

## [JavaScript](http://www.sitepoint.com/javascript/)

# Build a Real-time SignalR Dashboard with AngularJS



(<http://www.sitepoint.com/author/louiebacaj/>)

Louie Bacaj (<http://www.sitepoint.com/author/louiebacaj/>)

Published March 20, 2015

 [Tweet \(Https://Twitter.Com/Share?Text=Build+A+Real-Time+SignalR+Dashboard+With+AngularJS&Via=Sitepointdotcom\)](https://twitter.com/Share?Text=Build+A+Real-Time+SignalR+Dashboard+With+AngularJS&Via=Sitepointdotcom)

[Subscribe \(Https://Confirmsubscription.Com/H/Y/1FD5B523FA48AA2B\)](https://confirmsubscription.com/H/Y/1FD5B523FA48AA2B)

Let's build a real-time service dashboard!



Our service dashboard will show us real data in real time. It will show us what's happening on our server and our micro service in near real time, asynchronous, non-blocking fashion.

**Take a look at what a full client can look like [here](http://angulardsignalrdashboardlouiebacaj.azurewebsites.net/) (<http://angulardsignalrdashboardlouiebacaj.azurewebsites.net/>).**

**A demo of the server can be seen [here](http://sitepointsignal.cloudapp.net/) (<http://sitepointsignal.cloudapp.net/>).**

We'll build a smaller version of this dashboard using the AngularJS framework and lots of cool real time charts with lots of real time data. We'll also build our service using the [SignalR](http://signalr.net/) (<http://signalr.net/>) and [Web API](http://www.asp.net/web-api) (<http://www.asp.net/web-api>) libraries from .NET 4.5.

## Technology Architecture

### The Client

[AngularJS](https://angularjs.org/) (<https://angularjs.org/>) forces great application development practices right out of the box. Everything is injected in, which means there is low coupling of dependencies. Additionally, Angular has a great separation between views, models and controllers.

Angular compliments .NET here by allowing the server side code to remain small, manageable and testable. The server side code is leveraged solely for its strengths – which is to do the heavy lifting.

### The Server

Using SignalR with Web API for .NET 4.5 is very similar to using Node.js with Socket.IO, and allows for the same type of non-blocking, asynchronous push from the server to subscribing clients. SignalR uses web sockets underneath, but because it abstracts away the communication, it will fall back to whatever technology the client browser supports when running inside Angular. (For example, it may fall back to long polling for older browsers.)

Additionally, with the dynamic tag and the magic of Json.NET, JavaScript is treated like a first class citizen by the .NET framework. In fact, it is often easier to consume Web API and SignalR technologies in JavaScript than even through native .NET clients, because they were built with JavaScript in mind.

# The Meat and Potatoes

## Get Setup

All of the AngularJS code used in this tutorial can be found [here](https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient) (<https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient>).

I will go over creating this with your favorite text editor and plain folders, as well as with Visual Studio for those creating a project.

## Setup with Plain Text Files

The folder and file structure will look like this:

---

```
root
  app      (Angular application specific JavaScript)
  Content  (CSS etc.)
  Scripts  (Referenced JavaScript etc.)
  ...
  index.html
```

---

## Main Dependencies

You will need to download the following files:

- [jQuery \(http://jquery.com/download/\)](http://jquery.com/download/) (choose the “Download the compressed, production jQuery 2.1.1” link)
- [AngularJS \(https://angularjs.org/\)](https://angularjs.org/) (click on the large Download option, then click the latest version of Angular 1.3.+)
- [Bootstrap \(http://getbootstrap.com/getting-started/#download\)](http://getbootstrap.com/getting-started/#download) (click the “Download Bootstrap” option)
- [SignalR \(https://github.com/SignalR/bower-signalr\)](https://github.com/SignalR/bower-signalr) (click the “Download ZIP” button on the right)
- [D3.js \(http://d3js.org/\)](http://d3js.org/) (click the “d3.zip” link half way down the page)
- [Epoch \(http://fastly.github.io/epoch/\)](http://fastly.github.io/epoch/) (click the “Download v0.6.0 link)
- [ng-epoch \(https://github.com/dainbrump/ng-epoch\)](https://github.com/dainbrump/ng-epoch) (click the “Download ZIP” button on the right)
- [n3-pie \(https://github.com/n3-charts/pie-chart\)](https://github.com/n3-charts/pie-chart) (click the “Download ZIP” button on the right)

In our `Scripts` folder we will need:

- `jquery-2.1.1.min.js`
- `angular.min.js`
- `bootstrap.min.js`
- `jquery.signalR.min.js`
- `d3.min.js`
- `epoch.min.js`
- `pie-chart.min.js`

In our `Content` folder:







- `bootstrap.min.css`
- `epoch.min.css`

## Setup with Visual Studio


Setting this up through Visual Studio is extremely simple, if text files are too simplistic for you.

