

# Introduction to WebSockets with SignalR in .NET Part 3

MAY 22, 2014    [LEAVE A COMMENT \(HTTPS://DOTNETCODR.COM/2014/05/22/INTRODUCTION-TO-WEB\\_SOCKETS-WITH-SIGNALR-IN-NET-PART-3/#RESPOND\)](https://dotnetcodr.com/2014/05/22/introduction-to-websockets-with-signalr-in-net-part-3/#RESPOND)

## Introduction

We'll continue our discussion of SignalR where we left off in the [previous post \(https://dotnetcodr.com/2014/05/19/introduction-to-websockets-with-signalr-in-net-part-2-code-basics/\)](https://dotnetcodr.com/2014/05/19/introduction-to-websockets-with-signalr-in-net-part-2-code-basics/). So open the SignalRWeb demo project and let's get to it!

## Demo continued

We left off having the following "scripts" section in Index.cshtml:

```
1 | @section scripts
2 | {
3 |     <script src="~/Scripts/jquery.signalR-2.0.3.js"></script>
4 |     <script src="~/Scripts/knockout-3.1.0.js"></script>
5 |     <script src="~/Scripts/results.js"></script>
6 | }
```

We'll build on this example to see how a client can interact with the server through WebSockets. At present we have an empty Hello() method in our ResultsHub class. Insert another method which will allow the clients to send a message to the server:

```
1 | public void SendMessage(String message)
2 | {
3 |
4 | }
```

There's a property called Context that comes with SignalR. It is similar to HttpContext in ASP.NET: it contains a lot of information about the current connection: headers, query string, authentication etc. Just type "Context." in the method body and inspect the available properties, they should be self-explanatory. We have no authentication so we'll use the connection ID property of Context to give the sender some identifier:

```
1 | public void SendMessage(String message)
2 | {
3 |     string completeMessage = string.Concat(Context.ConnectionId
4 |         , " has registered the following message: ", message);
5 |
6 |     Clients.All.registerMessage(completeMessage);
7 | }
```

You'll recall from the previous post that we need to deal with JavaScript in SignalR. The above piece of code will result in a JavaScript method called "registerMessage" to be invoked from the server. Where is that method? We need to write it of course. We inserted a JavaScript file called "results.js" previously. Open that file and enter the following stub:

```
1 | (function () {
2 |     var resultsHub = $.connection.resultsHub;
3 | }());
```

resultsHub is a reference to the hub we've been working on. The "connection" property comes from the SignalR jQuery library and creates a SignalR connection. Through the connection property you'll be able to reference your hubs. There's no IntelliSense for the hub names, so be careful with the spelling.

We need to stop for a second and go back to Index.cshtml. There's one more JavaScript source we need to reference, but it's not available in the Scripts folder. Recall that we used the MapSignalR OWIN extension in Startup.cs. That extension will map the SignalR hubs to the /signalr endpoint. Add the following script declaration to the scripts section below the jquery.signalR-2.x.x.js script reference:

```
1 | <script src="~/SignalR/hubs"></script>
```

Start the application and navigate to <http://localhost:xxxxx/SignalR/hubs> (<http://localhost:xxxxx/SignalR/hubs>). You should see some JavaScript related to SignalR in the browser, so the script reference is valid. Basically this code generates proxies for the hubs. If you scroll down you'll see that it has found the ResultsHub and its two dynamic methods:

```
1 | proxies.resultsHub = this.createHubProxy('resultsHub');
2 | proxies.resultsHub.client = { };
3 | proxies.resultsHub.server = {
4 |     hello: function () {
5 |         return proxies.resultsHub.invoke.apply(proxies.resultsHub, $.merge(["Hello"], $.makeArray(arguments)));
6 |     },
7 |
8 |     sendMessage: function (message) {
9 |         return proxies.resultsHub.invoke.apply(proxies.resultsHub, $.merge(["SendMessage"], $.makeArray(arguments)));
10 |     }
11 | };

```

As you add other hubs and other dynamic methods they will be registered here in this script generated by SignalR using Reflection. Note that the hello and sendMessage functions have been registered on the server – proxies.resultsHub.server – which is expected.

