections, groups and messaging. The **SendMessage** method sends the received message to all clients and can be called by any connected client.

## Configuring SignalR

Configuring **SignalR** in our application is pretty simple. Go to **Startup.cs**:



And in **Configure()** method we configure the endpoint:



## View

Not it’s time for the client-side. Let’s replace the content of **Index.cshtml**.



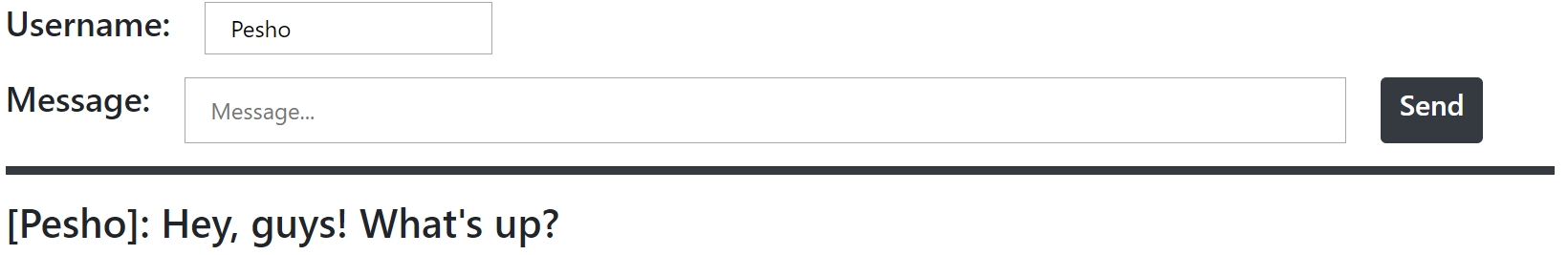
## JavaScript code

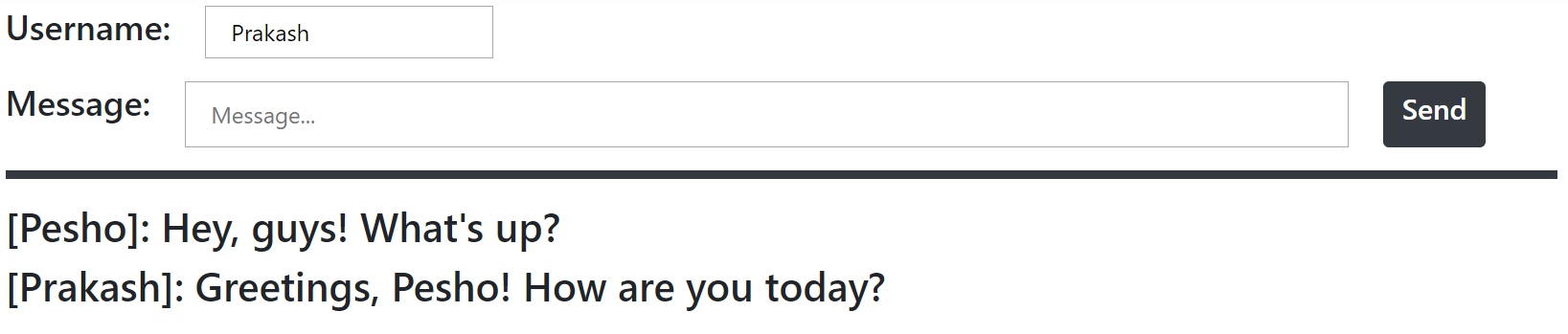
Create a **chat.js** file in the **wwwroot/js** folder. First we initialize a **SignalR connection**. When the back-end calls the method for **receiving a message**, a function will run for **showing** the sent content on our page. We also have an **error handler** for catching errors if something is wrong during the connection. Last, we have a function called when we click on **Send** button. The client-side calls the back-end method for **sending a message** and takes the **input values.**



## Test the Application

Now we are ready to test our application. Let’s **run the app** and open **2 browser instances** on the **URL** of the app. Enter a **Username** and a **Message** on one of the browsers and send it. Check the other one.



