**Exercise: SignalR**

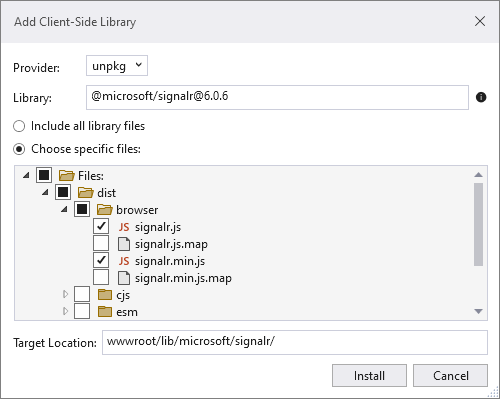
Problems for exercises and homework for the [“ASP.NET Core” course @ SoftUni](https://softuni.bg/trainings/3601/asp-dot-net-core-february-2022).

## Chat

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

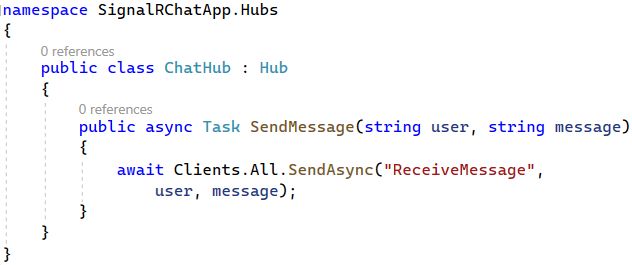
### Step 1: Create Project and include SignalR

Create a new Solution and add a new **ASP.NET Core Web App (MVC)**. In **Solution Explorer**, right-click on the project, **Add → Client-Side Library**.



### Step 2: Create Hub class

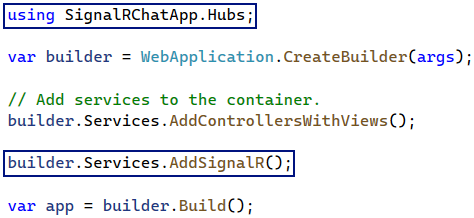
Create a **Hubs** folder in the project and create a **ChatHub.cs** class file.



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.

### Step 3: Configuring SignalR

Configuring **SignalR** in our application is pretty simple. Go to **Program.cs** and add:



Then, we configure the routing systems:



### Step 4: View

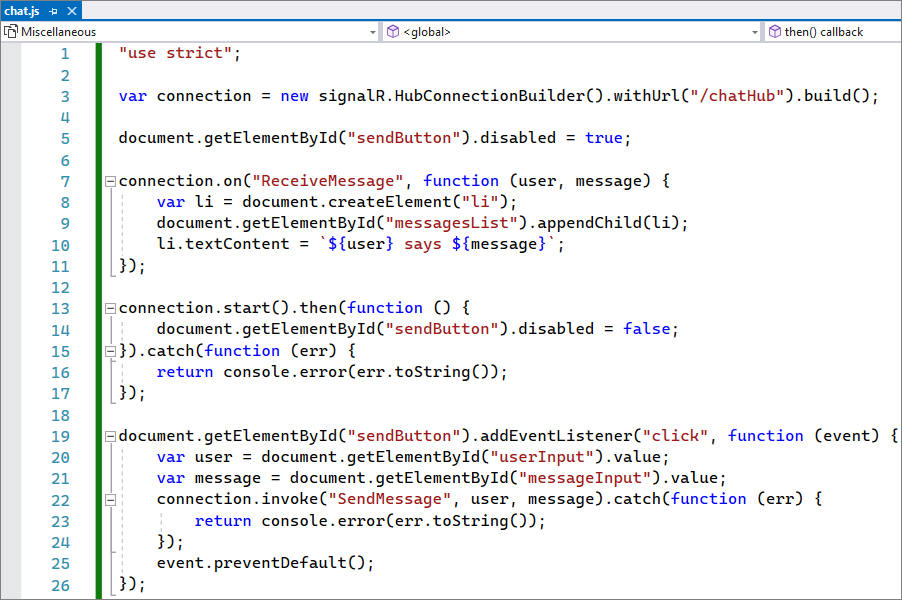
Now it's time for the client-side. Let's replace the content of **Index.cshtml**.

You can copy the code from here as it is a lot to write:

|  |
| --- |
| @{  ViewData["Title"] = "Home Page";  }  <div class="container">  <div class="row">&nbsp;</div>  <div class="row">  <div class="col-2">User</div>  <div class="col-4"><input type="text" id="userInput" /></div>  </div>  <div class="row">  <div class="col-2">Message</div>  <div class="col-4"><input type="text" id="messageInput" /></div>  </div>  <div class="row">&nbsp;</div>  <div class="row">  <div class="col-6">  <input type="button" id="sendButton" value="Send Message" />  </div>  </div>  </div>  <div class="row">  <div class="col-12">  <hr />  </div>  </div>  <div class="row">  <div class="col-6">  <ul id="messagesList"></ul>  </div>  </div>  <**script** src="~/js/signalr/dist/browser/signalr.js"></**script**>  <**script** src="~/js/chat.js"></**script**> |