Add the following code to results.js just below the resultsHub reference:

```
1 | $.connection.hub.logging = true;
2 | $.connection.hub.start();
```

We turn on logging so that we can see what SignalR is doing behind the scenes. Then we tell SignalR to start the communication. This method will go through the [4 ways of establishing a connection with the client](https://dotnetcodr.com/2014/05/15/introduction-to-websockets-with-signalr-in-net-part-1-the-basics/) (<https://dotnetcodr.com/2014/05/15/introduction-to-websockets-with-signalr-in-net-part-1-the-basics/>) and determine which one works best.

Next we need the JavaScript methods that will be invoked: hello and registerMessage. Add the following stubs just below the call to hub.start():

```
1 | resultsHub.client.hello = function(){
2 |
3 | }
4 |
5 | resultsHub.client.registerMessage = function (message) {
6 |
7 | };

```

We'll concentrate on the registerMessage function, hello can remain empty. Next we need to show the responses on the screen. As mentioned before we'll use [knockout.js](http://knockoutjs.com/) (<http://knockoutjs.com/>) as it provides for a very responsive GUI. I in fact only know the basics of knockout.js, as client side programming is not really my cup of tea. If you don't know anything about knockout.js then you might want to go through the first couple of tutorials [here](http://learn.knockoutjs.com/#/?tutorial=intro) (<http://learn.knockoutjs.com/#/?tutorial=intro>). It is very much based on view-models which are bound to HTML elements. As the properties change so do the values of those elements in a seamless fashion. We won't go into any detail about the knockout specific details here. Add the following code below the resultsHub.client.registerMessage stub:

```
1 | var messageModel = function () {
2 |     this.registeredMessage = ko.observable("");
3 |     this.registeredMessageList = ko.observableArray()
4 | };

```

The registeredMessage property will show a message sent by the client. registeredMessageList will hold all messages that have been sent from the server. Next add a model prototype:

```

1 | messageModel.prototype = {
2 |
3 |     newMessage: function () {
4 |         resultsHub.server.sendMessage(this.registeredMessage());
5 |         this.registeredMessage("");
6 |     },
7 |     addMessageToList: function (message) {
8 |         this.registeredMessageList.push(message);
9 |     }
10 |
11 | };

```

newMessage is meant to be a function that can be invoked upon a button click. It sends the message to the server and then clears it. Recall that we called our function in ResultsHub “SendMessage”. You can call it using the “server” property followed by the name of the function you’d like to invoke. addMessageToList does exactly what the function name implies. Then we create an instance of the view-model and instruct knockout to start binding it to our GUI elements:

```

1 | var viewModel = new messageModel();
2 | $(function () {
3 |     ko.applyBindings(viewModel);
4 | });

```

We can now fill in the resultsHub.client.registerMessage stub:

```

1 | resultsHub.client.registerMessage = function (message) {
2 |     viewModel.addMessageToList(message);
3 | };

```

It simply calls upon the view-model instance to add the new message to the message list.

Now we can create the corresponding HTML code on index.cshtml. Add the following right above the scripts section.

```

1 | <div>
2 |     <input type="text" placeholder="Your message..." data-bind="value:registeredMessage" />
3 |     <button data-bind="click:newMessage">Register message</button>
4 | </div>

```

This bit of markup will serve as the “register” section where the user enters a message in the text box and sends it to the server using the button. Note the knockout-related data-bind attributes. The text box is bound to the registeredMessage property and the click event of the button is bound to the newMessage function of the model, both defined in our results.js file.