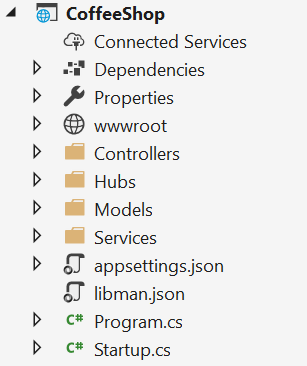


## Coffee

Let’s create a **Coffee Shop** for ordering coffee and displaying the process real-time for its delivery.

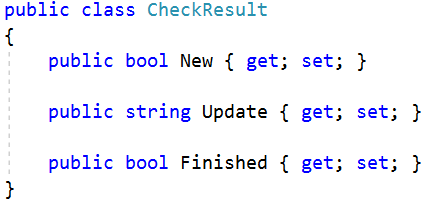
## Project Architecture

Following the first step from the previous exercise, **create new project** (e.g. **CoffeeShop**) and include **SignalR**. Then create folders for our **Controllers**, **Hubs**, **Models** and **Services**. The project should look like this:



## Models

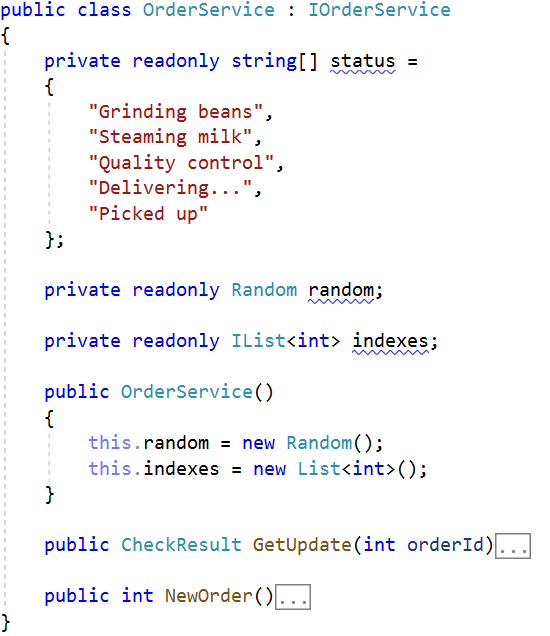
Create class **Order.cs** holding **Product** and **Size** as **strings**. Create another class called **CheckResult.cs** which will hold properties for checking **the state** of the made **order**.



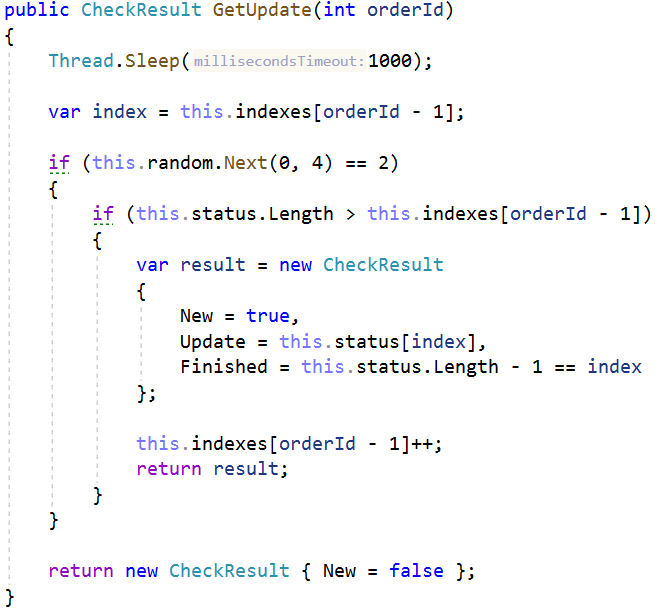
## Services

We will have only one **Service** for the **Orders**. Let’s create **an interface** holding two methods – **GetUpdate()** by **order’s id** which returns **CheckResult**, and **NewOrder()** which returns **integer**.

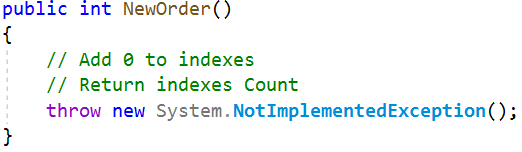
Now it’s time to implement **OrderService.cs**. Here we will have an **array of strings** holding the steps of making the ordered coffee and delivering it. We will also have a **Random** object for mimic the process duration of each step and **List** of **indexes** which will help us to access the current status of the **Order**.



