Introduction to WebSockets with SignalR in .NET Part 4: stock price ticker

MAY 26, 2014 2 COMMENTS (HTTPS://DOTNETCODR.COM/2014/05/26/INTRODUCTION-TO-WEBSOCKETS-WITH-SIGNALR-IN-NET-PART-4-STOCK-PRICE-TICKER/#COMMENTS)

Introduction

In the last post (https://dotnetcodr.com/2014/05/22/introduction-to-websockets-with-signalr-in-net-part-3/) we saw the first basic, yet complete example of SignalR in action. We demonstrated the core functionality of a messaging application where messages are shown on screen in real time.

We're now ready to implement the demo that I promised before: update a stock price in real time and propagate the changes to all listening browsers.

We'll build on the sample project we've been working on so far in this series, so have it open and let's get into it!

Demo: server side

Add a folder to the solution called Stocks. Add a new class called Stock:

```
1  public class Stock
2  {
3      public Stock(String name)
4      {
5          Name = name;
6      }
7
8      public string Name { get; set; }
9      public double CurrentPrice
10     {
11         get
12         {
13             Random random = new Random();
14             return random.NextDouble() * 10.0;
15         }
16     }
17 }
```

We need a stock name in the constructor. We'll simply randomize the stock price using Random.NextDouble(). In reality it will be some proper repository that does this but again, we'll keep the example simple.

Next add a new class called StockService which will wrap stock-related methods.

Note that such service classes should be hidden behind an interface and injected into the consuming class to avoid tight coupling. However, for this sample we'll ignore all such "noise" and concentrate on the subject matter. If you're not sure what I'm on about then read about dependency inversion here (https://dotnetcodr.com/2013/08/26/solid-design-principles-in-net-the-dependency-inversion-principle-and-the-dependency-injection-pattern/). The next post on SignalR will explore dependency injection.

StockService takes the following form:

```
public class StockService
{
    private List<Stock> _stocks;

    public StockService()
    {
        _stocks = new List<Stock>();
        _stocks.Add(new Stock("GreatCompany"));
        _stocks.Add(new Stock("NiceCompany"));
        _stocks.Add(new Stock("EvilCompany"));
    }

    public dynamic GetStockPrices()
    {
        return _stocks.Select(s => new { name = s.Name, price = s.CurrentPrice });
    }
}
```

We maintain a list of companies whose stocks are monitored. We return an anonymous class for each Stock in the list, hence the dynamic keyword.

We'll read these values from our ResultsHub hub. We'll need to constantly monitor the stock prices in a loop on a separate thread so that it doesn't block all other code. This calls for a Task from the Task Parallel Library. If you don't know what Tasks are then check out the first couple of posts on TPL here (https://dotnetcodr.com/task-parallel-library/), a basic knowledge will suffice. We'll need to start monitoring the prices as soon as someone opens the page in a browser. We can start the process directly from the hub constructor:

```
public ResultsHub()
{
    StartStockMonitoring();
}

private void StartStockMonitoring()
{

}
```

Here's the complete StartStockMonitoring method:

```
private void StartStockMonitoring()
{
    Task stockMonitoringTask = Task.Factory.StartNew(async () =>
        {
            StockService stockService = new StockService();
            while(true)
            {
                dynamic stockPriceCollection = stockService.GetStockPrices();
                Clients.All.newStockPrices(stockPriceCollection);
                await Task.Delay(5000);
            }
        }, TaskCreationOptions.LongRunning);
}
```

We start a task that instantiates a stock service and starts an infinite loop. We retrieve the stock prices and inform all clients through a newStockPrices JavaScript method and pass in the stock prices dynamic object. Then we pause the loop for 5 seconds using Task.Delay. Task.Delay returns an awaitable task so we need the await keyword. In order for that to take effect through we'll need to decorate the lambda expression with async. You've never heard of await and async? Start here (https://dotnetcodr.com/2012/12/31/await-and-async-in-net-4-5-with-c/). Finally, we notify the task scheduler in the task constructor that this will be a long running process.

Demo: client side

Let's extend our GUI and results.js to show the stock prices in real time. We'll use a similar approach as we had in case of the messaging demo: knockout.js with a view-model. Add the following code to results.js:

```
resultsHub.client.newStockPrices = function (stockPrices) {
     viewModel.addStockPrices(stockPrices);
}
```

…which will be the JS function that's called from ResultsHub.StartStockMonitoring. We'll complete the addStockPrices later, we need some building blocks first. We'll need a new custom JS object to store a single stock. Add the following constructor function to results.js:

```
var stock = function (stockName, price) {
        this.stockName = stockName;
        this.price = price;
};
```

Also, we need a new property in the messageModel constructor function to store the stock prices in an array:

```
var messageModel = function () {
        this.registeredMessage = ko.observable(""),
        this.registeredMessageList = ko.observableArray(),
        this.stockPrices = ko.observableArray()
    };
```

Add the following function property to messageModel.prototype:

```
addStockPrices: function (updatedStockPrices) {
            var self = this;

            $.each(updatedStockPrices, function (index, updatedStockPrice) {
                var stockEntry = new stock(updatedStockPrice.name, updatedStockPrice.price);
                self.stockPrices.push(stockEntry);
            });
}
```

We simply add each update to the observable stockPrices array. updatedStockPrice.name and updatedStockPrice.price come from our dynamic function in StockService:

```
public dynamic GetStockPrices()
{
    return _stocks.Select(s => new { name = s.Name, price = s.CurrentPrice });
}
```

We just need some extra HTML in Index.cshtml:

```
<div data-bind="foreach:stockPrices">
    <p><span data-bind="text:stockName"></span>: <span data-bind="text:price"></span></p>
</div>
```

The stockName and price properties in the text bindings come from the stock object in results.js.

This should be it. Start the application and you'll see the list of stocks and their most recent prices coming in from the server in a textual form.

View the next part of this series here (https://dotnetcodr.com/2014/05/29/introduction-to-websockets-with-signalr-in-net-part-5-dependency-injection-in-hub/).

View the list of posts on Messaging here (https://dotnetcodr.com/messaging/).
FILED UNDER .NET, MESSAGING TAGGED WITH C#, SIGNALR, WEB SOCKETS

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

2 Responses to *Introduction to WebSockets with SignalR in .NET Part 4: stock price ticker*

Glen Jeffrey says:
May 9, 2015 at 8:55 pm
Thanks your very much for such a detailed and step by step tutorial! I am pretty new to the world of websockets and your tutorial really helped me to get my first project up and running really quickly!

Reply

Andras Nemes says:
May 9, 2015 at 9:00 pm
Hi Glen, I'm glad you've liked! //Andras

Reply