Simply set up an empty web application by going to `File -> New -> Project`, then select `Web` as the template type.

Select a template:

 Empty	 Web Forms	 MVC	 Web API
 Single Page Application	 Facebook		

An empty project template for creating ASP.NET applications. This template does not have any content in it.

[Learn more](#)[Change Authentication](#)Authentication: **No Authentication** **Windows Azure**☐ Create remote resources

Web Site

[Manage Subscriptions](#)

Add folders and core references for:

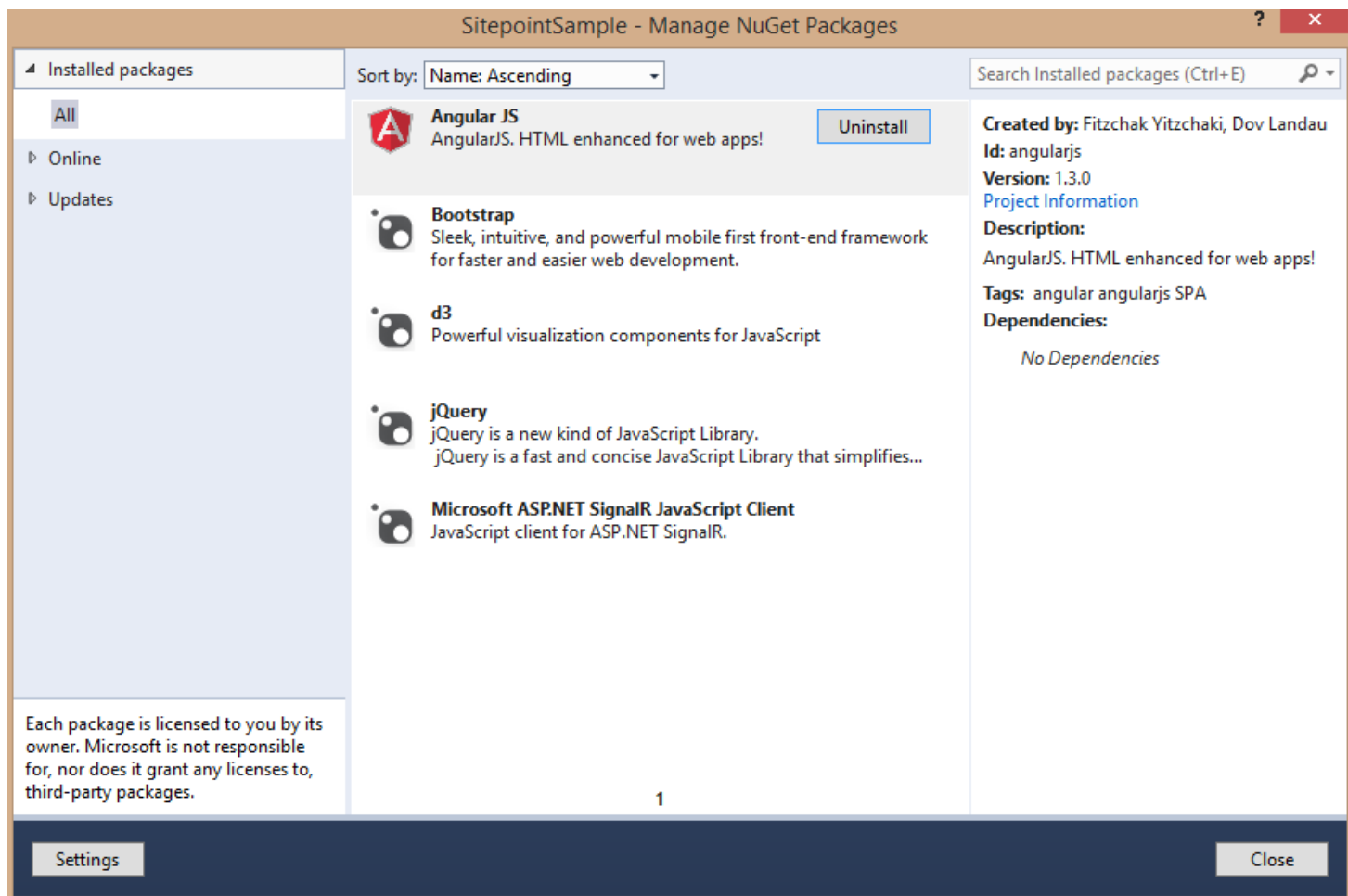
☐ Web Forms ☐ MVC ☐ Web API☐ Add unit testsTest project name: 

OK

Cancel

Then simply right click on the project, go to **Manage Nuget Packages** and search for and download jQuery, AngularJS, Bootstrap, D3 and the SignalR JavaScript Client.

