Tutorial: Server broadcast with SignalR 2

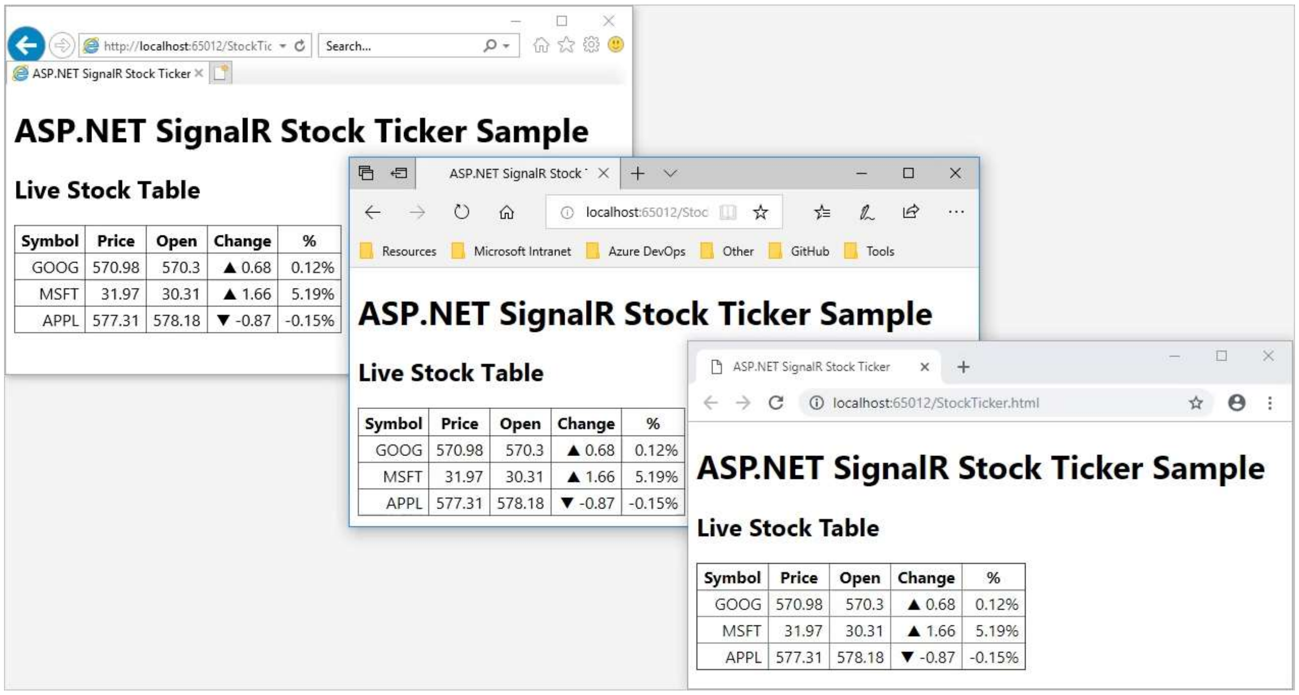
Article • 07/12/2022 • 26 minutes to read

⚠ **Warning**

This documentation isn't for the latest version of SignalR. Take a look at **ASP.NET Core SignalR**.

This tutorial shows how to create a web application that uses ASP.NET SignalR 2 to provide server broadcast functionality. Server broadcast means that the server starts the communications sent to clients.

The application that you'll create in this tutorial simulates a stock ticker, a typical scenario for server broadcast functionality. Periodically, the server randomly updates stock prices and broadcast the updates to all connected clients. In the browser, the numbers and symbols in the **Change** and % columns dynamically change in response to notifications from the server. If you open additional browsers to the same URL, they all show the same data and the same changes to the data simultaneously.



http://localhost:65012/StockTic
Search...
ASP.NET SignalR Stock Ticker
localhost:65012/Stoc
Resources
Microsoft Intranet
Azure DevOps
Other
GitHub
Tools
localhost:65012/StockTicker.html

ASP.NET SignalR Stock Ticker Sample

Live Stock Table

Symbol	Price	Open	Change	%
GOOG	570.98	570.3	▲ 0.68	0.12%
MSFT	31.97	30.31	▲ 1.66	5.19%
APPL	577.31	578.18	▼ -0.87	-0.15%

In this tutorial, you:

- ✓ Create the project
- ✓ Set up the server code
- ✓ Examine the server code

- ✓ Set up the client code
- ✓ Examine the client code
- ✓ Test the application
- ✓ Enable logging

> **Important**
>
> If you don't want to work through the steps of building the application, you can install the SignalR.Sample package in a new Empty ASP.NET Web Application project. If you install the NuGet package without performing the steps in this tutorial, you must follow the instructions in the *readme.txt* file. To run the package you need to add an OWIN startup class which calls the `ConfigureSignalR` method in the installed package. You will receive an error if you do not add the OWIN startup class. See the **Install the StockTicker sample** section of this article.