In **GetUpdate()** method we set some milliseconds for **pausing** just to mimic the process of ordering. We create an **index**, taking the **order id** and subtracting it with one. We take **random number** **in the range of the array elements** and when it’s equal to 2 we check if **the length of the status array** is more than the **index**. If it is, we create a **new CheckResult**. We set **New** to **true**, we take **new message** from our **status array** using the index as our **Update**, and we set **Finished** to be **true** if the **length of the status messages** and our **index** are **equal**. Next we increment the index on **orderId-1** and **return the result**. It this method is our logic for changing the status of the order.



In **NewOrder()** method we simply **add 0** and **return indexes** **Count**.



## Hubs

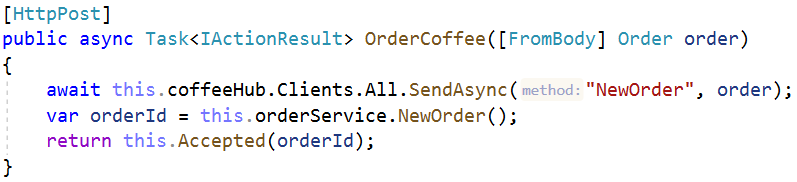
We will also have one hub – **CoffeeHub.cs**. Inside we will have one method – **GetUpdateForOrder()**, which will be called in our client-side. Inside while the order hasn’t finished we use Service’s **GetUpdate()** and if the result is **New** we send **ReceiveOrderUpdate** method. When the order is finished we send to the client-side method **Finished.**



## Controllers

In **CoffeeController** we will need the **Order Service** and the **Coffee Hub**.

The **OrderCoffee()** method is the **Post** method called when we submit an order. It takes the **input parameters**, send them to **the server-side**, creates **New Order** and returns status code **Accepted**.



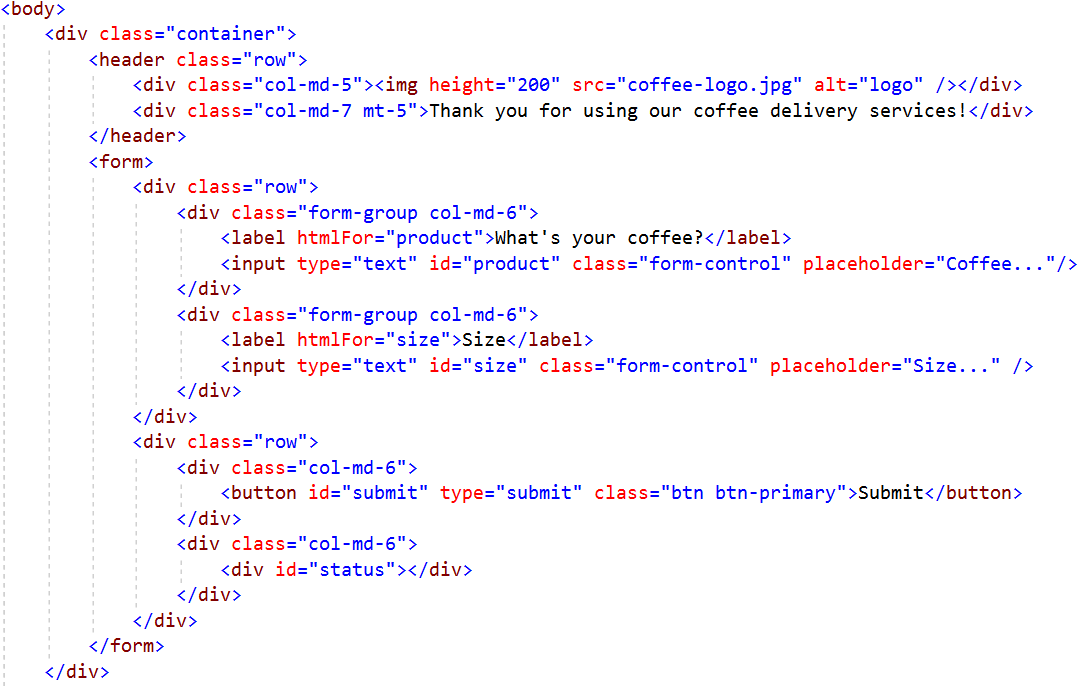
## Configuring SignalR

Before configuring **StartUp.cs**, let’s install the NuGet package **Microsoft.AspNetCore.SignalR.Protocols.MessagePack**. It will enable for us the **MessagePack** protocol for **SignalR**.