After you download and install those, you should see them all in the Scripts and the Contents folders. Additionally, under installed Nuget Packages, you will see the following:



Finally, Nuget does not contain the Epoch, ng-epoch and n3 charting libraries, so you'll need to add them manually. Simply follow the steps detailed in the previous section to get these.

## Let's Write Our App

Now we are ready to write some code.

First, let's create our base `index.html` file that will house our Angular JavaScript code.

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" (http://www.w3.org/1999/xhtml)>
3 <head>
4   <meta charset="utf-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1">
7   <title>AngularJS - SignalR - ServiceDashboard</title>
8   <link rel="stylesheet" href="Content/bootstrap.min.css" />
9   <link rel="stylesheet" href="Content/epoch.min.css" />
10

```

```
11 <script src="Scripts/jquery-1.11.0.js"></script>
12 <script src="Scripts/bootstrap.min.js"></script>
13 <script src="Scripts/jquery.signalR-2.1.2.min.js"></script>
14 <script src="Scripts/angular.min.js"></script>
15
16 <script src="Scripts/d3.min.js"></script>
17 <script src="Scripts/epoch.min.js"></script>
18 <script src="Scripts/ng-epoch.js"></script>
19 <script src="Scripts/pie-chart.min.js"></script>
20
21 <script src="app/app.js"></script>
22 <script src="app/services.js"></script>
23 <script src="app/directives.js"></script>
24 <script src="app/controllers.js"></script>
25
26 </head>
27 <body ng-app="angularServiceDashboard">
28 </body>
29 </html>
```

There are a few things going on here. We are, first and foremost, adding all of our dependencies so they load up. Secondly, we are referencing a few new files (all of the files in the app folder) that do not exist yet. We will write those next.

Let's go into our app folder and create our `app.js` file. This is a very simple file.

```
1 'use strict';
2
3 var app = angular.module('angularServiceDashboard', ['ng.epoch', 'n3-pie-chart']);
4 app.value('backendServerUrl', 'http://sitepointsignal.cloudapp.net/ (http://sitepointsignal
```

This file does a few things for us. It sets up our main application module

`angularServiceDashboard` and injects in two of our external references – `ng.epoch`, which is our Epoch.js Directive for Angular, and the `n3-pie-chart`, which is a charting library made for Angular and is properly structured.

If you notice, we also inject in a value for the `backendServerUrl`, which of course is hosted somewhere else and which we plan to consume here.

Let's create a service factory class that will bind to the URL of the server. This will be our `services.js` file we referenced in our HTML, and it will go into the app folder:

```
1  'use strict';
2
3  app.factory('backendHubProxy', ['$rootScope', 'backendServerUrl',
4    function ($rootScope, backendServerUrl) {
5
6      function backendFactory(serverUrl, hubName) {
7        var connection = $.hubConnection(backendServerUrl);
8        var proxy = connection.createHubProxy(hubName);
9
10       connection.start().done(function () { });
11
12       return {
13         on: function (eventName, callback) {
14           proxy.on(eventName, function (result) {
15             $rootScope.$apply(function () {
16               if (callback) {
17                 callback(result);
18               }
19             });
20           });
21         },
22         invoke: function (methodName, callback) {
23           proxy.invoke(methodName)
24             .done(function (result) {
25               $rootScope.$apply(function () {
26                 if (callback) {
27                   callback(result);
28                 }
29               });
30             });
31         }
32       };
33     });
34
35     return backendFactory;
36   }]);
```



This bit of code uses the popular `on` and `off` (with no `off` since we don't need it here) subscription pattern, and encapsulates all of the communication with SignalR for our app by using an Angular factory.

This code may seem a bit overwhelming at first, but you will understand it better when we build our controllers. All it does is take in the URL of our back-end SignalR server and the SignalR hub name. (In SignalR you can use multiple hubs in the same server to push data.)

Additionally, this code allows the SignalR Server, which is sitting on another box somewhere, to call our app through the `on` method. It allows our app to call functions inside of the SignalR Server through the `invoke` method.

Next up, we need our controllers, which will bind our data from the service to our scope. Let's create a file called `controllers.js` in our app folder.

---