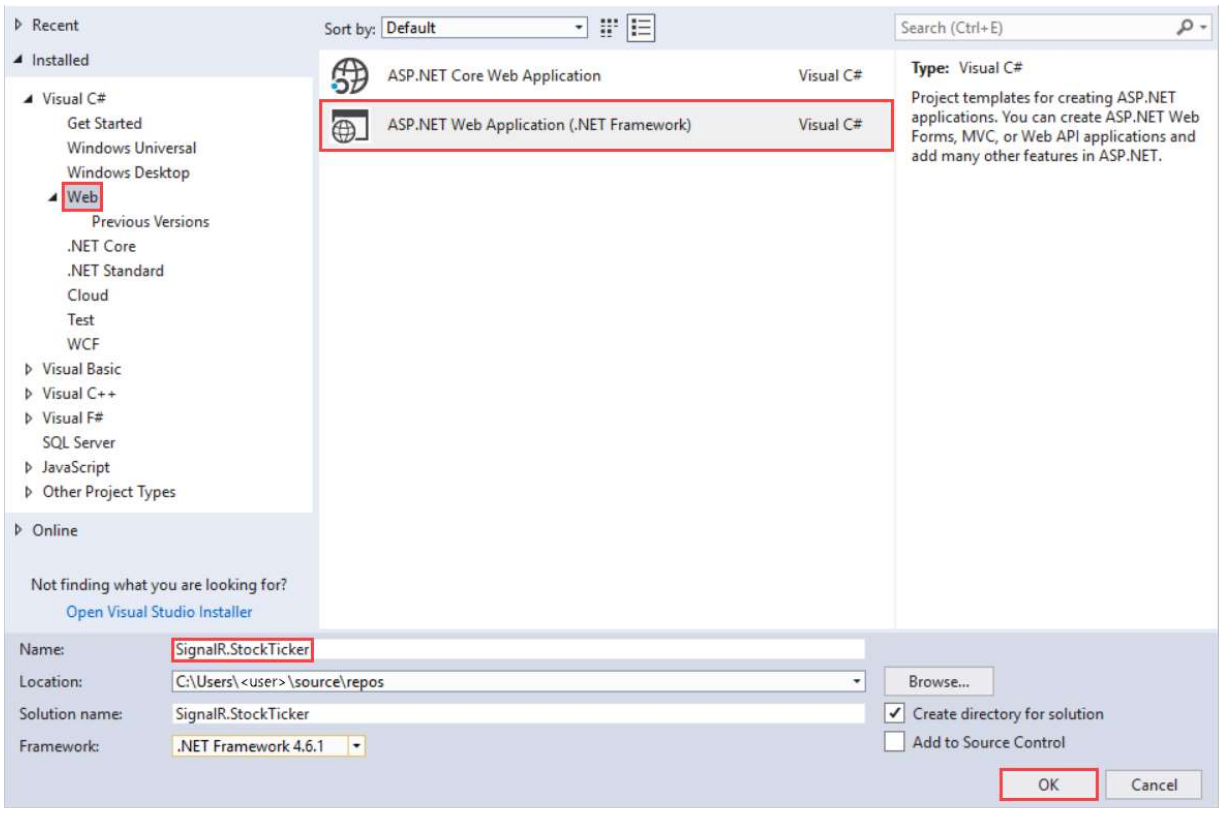
Prerequisites

- Visual Studio 2017 with the **ASP.NET and web development** workload.

Create the project

This section shows how to use Visual Studio 2017 to create an empty ASP.NET Web Application.

1. In Visual Studio, create an ASP.NET Web Application.



Type: Visual C#

Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Name: SignalR.StockTicker
Location: C:\Users\<user>\source\repos
Solution name: SignalR.StockTicker
Framework: .NET Framework 4.6.1
✓ Create directory for solution
☐ Add to Source Control

2. In the **New ASP.NET Web Application - SignalR.StockTicker** window, leave **Empty** selected and select **OK**.

Set up the server code

In this section, you set up the code that runs on the server.

Create the Stock class

You begin by creating the *Stock* model class that you'll use to store and transmit information about a stock.

1. In **Solution Explorer**, right-click the project and select **Add** > **Class**.

2. Name the class *Stock* and add it to the project.

3. Replace the code in the *Stock.cs* file with this code:

C#

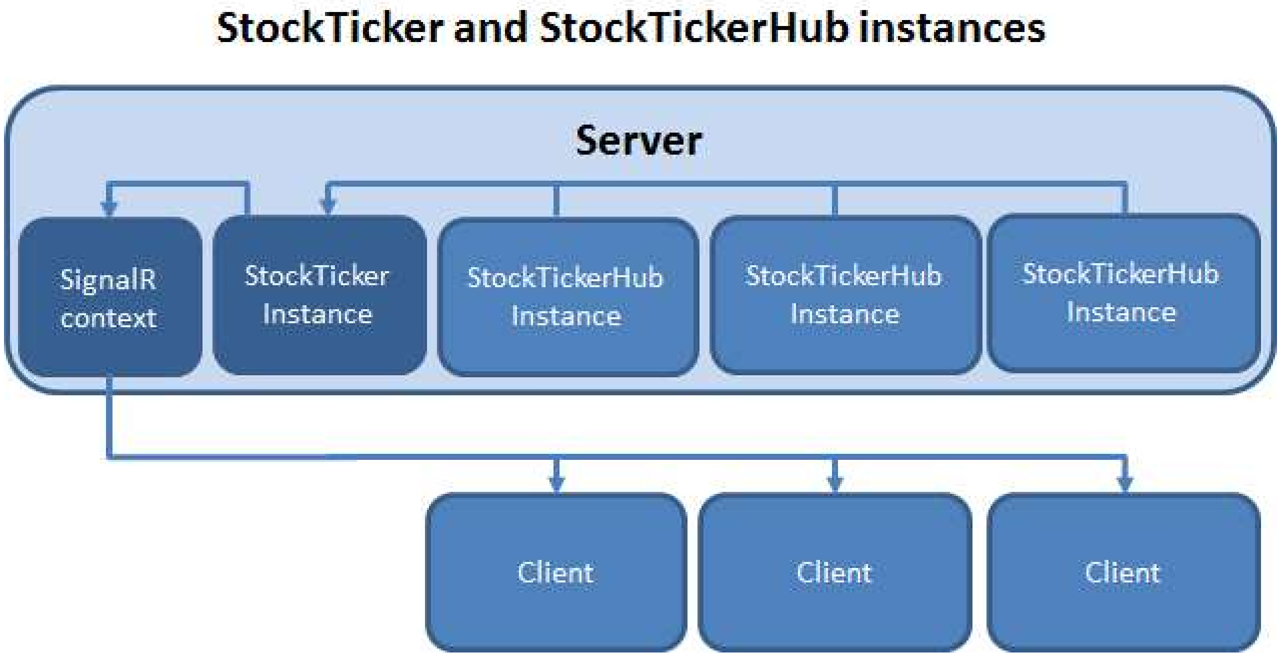
```
using System;

namespace SignalR.StockTicker
```

```
{
    public class Stock
    {
        private decimal _price;

        public string Symbol { get; set; }

        public decimal Price
        {
            get
            {
                return _price;
            }
            set
            {
                if (_price == value)
                {
                    return;
                }

                _price = value;

                if (DayOpen == 0)
                {
                    DayOpen = _price;
                }
            }
        }

        public decimal DayOpen { get; private set; }

        public decimal Change
        {
            get
            {
                return Price - DayOpen;
            }
        }

        public double PercentChange
        {
            get
            {
                return (double)Math.Round(Change / Price, 4);
            }
        }
    }
}
```

The two properties that you'll set when you create stocks are `Symbol` (for example, MSFT for Microsoft) and `Price`. The other properties depend on how and when you set `Price`. The first time you set `Price`, the value gets propagated to `DayOpen`. After that, when you set `Price`, the app calculates the `Change` and `PercentChange` property values based on the difference between `Price` and `DayOpen`.