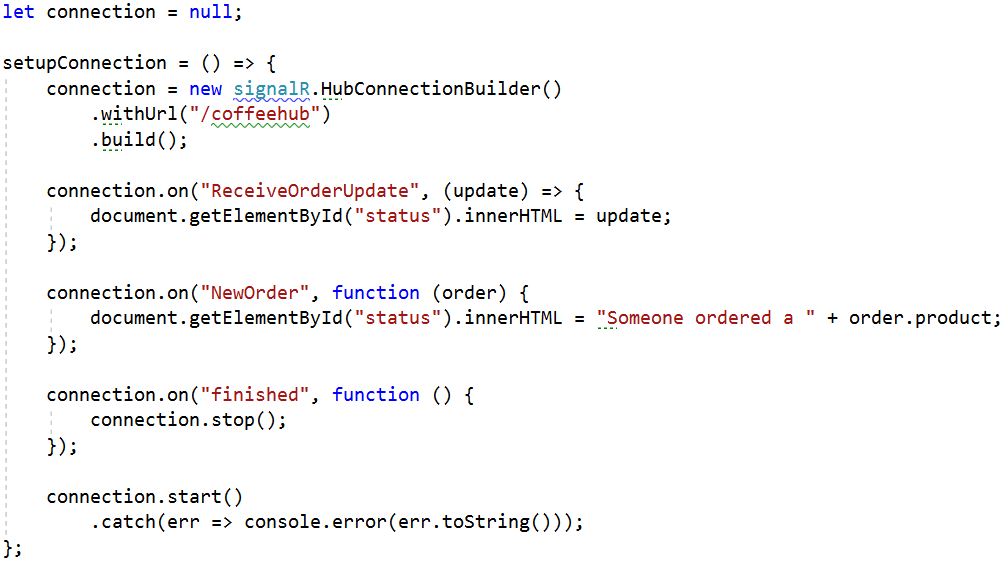
## View

Create an **HTML** file called **index** in **wwwroot** folder. Import **bootstrap** stylesheet and **signalr** script. Also you can import **coffee.js** file from **wwwroot/js** which we will implement in the next step. The body should looks something like this:

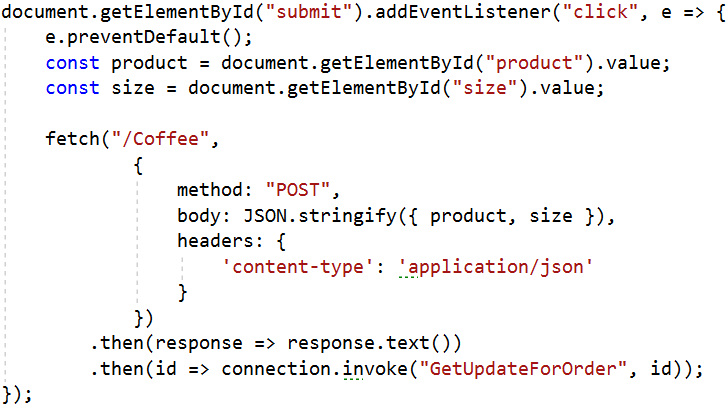


## JavaScript Code

In **coffee.js** we create a **connection**. By received methods from the back-end we change our **HTML**. If we have **ReceiveOrderUpdate** it means that our order is **updated** and we can **display the status of the order**. If we have a **new order** we can display **what is ordered**. And if the order is **finished** we can **stop the connection**.



Next we have to **listen** our **submit button** and when it’s clicked to take the input values for **product** and **size**, and send them to our **post method** as **JSON object**. Then we invoke the **GetUpdateForOrder** method.



## Test the Application

Now we are ready to test our application. Let’s **run the app** and enter a **Coffee** and a **Size**. By clicking **Submit** button we can see the whole process for delivering our ordered coffee. 😊