```
1  'use strict';
2
3  app.controller('PerformanceDataController', ['$scope', 'backendHubProxy',
4    function ($scope, backendHubProxy) {
5      console.log('trying to connect to service')
6      var performanceDataHub = backendHubProxy(backendHubProxy.defaultServer, 'performanceHul
7      console.log('connected to service')
8      $scope.currentRamNumber = 68;
9
10     performanceDataHub.on('broadcastPerformance', function (data) {
11       data.forEach(function (dataItem) {
12         switch(dataItem.categoryName) {
13           case 'Processor':
14             break;
15           case 'Memory':
16             $scope.currentRamNumber = dataItem.value;
17             break;
18           case 'Network In':
19             break;
20           case 'Network Out':
21             break;
22           case 'Disk Read Bytes/Sec':
23             break;
24           case 'Disk Write Bytes/Sec':
25             break;
26           default:
27             //default code block
```

```
28         break;
29     }
30 });
31 });
32 }
33 ]);
```

This controller does a few things here. It creates our Angular Service object and binds a callback function to it, so that the server has something to call in our controller.

You will see that we are looping through the JSON array returned by the server each time it calls us back. We then have a switch statement for each performance type. For now, we will set the RAM and come back and flesh out the rest.

As far as our directives are concerned, we really only need one for our Epoch charts. We'll use an open-source directive called `ng-epoch.js` (<https://github.com/dainbrump/ng-epoch>), which we already have a reference for in our stub `index.html` file.

We could split all of these charts into different directives, use some templates and use UI-Router (<https://github.com/angular-ui/ui-router>), but we'll keep things simple here and dump all our views in our `index.html` file.

Let's add our views to the `index.html` file now. We can do this by adding the following under the body tags:

```
1  <div class="row" ng-controller="PerformanceDataController">
2    <div class="col-lg-3 col-md-6">
3      <div class="panel panel-dashboard">
4        <div class="center">Memory Performance</div>
5        <div class="panel-body">
6          <div class="huge">{{currentRamNumber}}</div>
7          <div class="clearfix"></div>
8        </div>
9      </div>
10    </div>
11  </div>
12 </div>
```

This will simply create a place for the server to push back the RAM data. Data will first go to our service, then to the controller and then finally to the view.

It should look something like this:

## Memory Performance

1211

Now let's add some charting, which is what we really want to do. We will add a variable called `timestamp` for the `epoch.js` timeline. We'll also add an array called `chartEntry`, which we'll bind to our `epoch.ng` directive.

```
1  var timestamp = ((new Date()).getTime() / 1000) | 0;
2  var chartEntry = [];
```

Then let's map the data in our `switch` statement and add the rest of the required `epoch.js` data items. We could, of course, break this out further (such as use some more functions and filters), but we'll keep things simple for the sake of the tutorial.

```
1  'use strict';
2
3  app.controller('PerformanceDataController', ['$scope', 'backendHubProxy',
4    function ($scope, backendHubProxy) {
5      ...
6
7      $scope.currentRamNumber = 68;
8      $scope.realtimeArea = [{ label: 'Layer 1', values: [] }];
9
10     performanceDataHub.on('broadcastPerformance', function (data) {
11       var timestamp = ((new Date()).getTime() / 1000) | 0;
12       var chartEntry = [];
13
14       data.forEach(function (dataItem) {
15         switch(dataItem.categoryName) {
16           case 'Processor':
17             $scope.cpuData = dataItem.value;
18             chartEntry.push({ time: timestamp, y: dataItem.value });
19             console.log(chartEntry)
20             break;
21           case 'Memory':
22             $scope.currentRamNumber = dataItem.value;
23             break;
```

```

24         case 'Network In':
25             break;
26         case 'Network Out':
27             break;
28         case 'Disk Read Bytes/Sec':
29             break;
30         case 'Disk Write Bytes/Sec':
31             break;
32         default:
33             //default code block
34             break;
35     }
36 });
37 $scope.realtimeAreaFeed = chartEntry;
38 });
39 $scope.areaAxes = ['left', 'right', 'bottom'];
40 }
41 ]);

```

Our controller looks a bit more fleshed out. We have added a `realtimeAreaFeed` to the scope, which we'll bind to our view via the `ng-epoch` directive, and we have also added the `areaAxes` to the scope, which dictates the layout of the area chart.

Now let's add the directive to `index.html` and display the data coming in for CPU values:

