# Real Time Applications with Web Sockets and SignalR

**Prepared for: Southwest Fox 2017**

By Rick Strahl

[weblog.west-wind.com](http://www.west-wind.com)

[rstrahl@west-wind.com](mailto:rstrahl@west-wind.com)

We’ve been using the Web for a long time now, but all those years we’ve been stuck in a world of basic **Transactional Request and Response** operations. The Web as we use it today is mostly based on **one to one** connections where the client makes a request and the server responds with some data in a transactional manner. Ask and you shall receive.

In recent years the **Real Time Web** has gained momentum predicated by the emergence of a credible **WebSockets** technology**,** that makes it possible to effectively create connected applications with reliable and fast messaging in real time. Although the technology requires a server in the middle for forwarding requests, essentially allows for build peer to peer applications where a client can to talk to one or more other client simultaneously via Push notifications.

In this article I cover a specific implementation of this technology using **Microsoft’s SignalR** and describe how you can take advantage of this technology in your FoxPro applications. It’s great technology to integrate disparate systems in a technology agnostic way, and another great way to extend the reach of your FoxPro applications with other technologies.

You can find the Samples and Source Code discussed in this article here:

* [**Source Code and Samples on BitBucket**](https://bitbucket.org/RickStrahl/swfox16_signalr)
* **Slides**

## The Web as we know it

Transactional Request and Response has gotten us a long way – almost everything we do today on the Web is based on this simple and relatively efficient paradigm. Static pages, dynamic content generated on the server, Web Services, REST and API services, even rich client applications – they all use this very same mechanism to request and receive data and it works just fine. For most typical use cases request/response works great.



### Limitations

In the traditional Web as we’ve known it the contract of communication is basically one to one: One client requests data to a server and the server returns a response back – on request – to the same client that requested the data. It turns out this linear approach to messages passing over the wire has a few limitations in certain scenarios.

### Long Running Requests

Long Running requests are one example. Have you ever run a long running request in a browser and sat there waiting for a response to come back while the browser appears to do nothing? We all have and that’s because the Request/Response cycle for this sort of operation is not very well suited. Because connections are one to one, the one client has to wait to receive a response.

There are some workarounds for this scenario, namely by using a polling mechanism. The client can submit a request to the server, and then sit there and keep asking “Are you done yet? Are you done yet? Are you done yet?”. It’s inelegant solution to a common problem, a problem that is a pain in the ass to implement in Web applications.

### One to One is no Fun

One to One connectivity makes it difficult to build truly distributed applications. Imagine you have an application where you have many clients that want to receive information as soon as it becomes available. With traditional Web applications this is very difficult to do, because the server effectively has no way to know who is connected. And even if it does – there’s no way to directly send a message back to the client because Web servers have exactly one connection that they can send data back to at a time and the connection is transitory – it lasts only as long as the request that’s executing.

Again, there are workarounds for this using polling where each client sits around and sends ping requests to the server to determine if there is something waiting for the client. As you can imagine this process is very inefficient.

### Lack of Server Push

Along the same lines it’s difficult to notify clients when some action occurs on the server. HTTP is basically a disconnected protocol where each request is made in its own context that is released at the end of the request. An HTTP connection is established and torn down with each and every request and there’s no lasting persistence or state between the client and server.

If you want to build an application where you want to have a Desktop application get notified when an order come in in your Web shop there’s no easy way to do this except perhaps again – using inefficient polling.