Create the StockTickerHub and StockTicker classes

You'll use the SignalR Hub API to handle server-to-client interaction. A `StockTickerHub` class that derives from the SignalR `Hub` class will handle receiving connections and method calls from clients. You also need to maintain stock data and run a `Timer` object. The `Timer` object will periodically trigger price updates independent of client connections. You can't put these functions in a `Hub` class, because Hubs are transient. The app creates a `Hub` class instance for each task on the hub, like connections and calls from the client to the server. So the mechanism that keeps stock data, updates prices, and broadcasts the price updates has to run in a separate class. You'll name the class `StockTicker`.



You only want one instance of the `StockTicker` class to run on the server, so you'll need to set up a reference from each `StockTickerHub` instance to the singleton `StockTicker` instance. The `StockTicker` class has to broadcast to clients because it has the stock data and triggers updates, but `StockTicker` isn't a `Hub` class. The `StockTicker` class has to get a reference to the SignalR Hub connection context object. It can then use the SignalR connection context object to broadcast to clients.

Create StockTickerHub.cs

1. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.

2. In **Add New Item - SignalR.StockTicker**, select **Installed** > **Visual C#** > **Web** > **SignalR** and then select **SignalR Hub Class (v2)**.

3. Name the class *StockTickerHub* and add it to the project.

   This step creates the *StockTickerHub.cs* class file. Simultaneously, it adds a set of script files and assembly references that supports SignalR to the project.

4. Replace the code in the *StockTickerHub.cs* file with this code:

   C#

```
using System.Collections.Generic;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    [HubName("stockTickerMini")]
    public class StockTickerHub : Hub
    {
        private readonly StockTicker _stockTicker;

        public StockTickerHub() : this(StockTicker.Instance) { }

        public StockTickerHub(StockTicker stockTicker)
        {
            _stockTicker = stockTicker;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stockTicker.GetAllStocks();
        }
    }
}
```

5. Save the file.

The app uses the Hub class to define methods the clients can call on the server. You're defining one method: `GetAllStocks()`. When a client initially connects to the server, it will call this method to get a list of all of the stocks with their current prices. The method can

run synchronously and return `IEnumerable<Stock>` because it's returning data from memory.

If the method had to get the data by doing something that would involve waiting, like a database lookup or a web service call, you would specify `Task<IEnumerable<Stock>>` as the return value to enable asynchronous processing. For more information, see ASP.NET SignalR Hubs API Guide - Server - When to execute asynchronously.

The `HubName` attribute specifies how the app will reference the Hub in JavaScript code on the client. The default name on the client if you don't use this attribute, is a camelCase version of the class name, which in this case would be `stockTickerHub`.

As you'll see later when you create the `StockTicker` class, the app creates a singleton instance of that class in its static `Instance` property. That singleton instance of `StockTicker` is in memory no matter how many clients connect or disconnect. That instance is what the `GetAllStocks()` method uses to return current stock information.

Create StockTicker.cs

1. In **Solution Explorer**, right-click the project and select **Add** > **Class**.
2. Name the class *StockTicker* and add it to the project.
3. Replace the code in the *StockTicker.cs* file with this code:

C#

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    public class StockTicker
    {
        // Singleton instance
        private readonly static Lazy<StockTicker> _instance = new
Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>
().Clients));


        private readonly ConcurrentDictionary<string, Stock> _stocks = new
```

```
ConcurrentDictionary<string, Stock>();

        private readonly object _updateStockPricesLock = new object();

        //stock can go up or down by a percentage of this factor on each 
change
        private readonly double _rangePercent = .002;

        private readonly TimeSpan _updateInterval = 
TimeSpan.FromMilliseconds(250);
        private readonly Random _updateOrNotRandom = new Random();

        private readonly Timer _timer;
        private volatile bool _updatingStockPrices = false;

        private StockTicker(IHubConnectionContext<dynamic> clients)
        {
            Clients = clients;

            _stocks.Clear();
            var stocks = new List<Stock>
            {
                new Stock { Symbol = "MSFT", Price = 30.31m },
                new Stock { Symbol = "APPL", Price = 578.18m },
                new Stock { Symbol = "GOOG", Price = 570.30m }
            };
            stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

            _timer = new Timer(UpdateStockPrices, null, _updateInterval, 
_updateInterval);

        }

        public static StockTicker Instance
        {
            get
            {
                return _instance.Value;
            }
        }

        private IHubConnectionContext<dynamic> Clients
        {
            get;
            set;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stocks.Values;

        }
        private void UpdateStockPrices(object state)
        {
            lock (_updateStockPricesLock)
            {
                if (!_updatingStockPrices)
                {
                    _updatingStockPrices = true;

                    foreach (var stock in _stocks.Values)
                    {
                        if (TryUpdateStockPrice(stock))
                        {
                            BroadcastStockPrice(stock);
                        }
                    }

                    _updatingStockPrices = false;
                }
            }
        }

        private bool TryUpdateStockPrice(Stock stock)
        {
            // Randomly choose whether to update this stock or not
            var r = _updateOrNotRandom.NextDouble();
            if (r > .1)
            {
                return false;
            }

            // Update the stock price by a random factor of the range per-
cent
            var random = new Random((int)Math.Floor(stock.Price));
            var percentChange = random.NextDouble() * _rangePercent;
            var pos = random.NextDouble() > .51;
            var change = Math.Round(stock.Price * (decimal)percentChange,
2);
            change = pos ? change : -change;

            stock.Price += change;
            return true;
        }

        private void BroadcastStockPrice(Stock stock)
        {
            Clients.All.updateStockPrice(stock);
        }


    }
```

```
 }
```

Since all threads will be running the same instance of StockTicker code, the StockTicker class has to be thread-safe.

Examine the server code

If you examine the server code, it will help you understand how the app works.

Storing the singleton instance in a static field

The code initializes the static `_instance` field that backs the `Instance` property with an instance of the class. Because the constructor is private, it's the only instance of the class that the app can create. The app uses Lazy initialization for the `_instance` field. It's not for performance reasons. It's to make sure the instance creation is thread-safe.