```

1  <div class="row" ng-controller="PerformanceDataController">
2    <div class="panel-body" ng-controller="PerformanceDataController">
3
4      <epoch-live-area chart-class="category10"
5                      chart-height="200"
6                      chart-data="realtimeArea"
7                      chart-stream="realtimeAreaFeed"
8                      chart-axes="areaAxes">
9      </epoch-live-area>
10    </div>
11  </div>

```

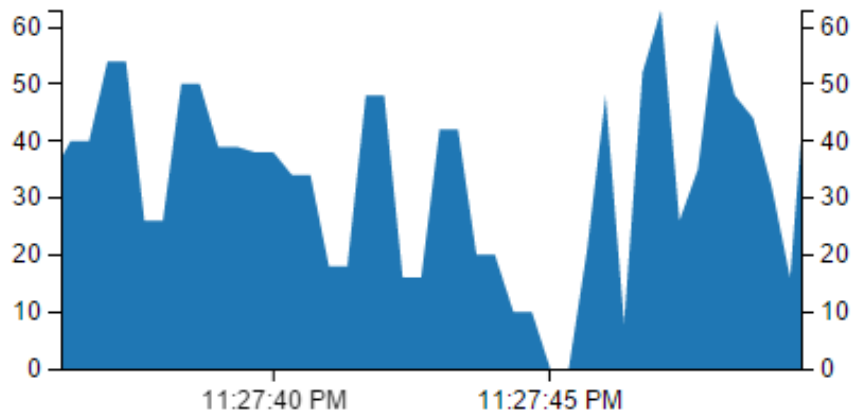
`chart-class` refers to the coloring scheme of D3.js, `chart-height` is what you suspect, and `chart-stream` is the data coming back from the SignalR server.

With that in place, we should see the chart come across in real time:

## Memory Performance

1073

## CPU Performance



Let's now wire up a whole bunch of data points to this chart, and add a whole other chart from the n3-pie framework (because who doesn't love pie!).

To add the pie chart from the n3-pie framework, simply add the following to our controller:

```
1 $scope.data = [  
2   { label: 'CPU', value: 78, color: '#d62728', suffix: '%' }  
3 ];
```

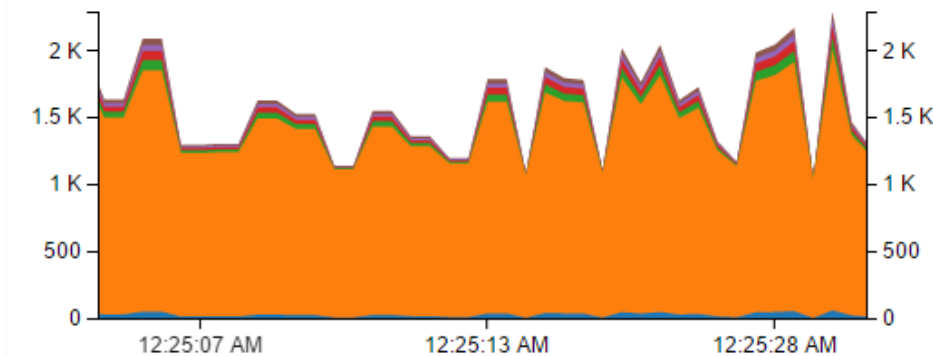
The `value`, of course, will be updated by the SignalR server. You can see this in [the full code for our controller \(https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient/blob/master/app/controllers.js\)](https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient/blob/master/app/controllers.js).

We should also take a moment to consider the [full code for our view \(https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient/blob/master/index.html\)](https://github.com/sitepoint-editors/SitePointTutorialSimpleAngularClient/blob/master/index.html).

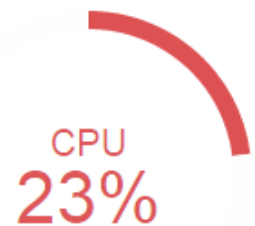
And we should be seeing the following data on screen:

1397

CPU Performance



CPU Performance Pie



We have seen that Angular can wire up to SignalR extremely easily – by simply plugging in the end point in an AngularJS service or factory. The AngularJS factory is an encapsulation mechanism to communicate with SignalR. Who knew that AngularJS and .NET would work so well together when “married up” (<http://louiebacaj.com/lets-marry-up-angular-to-net-2/>)?

## Core Aspects of the Server

I will go over a bit of the .NET code that allows this communication to happen on the back end. (You can find the source code [here \(https://github.com/sitepoint-editors/SitePointTutorialBackendServer\)](https://github.com/sitepoint-editors/SitePointTutorialBackendServer).)

To get started with building the server code first, you need to get SignalR running in your Visual Studio solution. To do this, simply follow the great tutorials over at [ASP.NET \(http://www.asp.net/signalr\)](http://www.asp.net/signalr) to get the base SignalR solution running. ([This \(http://www.asp.net/signalr/overview/getting-started/tutorial-getting-started-with-signalr\)](http://www.asp.net/signalr/overview/getting-started/tutorial-getting-started-with-signalr) is the simplest one.)

Once you have that up and running, change the C# Hub class to the following:

```

1 public class PerformanceHub : Hub
2 {
3     public void SendPerformance(IList<PerformanceModel> performanceModels)
4     {
5         Clients.All.broadcastPerformance(performanceModels);
6     }
7 
```

```
8     public void Heartbeat()
9     {
10         Clients.All.heartbeat();
11     }
12
13     public override Task OnConnected()
14     {
15         return (base.OnConnected());
16     }
17 }
```

---

Once you change the `Hub` class, Visual Studio will complain and you will need to add a performance model (this is automatically converted to JSON as it's pushed out by the server, thanks to `Json.NET`):

---

```
1     using System;
2     using System.Collections.Generic;
3     using System.Linq;
4     using System.Web;
5     using Newtonsoft.Json;
6
7     namespace SignalrWebService.Models
8     {
9         public class PerformanceModel
10        {
11            [JsonProperty("machineName")]
12            public string MachineName { get; set; }
13
14            [JsonProperty("categoryName")]
15            public string CategoryName { get; set; }
16
17            [JsonProperty("counterName")]
18            public string CounterName { get; set; }
19
20            [JsonProperty("instanceName")]
21            public string InstanceName { get; set; }
22
23            [JsonProperty("value")]
24            public double Value { get; set; }
25        }
26    }
```

---

The `JsonProperty` metadata is simply telling Json.NET to automatically convert the property name to lower case when converting to JSON for this model. JavaScript likes lower case.

Let's add a `PerformanceEngine` class, which pushes to anyone that will listen with real performance data. The engine sends these messages via SignalR to any listening clients on an asynchronous background thread.

Due to it's length, you can find [the code on our GitHub repo \(https://github.com/sitepoint-editors/SitePointTutorialBackendServer/blob/master/SignalrWebService/Performance/PerformanceEngine.cs\)](https://github.com/sitepoint-editors/SitePointTutorialBackendServer/blob/master/SignalrWebService/Performance/PerformanceEngine.cs).

This code basically pushes an array of performance metrics out to anyone that is subscribed in each `while` iteration. Those performance metrics are injected into the constructor. The speed of the push from the server is set on the constructor parameter `pollIntervalMillis`.

Note that this will work fine if you're hosting SignalR using [OWIN \(http://owin.org/\)](http://owin.org/) as a self host, and it should work fine if you're using a web worker.

The last thing to do, of course, is to start the background thread somewhere in your service `OnStart()` or in your `Startup` class.

---

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using Owin;
6  using System.Threading.Tasks;
7  using Microsoft.Owin;
8  using SignalrWebService.Performance;
9  using Microsoft.Owin.Cors;
10 using Microsoft.AspNet.SignalR;
11 using SignalrWebService.Models;
12
13 [assembly: OwinStartup(typeof(SignalrWebService.Startup))]
14
15 namespace SignalrWebService
16 {
17     public class Startup
18     {
19         public void Configuration(IAppBuilder app)
20         {
21             app.UseCors(CorsOptions.AllowAll);
```



```

22     var hubConfiguration = new HubConfiguration();
23     hubConfiguration.EnableDetailedErrors = true;
24     app.MapSignalR(hubConfiguration);
25
26
27     PerformanceEngine performanceEngine = new PerformanceEngine(800, GetRequiredPerformanceCounterNames);
28     Task.Factory.StartNew(async () => await performanceEngine.OnPerformanceMonitor());
29 }
30 }
31 }

```

The two lines that start the monitoring on the background thread (as I'm sure you've guessed) are those where we instantiate the `PerformanceEngine` and where we call the `OnPerformanceMonitor()`.