Add the following HTML below the “register” section:

```

1 | <div>
2 |     <div data-bind="foreach:registeredMessageList">
3 |         <div data-bind="text: $data"></div>
4 |     </div>
5 | </div>

```

This is a knockout foreach expression: for each message in the registeredMessageList array we print the message in a separate div. We bind the text property of the div to the message. The current value in the foreach loop can be accessed using “\$data” in knockout.

Let’s go through the expected process step by step:

1. The user enters a message in the text box and presses the register message button
2. The button invokes the newMessage function of the knockout view-model ‘messageModel’
3. The newMessage function will invoke the SendMessage(string input) method of the ResultsHub.cs Hub
4. The SendMessage function constructs some return message and calls the registerMessage JavaScript function on each client, in our case defined in results.js
5. The registerMessage function receives the string returned by the SendMessage function of ResultsHub
6. registerMessage adds the message to the messages array of the view-model
7. The view-model is updated and knockout magically updates the screen with the new list of messages

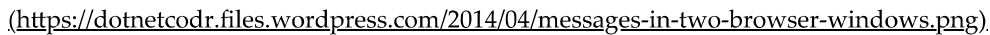
```
[22:25:39 GMT+0200 (W. Europe Summer Time)] SignalR: Negotiating with /signalr/negotiate?connectionData=A5BN7fmgNameK2ASAKv2&SignalRResultTokenK2NKT05S0dClientProtocol=1.3'. {source: signalR-2.6.3.js:76}
[22:25:39 GMT+0200 (W. Europe Summer Time)] SignalR: Connecting to websocket endpoint 'ws://localhost:51313/signalr/connect?transport=websocket&connectionToken=OKX7UabPZIoQJc5tYzHjCgldyhmVwGpA7v3VtINzF3Qn17J9D9mMPrL0MKHGBe7WkNZ4lFA9SP5PbI11CxvATVPqOeHatHgmU284.d29SOePdqbvSAlE7fJdc&connectionData=A5BN7fmgNameK2ASAKv2&SignalRResultTokenK2NKT05S0dIo=2'. {source: signalR-2.6.3.js:76}
[22:25:40 GMT+0200 (W. Europe Summer Time)] SignalR: Websocket opened. {source: signalR-2.6.3.js:76}
[22:25:40 GMT+0200 (W. Europe Summer Time)] SignalR: Now monitoring keep alive with a warning timeout of 13333.333333333333 and a connection loss timeout of 20000. {source: signalR-2.6.3.js:76}
```

It has found the ResultsHub and managed to open the web socket endpoint. Note the protocol of 'ws'. Now insert a message in the text box and press register. Code execution should stop at the breakpoint in Visual Studio meaning that we've managed to wire up the components correctly. I encourage you to inspect the Context property and see what's available in it. You should then see your message under the textbox as returned by the SendMessage:

(<https://dotnetcodr.files.wordpress.com/2014/04/message-collection-by-signalr.png>).

- SignalR: Invoking `resultshub.SendMessage`
- SignalR: Triggering client hub event 'registerMessage' on hub 'ResultsHub'.
- SignalR: webSockets reconnecting.

If you want to talk to yourself on two different screens then open up another browser window, Firefox or IE, and navigate to the same localhost address as in Chrome. Then start sending messages. You should see them in both windows:



Read the next post in this series [here](https://dotnetcodr.com/2014/05/26/introduction-to-websockets-with-signalr-in-net-part-4-stock-price-ticker/) (<https://dotnetcodr.com/2014/05/26/introduction-to-websockets-with-signalr-in-net-part-4-stock-price-ticker/>).

View the list of posts on Messaging [here](https://dotnetcodr.com/messaging/) (https://dotnetcodr.com/messaging/).


FILED UNDER [.NET](#), [MESSAGING](#) TAGGED WITH [C#](#), [SIGNALR](#), [WEBSOCKETS](#)

### About Andras Nemes

I'm a .NET/Java developer living and working in Stockholm, Sweden.

Create a free website or blog at WordPress.com.

#### Advertisements



Aspose

## Java XPS and PostScript API

REPORT THIS AD