```C#
private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(()
=> new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>
().Clients));

public static StockTicker Instance
{
    get
    {
        return _instance.Value;
    }
}
```

Each time a client connects to the server, a new instance of the StockTickerHub class running in a separate thread gets the StockTicker singleton instance from the `StockTicker.Instance` static property, as you saw earlier in the `StockTickerHub` class.

Storing stock data in a ConcurrentDictionary

The constructor initializes the `_stocks` collection with some sample stock data, and `GetAllStocks` returns the stocks. As you saw earlier, this collection of stocks is returned by

`StockTickerHub.GetAllStocks`, which is a server method in the `Hub` class that clients can call.

C#

```
private readonly ConcurrentDictionary<string, Stock> _stocks = new
ConcurrentDictionary<string, Stock>();
```

C#

```
private StockTicker(IHubConnectionContext<dynamic> clients)
{
    Clients = clients;

    _stocks.Clear();
    var stocks = new List<Stock>
    {
        new Stock { Symbol = "MSFT", Price = 30.31m },
        new Stock { Symbol = "APPL", Price = 578.18m },
        new Stock { Symbol = "GOOG", Price = 570.30m }
    };
    stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

    _timer = new Timer(UpdateStockPrices, null, _updateInterval,
_updateInterval);
}

public IEnumerable<Stock> GetAllStocks()
{
    return _stocks.Values;
}
```

The stocks collection is defined as a ConcurrentDictionary type for thread safety. As an alternative, you could use a Dictionary object and explicitly lock the dictionary when you make changes to it.

For this sample application, it's OK to store application data in memory and to lose the data when the app disposes of the `StockTicker` instance. In a real application, you would work with a back-end data store like a database.

Periodically updating stock prices

The constructor starts up a `Timer` object that periodically calls methods that update stock prices on a random basis.

C#

```
_timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);

private void UpdateStockPrices(object state)
{
    lock (_updateStockPricesLock)
    {
        if (!_updatingStockPrices)
        {
            _updatingStockPrices = true;

            foreach (var stock in _stocks.Values)
            {
                if (TryUpdateStockPrice(stock))
                {
                    BroadcastStockPrice(stock);
                }
            }

            _updatingStockPrices = false;
        }
    }
}

private bool TryUpdateStockPrice(Stock stock)
{
    // Randomly choose whether to update this stock or not
    var r = _updateOrNotRandom.NextDouble();
    if (r > .1)
    {
        return false;
    }

    // Update the stock price by a random factor of the range percent
    var random = new Random((int)Math.Floor(stock.Price));
    var percentChange = random.NextDouble() * _rangePercent;
    var pos = random.NextDouble() > .51;
    var change = Math.Round(stock.Price * (decimal)percentChange, 2);
    change = pos ? change : -change;

    stock.Price += change;
    return true;
}
```

`Timer` calls `UpdateStockPrices`, which passes in null in the state parameter. Before updating prices, the app takes a lock on the `_updateStockPricesLock` object. The code checks if another thread is already updating prices, and then it calls `TryUpdateStockPrice` on each

stock in the list. The `TryUpdateStockPrice` method decides whether to change the stock price, and how much to change it. If the stock price changes, the app calls `BroadcastStockPrice` to broadcast the stock price change to all connected clients.

The `_updatingStockPrices` flag designated volatile to make sure it is thread-safe.

C#

```
private volatile bool _updatingStockPrices = false;
```

In a real application, the `TryUpdateStockPrice` method would call a web service to look up the price. In this code, the app uses a random number generator to make changes randomly.

Getting the SignalR context so that the StockTicker class can broadcast to clients

Because the price changes originate here in the `StockTicker` object, it's the object that needs to call an `updateStockPrice` method on all connected clients. In a `Hub` class, you have an API for calling client methods, but `StockTicker` doesn't derive from the `Hub` class and doesn't have a reference to any `Hub` object. To broadcast to connected clients, the `StockTicker` class has to get the SignalR context instance for the `StockTickerHub` class and use that to call methods on clients.

The code gets a reference to the SignalR context when it creates the singleton class instance, passes that reference to the constructor, and the constructor puts it in the `Clients` property.

There are two reasons why you want to get the context only once: getting the context is an expensive task, and getting it once makes sure the app preserves the intended order of messages sent to the clients.

C#

```
private readonly static Lazy<StockTicker> _instance =
    new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>
().Clients));

private StockTicker(IHubConnectionContext<dynamic> clients)
{

    Clients = clients;
```

```
        // Remainder of constructor ...
    }

    private IHubConnectionContext<dynamic> Clients
    {
        get;
        set;
    }

    private void BroadcastStockPrice(Stock stock)
    {
        Clients.All.updateStockPrice(stock);
    }
```

Getting the `Clients` property of the context and putting it in the `StockTickerClient` property lets you write code to call client methods that looks the same as it would in a `Hub` class. For instance, to broadcast to all clients you can write `Clients.All.updateStockPrice(stock)`.

The `updateStockPrice` method that you're calling in `BroadcastStockPrice` doesn't exist yet. You'll add it later when you write code that runs on the client. You can refer to `updateStockPrice` here because `Clients.All` is dynamic, which means the app will evaluate the expression at runtime. When this method call executes, SignalR will send the method name and the parameter value to the client, and if the client has a method named `updateStockPrice`, the app will call that method and pass the parameter value to it.

`Clients.All` means send to all clients. SignalR gives you other options to specify which clients or groups of clients to send to. For more information, see HubConnectionContext .

Register the SignalR route

The server needs to know which URL to intercept and direct to SignalR. To do that, add an OWIN startup class:

1. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
2. In **Add New Item - SignalR.StockTicker** select **Installed** > **Visual C#** > **Web** and then select **OWIN Startup Class**.
3. Name the class *Startup* and select **OK**.
4. Replace the default code in the *Startup.cs* file with this code:

C#

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(SignalR.StockTicker.Startup))]

namespace SignalR.StockTicker
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go
here
            app.MapSignalR();
        }

    }
}
```

You have now finished setting up the server code. In the next section, you'll set up the client.