Now, I know you might be thinking that I'm randomizing the data from the server, and it's true. But to push real metrics, simply use the `System.Diagnostics` library and the `PerformanceCounter` provided by Windows. I am trying to keep this simple, but here is what that code would look like:

```

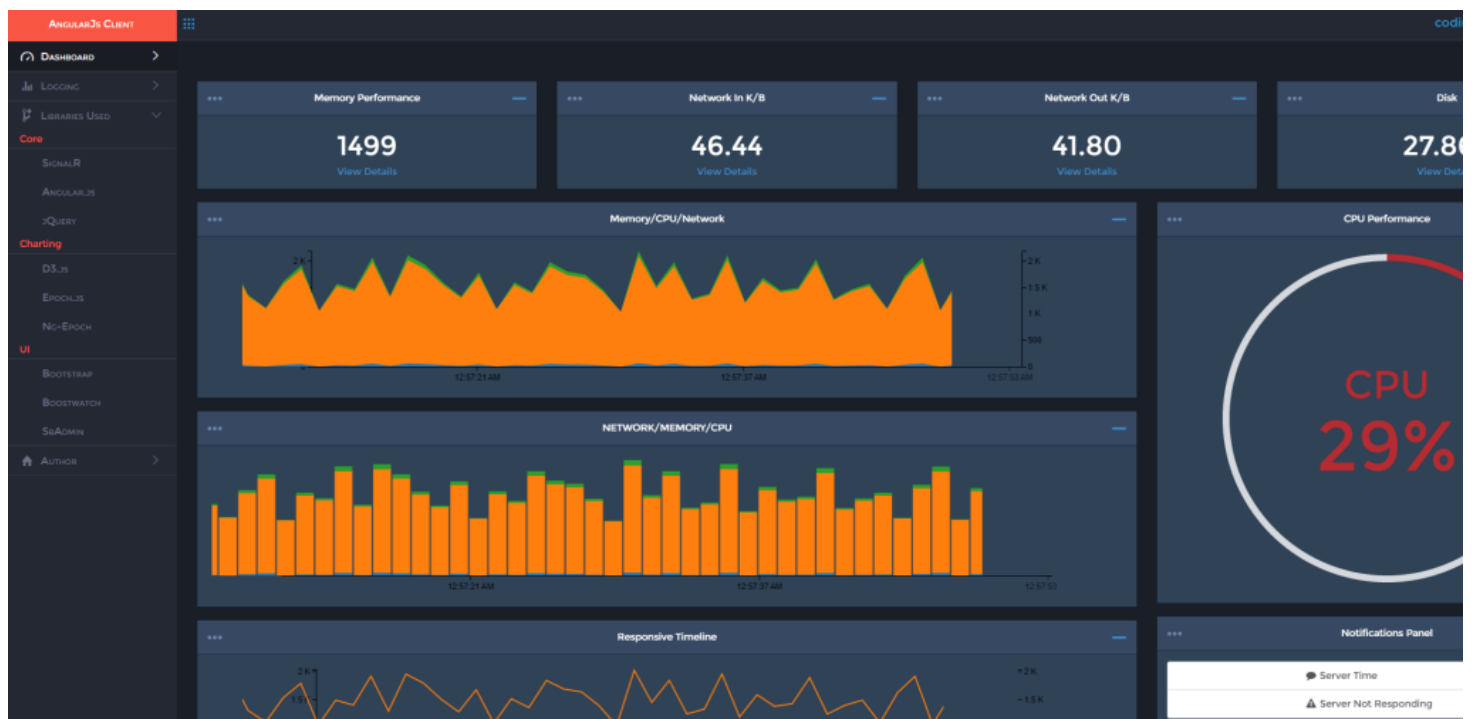
1  public static readonly IEnumerable<PerformanceCounter> ServiceCounters = new[]
2  {
3      //http://weblogs.thinktecture.com/ingo/2004/06/getting-the-current-process-your-own-cpu-usage/
4      new PerformanceCounter("Processor Information", "% Processor Time", "_Total"),
5      new PerformanceCounter("Memory", "Available MBytes"),
6      new PerformanceCounter("Process", "% Processor Time", GetCurrentProcessInstanceName(), true),
7      new PerformanceCounter("Process", "Working Set", GetCurrentProcessInstanceName(), true)
8  };

```

## Conclusion

We've seen how to consume SignalR data through Angular, and we've hooked that data up to real time charting frameworks on the Angular side.

A demo of the final version of the **client** can be seen [here](http://angularsignalrdashboardlouiebacaj.azurewebsites.net/) (<http://angularsignalrdashboardlouiebacaj.azurewebsites.net/>), and you can get the code from [here](https://github.com/sitepoint-editors/SitePointTutorialFinalAngularClient) (<https://github.com/sitepoint-editors/SitePointTutorialFinalAngularClient>).



A demo of the final version of the **server** can be seen [here](http://sitepointsignal.cloudapp.net/) (<http://sitepointsignal.cloudapp.net/>), and you can get the code from [here](https://github.com/sitepoint-editors/SitePointTutorialBackendServer) (<https://github.com/sitepoint-editors/SitePointTutorialBackendServer>).

# SignalR/Web API 2.0 service

This service pushes performance data out in real-time about itself and about the server which it is hosted in. Additionally a JavaScript proxy is generated so that the data can be easily consumed by a client application.