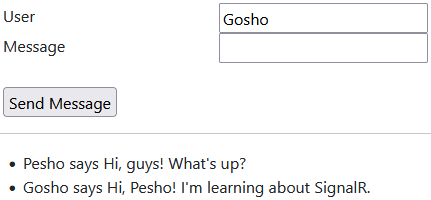
### Step 5: JavaScript code

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 the **[Send]** button. The client-side calls the back-end method for **sending a message** and takes the **input values.**



### Step 6: 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.

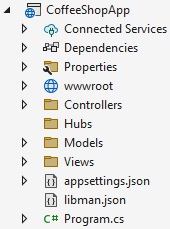


## Coffee

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

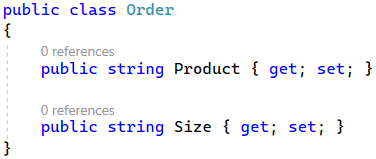
### Step 1: Project Architecture

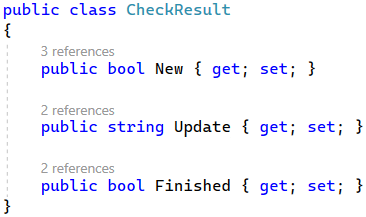
Following the first step from the previous exercise, **create a new project** (e.g. **CoffeeShop**) and include **SignalR**. Then create folders for our **Controllers**, **Hubs**, **Models** and **Services**. The project should look like this:



### Step 2: 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**.

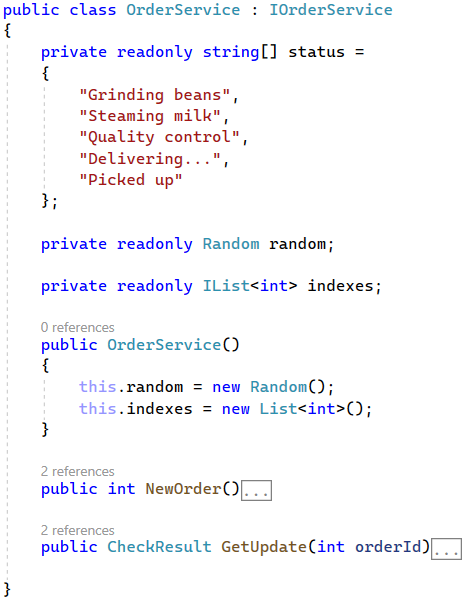




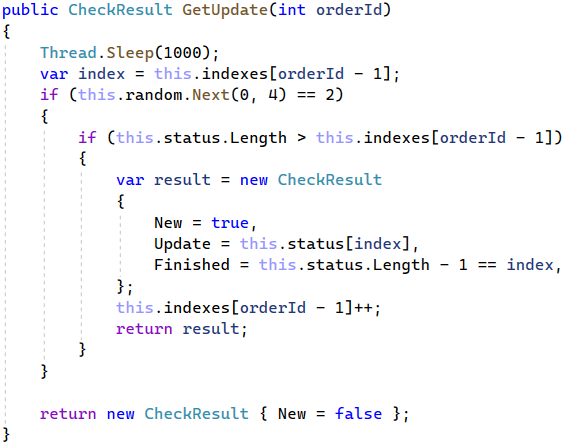
### Step 3: Services

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 an **integer**.

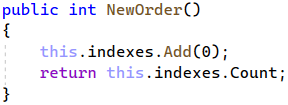
Now it's time to implement **OrderService.cs**. Here we will have an **array of strings** holding the steps of making the ordered coffee and delivering it. We will also have a **Random** object to mimic the process duration of each step and a **List** of **indexes** which will help us to access the current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

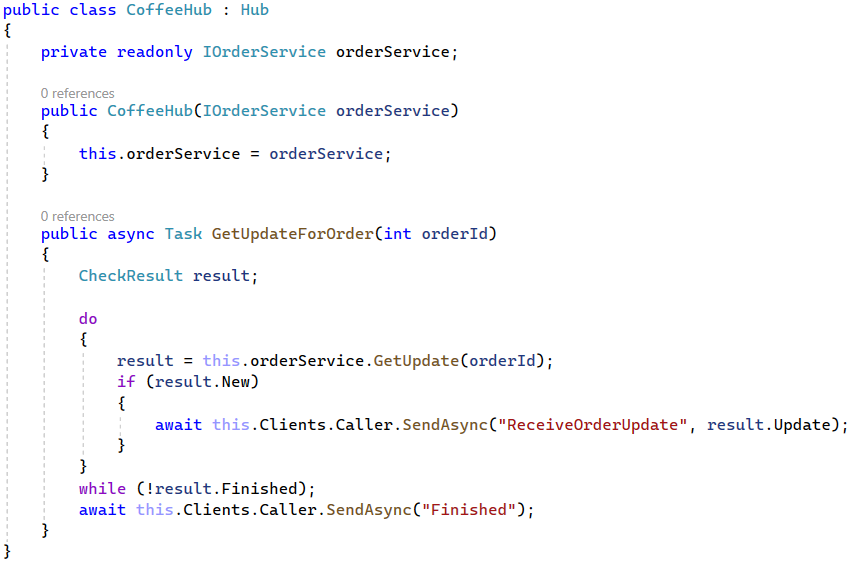


In **NewOrder()** method we simply **add 0** and **return indexes** **Count**.