Set up the client code

In this section, you set up the code that runs on the client.

Create the HTML page and JavaScript file

The HTML page will display the data and the JavaScript file will organize the data.

Create StockTicker.html

First, you'll add the HTML client.

1. In **Solution Explorer**, right-click the project and select **Add** > **HTML Page**.

2. Name the file *StockTicker* and select **OK**.

3. Replace the default code in the *StockTicker.html* file with this code:

HTML

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>ASP.NET SignalR Stock Ticker</title>
    <style>
        body {
            font-family: 'Segoe UI', Arial, Helvetica, sans-serif;
            font-size: 16px;
        }
        #stockTable table {
            border-collapse: collapse;
        }
            #stockTable table th, #stockTable table td {
                padding: 2px 6px;
            }
            #stockTable table td {
                text-align: right;
            }
        #stockTable .loading td {
            text-align: left;
        }
    </style>
</head>
<body>
    <h1>ASP.NET SignalR Stock Ticker Sample</h1>

    <h2>Live Stock Table</h2>
    <div id="stockTable">
        <table border="1">
            <thead>
                <tr><th>Symbol</th><th>Price</th><th>Open</th>
<th>Change</th><th>%</th></tr>
            </thead>
            <tbody>
                <tr class="loading"><td colspan="5">loading...</td></tr>
            </tbody>
        </table>
    </div>

    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="/Scripts/jquery-1.10.2.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="/Scripts/jquery.signalR-2.1.0.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="/signalr/hubs"></script>

    <!--Reference the StockTicker script. -->
    <script src="StockTicker.js"></script>
</body>
</html>
```

The HTML creates a table with five columns, a header row, and a data row with a single cell that spans all five columns. The data row shows "loading..." momentarily when the app starts. JavaScript code will remove that row and add in its place rows with stock data retrieved from the server.

The script tags specify:

- The jQuery script file.
- The SignalR core script file.
- The SignalR proxies script file.
- A StockTicker script file that you'll create later.

The app dynamically generates the SignalR proxies script file. It specifies the "/signalr/hubs" URL and defines proxy methods for the methods on the Hub class, in this case, for `StockTickerHub.GetAllStocks`. If you prefer, you can generate this JavaScript file manually by using SignalR Utilities . Don't forget to disable dynamic file creation in the `MapHubs` method call.

4. In **Solution Explorer**, expand **Scripts**.

   Script libraries for jQuery and SignalR are visible in the project.

   Important

   The package manager will install a later version of the SignalR scripts.

5. Update the script references in the code block to correspond to the versions of the script files in the project.

6. In **Solution Explorer**, right-click *StockTicker.html*, and then select **Set as Start Page**.

Create StockTicker.js

Now create the JavaScript file.

1. In **Solution Explorer**, right-click the project and select **Add** > **JavaScript File**.

2. Name the file *StockTicker* and select **OK**.

3. Add this code to the *StockTicker.js* file:

JavaScript

```
// A simple templating method for replacing placeholders enclosed in curly
braces.
if (!String.prototype.supplant) {
    String.prototype.supplant = function (o) {
        return this.replace(/{([^{}]*)}/g,
            function (a, b) {
                var r = o[b];
                return typeof r === 'string' || typeof r === 'number' ? r
: a;
            }
        );
    };
}

$(function () {

    var ticker = $.connection.stockTickerMini, // the generated client-
side hub proxy
        up = '▲',
        down = '▼',
        $stockTable = $('#stockTable'),
        $stockTableBody = $stockTable.find('tbody'),
        rowTemplate = '<tr data-symbol="{Symbol}"><td>{Symbol}</td><td>
{Price}</td><td>{DayOpen}</td><td>{Direction} {Change}</td><td>
{PercentChange}</td></tr>';

    function formatStock(stock) {
        return $.extend(stock, {
            Price: stock.Price.toFixed(2),
            PercentChange: (stock.PercentChange * 100).toFixed(2) + '%',
            Direction: stock.Change === 0 ? '' : stock.Change >= 0 ? up :
down
        });
    }

    function init() {
        ticker.server.getAllStocks().done(function (stocks) {
            $stockTableBody.empty();
            $.each(stocks, function () {
                var stock = formatStock(this);
                $stockTableBody.append(rowTemplate.supplant(stock));
                });
            });
        }

        // Add a client-side hub method that the server will call
        ticker.client.updateStockPrice = function (stock) {
            var displayStock = formatStock(stock),
                $row = $(rowTemplate.supplant(displayStock));

            $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
                .replaceWith($row);
            }

        // Start the connection
        $.connection.hub.start().done(init);

    });
```

Examine the client code

If you examine the client code, it will help you learn how the client code interacts with the server code to make the app work.

Starting the connection

`$.connection` refers to the SignalR proxies. The code gets a reference to the proxy for the `StockTickerHub` class and puts it in the `ticker` variable. The proxy name is the name that was set by the `HubName` attribute:

JavaScript

```javascript
var ticker = $.connection.stockTickerMini
```

C#

```csharp
[HubName("stockTickerMini")]
public class StockTickerHub : Hub
```

After you define all the variables and functions, the last line of code in the file initializes the SignalR connection by calling the SignalR `start` function. The `start` function executes

asynchronously and returns a jQuery Deferred object . You can call the done function to specify the function to call when the app finishes the asynchronous action.

JavaScript

```
$.connection.hub.start().done(init);
```

Getting all the stocks

The `init` function calls the `getAllStocks` function on the server and uses the information that the server returns to update the stock table. Notice that, by default, you have to use camelCasing on the client even though the method name is pascal-cased on the server. The camelCasing rule only applies to methods, not objects. For example, you refer to `stock.Symbol` and `stock.Price`, not `stock.symbol` or `stock.price`.

JavaScript

```
function init() {
    ticker.server.getAllStocks().done(function (stocks) {
        $stockTableBody.empty();
        $.each(stocks, function () {
            var stock = formatStock(this);
            $stockTableBody.append(rowTemplate.supplant(stock));
        });
    });
}
```

C#