[Learn more over at LouieBacaj.com / Sitepoint.com »](#)

## Server Side Data

Performance Type	Performance Value
% Processor Time	10
Available MBytes	1131
% Processor Time	34
Available MBytes	1373
% Processor Time	14



I hope you've enjoyed this walk-through. If you've tried something similar, tell us about it in the comments!




[\(http://www.sitepoint.com/author/luljanbacaj/\)](http://www.sitepoint.com/author/luljanbacaj/)

[Louie Bacaj](#) (<http://www.sitepoint.com/author/luljanbacaj/>)

Louie Bacaj is a Senior Software Engineer with an M.S. in computer science; experienced scaling multi-million dollar software projects to several tier 1 financial institutions. Passionate about technology, fitness, and an active blogger over at <http://coding.fitness>

 (<https://twitter.com/lbacaj>)  (<https://github.com/lbacaj>)

 (<http://www.linkedin.com/in/luljanbacaj>)

 (<https://plus.google.com/101391588459459341360/posts>)

---

You might also like:

---

**Anatomy of a JavaScript MV\* Framework**  
**(<http://www.sitepoint.com/anatomy-javascript-mv-framework/>)**

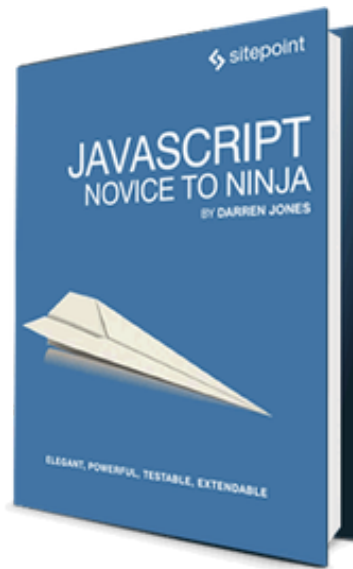


**Book: Jump Start JavaScript**  
**([https://learnable.com/books/jump-start-javascript?utm\\_source=sitepoint&utm\\_medium=related-items&utm\\_content=js-javascript](https://learnable.com/books/jump-start-javascript?utm_source=sitepoint&utm_medium=related-items&utm_content=js-javascript))**



**Creating Charting Directives Using AngularJS and D3.js**  
**(<http://www.sitepoint.com/creating-charting-directives-using-angularjs-d3-js/>)**





## Free *JavaScript: Novice to Ninja* Sample

Get a free 32-page chapter of *JavaScript: Novice to Ninja* and receive updates on exclusive offers from SitePoint.

**Claim Now**

## Comments

**Have Your Say (<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120>)**



**March 23, 2015** (<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120/2>)

**MrKing:** (<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120/2>)

signalr-  
dashboard-  
with-  
angularjs/116120/3)

Hello,  
Just confirm in your "Main Dependencies" section, you're n3-pie link was supposed to go <https://github.com/n3-charts/pie-chart> (<https://github.com/n3-charts/pie-chart>) instead it goes to <https://github.com/n3-charts/line-chart> (<https://github.com/n3-charts/line-chart>)

Thanks,

Jamie



March 23, 2015 (<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120/3>)

**Pullo: (<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120/3>)**

(<http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120/3>)

Hi,

please note that <https://github.com/n3-charts/pie-chart> (<https://github.com/n3-charts/pie-chart>) was the intended link



lbacaj:

because who doesn't love pie!

"Who doesn't love lines!" would have a bit of a different connotation.

Anyway, I corrected the link in the article.

Thank you for taking the time to point this out.

**[Have Your Say \(http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120\)](http://community.sitepoint.com/t/build-a-real-time-signalr-dashboard-with-angularjs/116120)**

About

[About us \(/about-us/\)](/about-us/)

[Advertise \(/advertising\)](#)

[Press Room \(/press\)](#)

[Legals \(/legals\)](#)

[Feedback \(mailto:feedback@sitepoint.com\)](mailto:feedback@sitepoint.com)

[Write for Us \(/write-for-us\)](#)

## **Our Sites**

[Learnable \(https://learnable.com\)](https://learnable.com)

[Reference \(http://reference.sitepoint.com\)](http://reference.sitepoint.com)

[Web Foundations \(/web-foundations/\)](#)

## **Connect**



[\(/feed\)](#) [\(/newsletter\)](#) [\(https://www.facebook.com/sitepoint\)](https://www.facebook.com/sitepoint)



[\(http://twitter.com/sitepointdotcom\)](http://twitter.com/sitepointdotcom)

[\(https://plus.google.com/+sitepoint\)](https://plus.google.com/+sitepoint)