### Step 4: Hubs

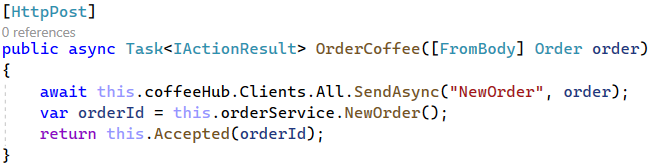
We will also have one hub – **CoffeeHub.cs**. Inside we will have one method – **GetUpdateForOrder()**, which will be called on our client-side. Inside while the order hasn’t finished we use Service’s **GetUpdate()** and if the result is **New** we send **ReceiveOrderUpdate** method. When the order is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



### Step 5: 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**, sends them to **the server-side**, creates **New Order,** and returns the status code **Accepted**.



### Step 6: Configuring SignalR

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



Don't forget to configure the routing systems.

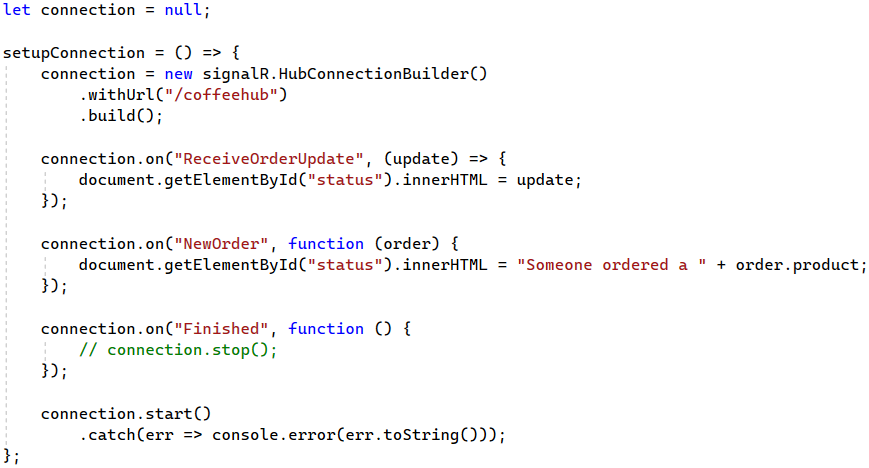
### Step 7: View

The view should look something like this:

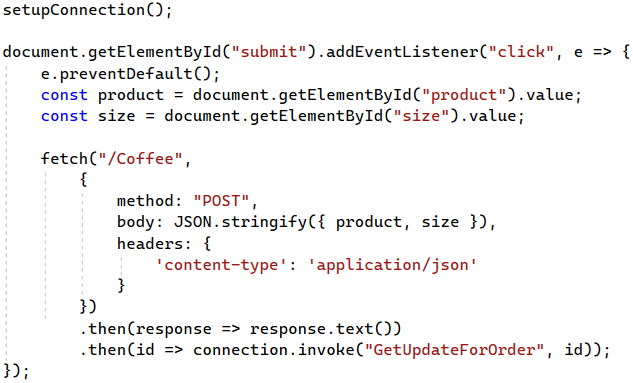
|  |
| --- |
| @{  ViewData["Title"] = "Home Page";  }  <body>  <div class="containter">  <header class="row">  <div class="row">  <div class="text-center" ">  <img height="200" max-width="20%" src="https://static.vecteezy.com/system/resources/previews/000/565/744/large\_2x/vector-coffee-cup-logo-template.jpg"  alt="logo" />  </div>  </div>  <div class="row text-center">  <h3>  Thank you for using our coffee delivery services!  </h3>  </div>  </header>  <**form**>  <div class="row">  <div class="form-group col-md-6">  <label for="product">What's your coffee?</label>  <input type="text" id="product" class="form-control"  placeholder="Coffee..." />  </div>  <div class="form-group col-md-6">  <label for="size">Size</label>  <input type="text" id="size" class="form-control"  placeholder="Size..." />  </div>  </div>  <div class="row">  <div class="col-md-6">  <button id="submit" type="submit" class="btn btn-primary">Submit</button>  </div>  <div class="col-md-6">  <div id="status"></div>  </div>  </div>  </**form**>  </div>  </body>  <**script** src="~/lib/microsoft/signalr/dist/browser/signalr.js"></**script**>  <**script** src="~/js/coffee.js"></**script**> |

### Step 8: JavaScript Code

In **coffee.js** we create a **connection**. By receiving 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 to** 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 objects**. Then we invoke the **GetUpdateForOrder** method.



### Step 9: 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.