```
public IEnumerable<Stock> GetAllStocks()
{
    return _stockTicker.GetAllStocks();
}
```

In the `init` method, the app creates HTML for a table row for each stock object received from the server by calling `formatStock` to format properties of the `stock` object, and then by calling `supplant` to replace placeholders in the `rowTemplate` variable with the `stock` object property values. The resulting HTML is then appended to the stock table.

Note

You call `init` by passing it in as a `callback` function that executes after the asynchronous `start` function finishes. If you called `init` as a separate JavaScript statement after calling `start`, the function would fail because it would run immediately without waiting for the start function to finish establishing the connection. In that case, the `init` function would try to call the `getAllStocks` function before the app establishes a server connection.

Getting updated stock prices

When the server changes a stock's price, it calls the `updateStockPrice` on connected clients. The app adds the function to the client property of the `stockTicker` proxy to make it available to calls from the server.

JavaScript

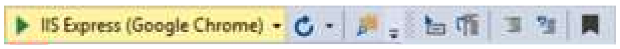
```
ticker.client.updateStockPrice = function (stock) {
    var displayStock = formatStock(stock),
        $row = $(rowTemplate.supplant(displayStock));

    $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
        .replaceWith($row);
    }
```

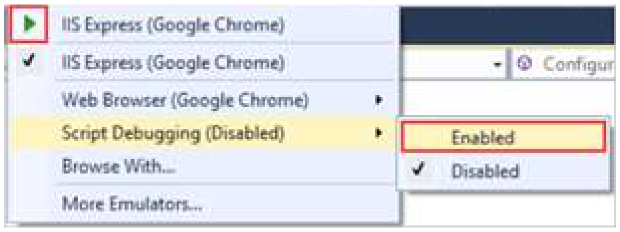
The `updateStockPrice` function formats a stock object received from the server into a table row the same way as in the `init` function. Instead of appending the row to the table, it finds the stock's current row in the table and replaces that row with the new one.

Test the application

You can test the app to make sure it's working. You'll see all browser windows display the live stock table with stock prices fluctuating.

1. In the toolbar, turn on **Script Debugging** and then select the play button to run the app in Debug mode.




IIS Express (Google Chrome)
IIS Express (Google Chrome)
Web Browser (Google Chrome)
Script Debugging (Disabled)
Browse With...
More Emulators...
Configur
Enabled
Disabled

A browser window will open displaying the **Live Stock Table**. The stock table initially shows the "loading..." line, then, after a short time, the app shows the initial stock data, and then the stock prices start to change.

2. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.

   The initial stock display is the same as the first browser and changes happen simultaneously.

3. Close all browsers, open a new browser, and go to the same URL.

   The StockTicker singleton object continued to run in the server. The **Live Stock Table** shows that the stocks have continued to change. You don't see the initial table with zero change figures.

4. Close the browser.

Enable logging

SignalR has a built-in logging function that you can enable on the client to aid in troubleshooting. In this section, you enable logging and see examples that show how logs tell you which of the following transport methods SignalR is using:

- WebSockets , supported by IIS 8 and current browsers.
- Server-sent events , supported by browsers other than Internet Explorer.
- Forever frame , supported by Internet Explorer.
- Ajax long polling , supported by all browsers.

For any given connection, SignalR chooses the best transport method that both the server and the client support.

1. Open *StockTicker.js*.
2. Add this highlighted line of code to enable logging immediately before the code that

initializes the connection at the end of the file:

JavaScript

```javascript
// Start the connection
$.connection.hub.logging = true;
$.connection.hub.start().done(init);
```

3. Press **F5** to run the project.

4. Open your browser's developer tools window, and select the Console to see the logs. You might have to refresh the page to see the logs of SignalR negotiating the transport method for a new connection.

   - If you're running Internet Explorer 10 on Windows 8 (IIS 8), the transport method is **WebSockets**.
   - If you're running Internet Explorer 10 on Windows 7 (IIS 7.5), the transport method is **iframe**.
   - If you're running Firefox 19 on Windows 8 (IIS 8), the transport method is **WebSockets**.

     Tip

     In Firefox, install the Firebug add-in to get a Console window.

   - If you're running Firefox 19 on Windows 7 (IIS 7.5), the transport method is **server-sent** events.

Install the StockTicker sample

The Microsoft.AspNet.SignalR.Sample installs the StockTicker application. The NuGet package includes more features than the simplified version that you created from scratch. In this section of the tutorial, you install the NuGet package and review the new features and the code that implements them.

Important

If you install the package without performing the earlier steps of this tutorial, you must add an OWIN startup class to your project. This readme.txt file for the NuGet package explains this step.

Install the SignalR.Sample NuGet package

1. In **Solution Explorer**, right-click the project and select **Manage NuGet Packages**.
2. In **NuGet Package manager: SignalR.StockTicker**, select **Browse**.
3. From **Package source**, select **nuget.org**.
4. Enter *SignalR.Sample* in the search box and select **Microsoft.AspNet.SignalR.Sample** > **Install**.
5. In **Solution Explorer**, expand the *SignalR.Sample* folder.

   Installing the SignalR.Sample package created the folder and its contents.

6. In the *SignalR.Sample* folder, right-click *StockTicker.html*, and then select **Set As Start Page**.

   Note

   Installing The SignalR.Sample NuGet package might change the version of jQuery that you have in your *Scripts* folder. The new *StockTicker.html* file that the package installs in the *SignalR.Sample* folder will be in sync with the jQuery version that the package installs, but if you want to run your original *StockTicker.html* file again, you might have to update the jQuery reference in the script tag first.

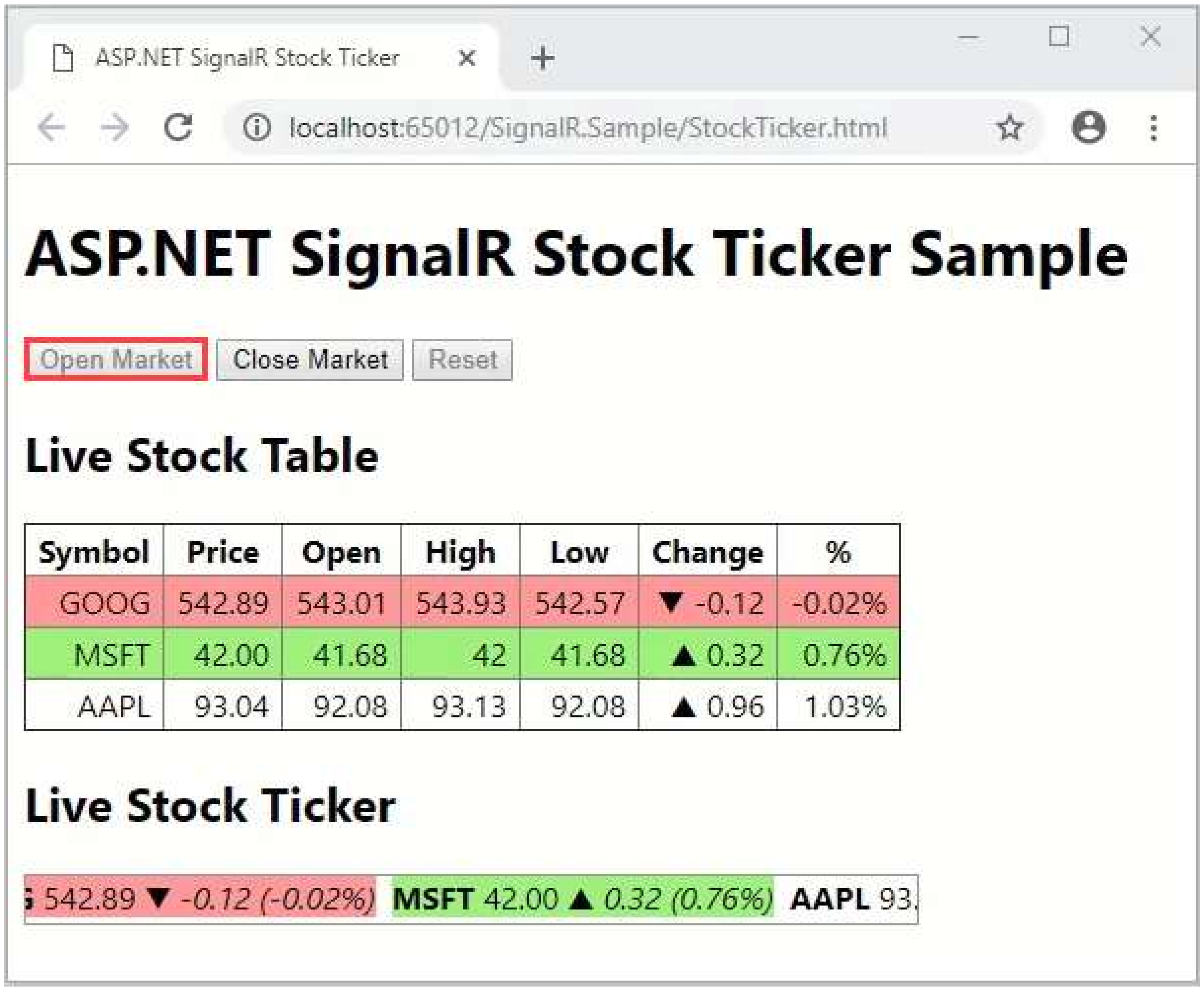
Run the application

The table that you saw in the first app had useful features. The full stock ticker application shows new features: a horizontally scrolling window that shows the stock data and stocks that change color as they rise and fall.

1. Press **F5** to run the app.

   When you run the app for the first time, the "market" is "closed" and you see a static table and a ticker window that isn't scrolling.

2. Select **Open Market**.


ASP.NET SignalR Stock Ticker
localhost:65012/SignalR.Sample/StockTicker.html

ASP.NET SignalR Stock Ticker Sample

Open Market
Close Market
Reset

Live Stock Table

Symbol	Price	Open	High	Low	Change	%
GOOG	542.89	543.01	543.93	542.57	▼ -0.12	-0.02%
MSFT	42.00	41.68	42	41.68	▲ 0.32	0.76%
AAPL	93.04	92.08	93.13	92.08	▲ 0.96	1.03%

Live Stock Ticker

542.89 ▼ -0.12 (-0.02%) **MSFT** 42.00 ▲ 0.32 (0.76%) **AAPL** 93.

- The **Live Stock Ticker** box starts to scroll horizontally, and the server starts to periodically broadcast stock price changes on a random basis.
- Each time a stock price changes, the app updates both the **Live Stock Table** and the **Live Stock Ticker**.
- When a stock's price change is positive, the app shows the stock with a green background.
- When the change is negative, the app shows the stock with a red background.

3. Select **Close Market**.

- The table updates stop.
- The ticker stops scrolling.

4. Select **Reset**.

- All stock data is reset.
- The app restores the initial state before price changes started.

5. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.
6. You see the same data dynamically updated at the same time in each browser.
7. When you select any of the controls, all browsers respond the same way at the same time.

Live Stock Ticker display

The **Live Stock Ticker** display is an unordered list in a `<div>` element formatted into a single line by CSS styles. The app initializes and updates the ticker the same way as the table: by replacing placeholders in an `<li>` template string and dynamically adding the `<li>` elements to the `<ul>` element. The app includes scrolling by using the jQuery `animate` function to vary the margin-left of the unordered list within the `<div>`.

SignalR.Sample StockTicker.html

The stock ticker HTML code:

HTML

```html
<h2>Live Stock Ticker</h2>
<div id="stockTicker">
    <div class="inner">
        <ul>
            <li class="loading">loading...</li>
        </ul>
    </div>
</div>
```

SignalR.Sample StockTicker.css

The stock ticker CSS code:

HTML

```
#stockTicker {
    overflow: hidden;
    width: 450px;
    height: 24px;
    border: 1px solid #999;
    }

    #stockTicker .inner {
        width: 9999px;
    }

    #stockTicker ul {
        display: inline-block;
        list-style-type: none;
        margin: 0;
        padding: 0;
    }

    #stockTicker li {
        display: inline-block;
        margin-right: 8px;
    }

    /*<li data-symbol="{Symbol}"><span class="symbol">{Symbol}</span><span
class="price">{Price}</span><span class="change">{PercentChange}</span></li>*/
    #stockTicker .symbol {
        font-weight: bold;
    }

    #stockTicker .change {
        font-style: italic;
    }
```

SignalR.Sample SignalR.StockTicker.js

The jQuery code that makes it scroll:

JavaScript

```javascript
function scrollTicker() {
    var w = $stockTickerUl.width();
    $stockTickerUl.css({ marginLeft: w });
    $stockTickerUl.animate({ marginLeft: -w }, 15000, 'linear', scrollTicker);
}
```

Additional methods on the server that the client can call

To add flexibility to the app, there are additional methods the app can call.

SignalR.Sample StockTickerHub.cs

The `StockTickerHub` class defines four additional methods that the client can call:

C#

```csharp
public string GetMarketState()
{
    return _stockTicker.MarketState.ToString();
}

public void OpenMarket()
{
    _stockTicker.OpenMarket();
}

public void CloseMarket()
{
    _stockTicker.CloseMarket();
}

public void Reset()
{
    _stockTicker.Reset();
}
```

The app calls `OpenMarket`, `CloseMarket`, and `Reset` in response to the buttons at the top of the page. They demonstrate the pattern of one client triggering a change in state immediately propagated to all clients. Each of these methods calls a method in the `StockTicker` class that causes the market state change and then broadcasts the new state.

SignalR.Sample StockTicker.cs

In the `StockTicker` class, the app maintains the state of the market with a `MarketState` property that returns a `MarketState` enum value:

C#

```
public MarketState MarketState
{
    get { return _marketState; }
    private set { _marketState = value; }
}

public enum MarketState
{
    Closed,
    Open
}
```

Each of the methods that change the market state do so inside a lock block because the `StockTicker` class has to be thread-safe:

C#

```
public void OpenMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Open)
        {
            _timer = new Timer(UpdateStockPrices, null, _updateInterval,
_updateInterval);
            MarketState = MarketState.Open;
            BroadcastMarketStateChange(MarketState.Open);
        }
    }
}

public void CloseMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState == MarketState.Open)
        {
            if (_timer != null)
            {
                _timer.Dispose();
            }
            MarketState = MarketState.Closed;
            BroadcastMarketStateChange(MarketState.Closed);
        }
    }
}

public void Reset()

{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Closed)
        {
            throw new InvalidOperationException("Market must be closed before
it can be reset.");
        }
        LoadDefaultStocks();
        BroadcastMarketReset();
    }
}
```

To make sure this code is thread-safe, the `_marketState` field that backs the `MarketState` property designated `volatile`:

C#

```
private volatile MarketState _marketState;
```

The `BroadcastMarketStateChange` and `BroadcastMarketReset` methods are similar to the BroadcastStockPrice method that you already saw, except they call different methods defined at the client:

C#

```
private void BroadcastMarketStateChange(MarketState marketState)
{
    switch (marketState)
    {
        case MarketState.Open:
            Clients.All.marketOpened();
            break;
        case MarketState.Closed:
            Clients.All.marketClosed();
            break;
        default:
            break;
    }
}

private void BroadcastMarketReset()
{
    Clients.All.marketReset();
}
```

Additional functions on the client that the server can call

The `updateStockPrice` function now handles both the table and the ticker display, and it uses `jQuery.Color` to flash red and green colors.

New functions in *SignalR.StockTicker.js* enable and disable the buttons based on market state. They also stop or start the **Live Stock Ticker** horizontal scrolling. Since many functions are being added to `ticker.client`, the app uses the jQuery extend function to add them.

JavaScript

```
$.extend(ticker.client, {
    updateStockPrice: function (stock) {
        var displayStock = formatStock(stock),
            $row = $(rowTemplate.supplant(displayStock)),
            $li = $(liTemplate.supplant(displayStock)),
            bg = stock.LastChange === 0
                ? '255,216,0' // yellow
                : stock.LastChange > 0
                    ? '154,240,117' // green
                    : '255,148,148'; // red

        $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
            .replaceWith($row);
        $stockTickerUl.find('li[data-symbol=' + stock.Symbol + ']')
            .replaceWith($li);

        $row.flash(bg, 1000);
        $li.flash(bg, 1000);
    },

    marketOpened: function () {
        $("#open").prop("disabled", true);
        $("#close").prop("disabled", false);
        $("#reset").prop("disabled", true);
        scrollTicker();
    },

    marketClosed: function () {
        $("#open").prop("disabled", false);
        $("#close").prop("disabled", true);
        $("#reset").prop("disabled", false);
        stopTicker();
    },

    marketReset: function () {
        return init();
    }
});
```

Additional client setup after establishing the connection

After the client establishes the connection, it has some additional work to do:

- Find out if the market is open or closed to call the appropriate `marketOpened` or `marketClosed` function.
- Attach the server method calls to the buttons.

JavaScript

```javascript
$.connection.hub.start()
    .pipe(init)
    .pipe(function () {
        return ticker.server.getMarketState();
    })
    .done(function (state) {
        if (state === 'Open') {
            ticker.client.marketOpened();
        } else {
            ticker.client.marketClosed();
        }

        // Wire up the buttons
        $("#open").click(function () {
            ticker.server.openMarket();
        });

        $("#close").click(function () {
            ticker.server.closeMarket();
        });

        $("#reset").click(function () {
            ticker.server.reset();
        });
    });
```

The server methods aren't wired up to the buttons until after the app establishes the connection. It's so the code can't call the server methods before they're available.

Additional resources

In this tutorial you've learned how to program a SignalR application that broadcasts messages from the server to all connected clients. Now you can broadcast messages on a periodic basis and in response to notifications from any client. You can use the concept of multi-threaded singleton instance to maintain server state in multi-player online game scenarios. For an example, see the ShootR game based on SignalR .

For tutorials that show peer-to-peer communication scenarios, see Getting Started with SignalR and Real-Time Updating with SignalR.

For more about SignalR, see the following resources:

- ASP.NET SignalR
- SignalR Project
- SignalR GitHub and Samples
- SignalR Wiki

Next steps

In this tutorial, you:

- ✓ Created the project
- ✓ Set up the server code
- ✓ Examined the server code
- ✓ Set up the client code
- ✓ Examined the client code
- ✓ Tested the application
- ✓ Enabled logging

Advance to the next article to learn how to create a real-time web application that uses ASP.NET SignalR 2.

Create real-time web app with SignalR