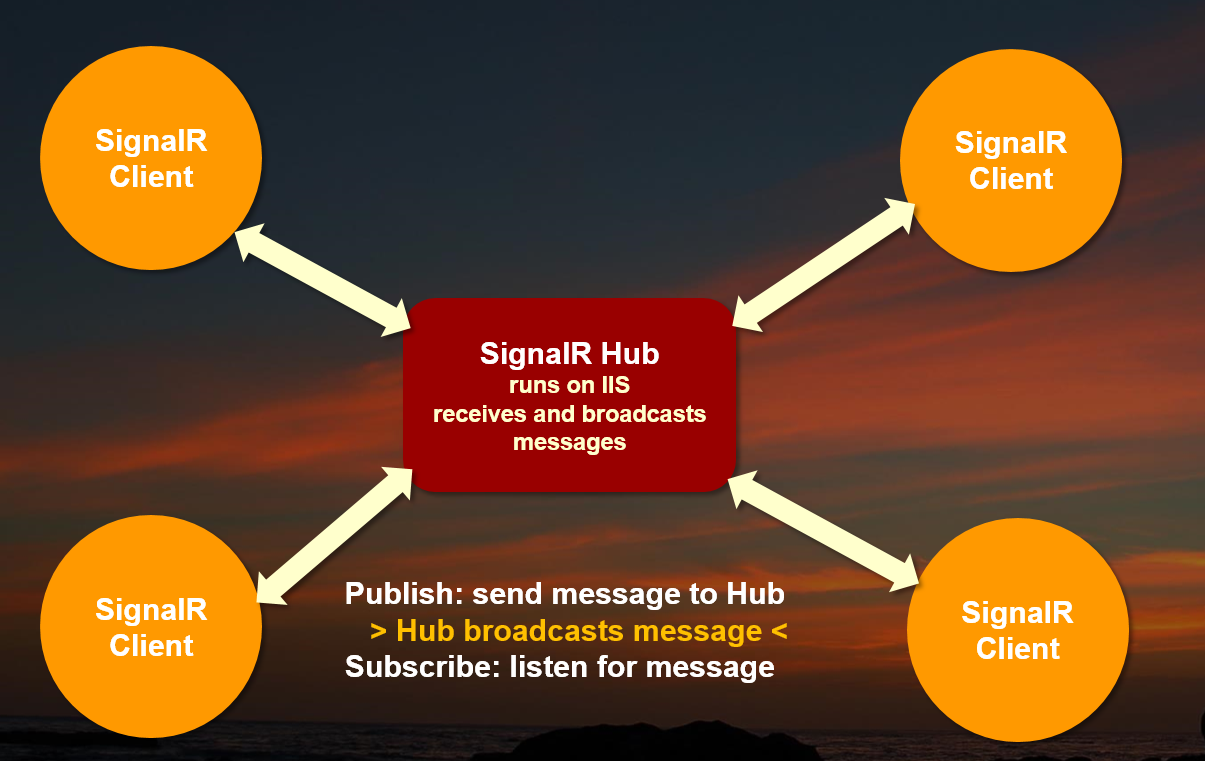
## The Real Time Web

Many of the issues discussed above are addressed by The Real Time Web, which became a thing around 2010 with introduction of WebSockets. WebSockets are based on a Hub and Spoke model where a central Web server acts a publishing warehouse for forwarding messages between Web Clients – in real time.

In the Microsoft world WebSockets – and a few fallback protocols – are provided by an ASP.NET based technology called SignalR. SignalR provides basic WebSocket/PubSub support with a very simple Publishing Hub model that makes it very easy to create applications that take advantage of this powerful programming model. In addition, SignalR also provides a full suite of clients for JavaScript, .NET, Java, Swift and many more so you can use SignalR on just about any client under the sun. For FoxPro you can interface to SignalR via .NET and that’s the focus of this article.

### Hub and Spoke

Real time applications typically use a Hub and Spoke model which looks something like this:



The main premise is that you have a central server through which all messages are sent (**Hub**). Clients (**Spokes**) can act both as a **Publisher** to send messages to other clients, as well as **Subscribers** that get notified on messages sent through the Hub from other clients or the server.

The server is connected to one or many clients (**Spokes**) and maintains persistant connections with these clients. Because there’s a permanent connection the server can **push** messages to each of the connected clients in **real time.**

### Some History

It’s somewhat ironic that the Web has taken so long to get to a point where this functionality is slowing becoming available to a more mainstream audience. Ironic because we’ve of course had TCP/IP sockets for nearly 50 years now that support this functionality.

The big problem with TCP/IP connections is that it’s a much more low-level protocol that requires a lot more legwork to get anything done with compared to HTTP. Additionally, TCP/IP requires custom ports in order to work as opposed to HTTP which is capable of tunneling all sorts of traffic over the common ports 80 and 443.

[WebSockets](https://en.wikipedia.org/wiki/WebSocket) is a newish protocol that was introduced in 2010 that provides a TCP/IP like Socket implementation over standard HTTP ports. The advantage of running over HTTP ports is that port 80 and 443 generally aren’t restricted and can be accessed without exposing additional ports in the network infrastructure. WebSockets are initiated through a plain HTTP connection by way of a connection upgrade process that then connects an actual socket between the HTTP client and server. This connection is kept open until the connection is explicitly closed by the client.

Web Sockets work in most common browsers and most development environments have some way to create Web Socket clients (and servers). On the Windows platform and for .NET/Desktop apps in particular SignalR is Microsoft’s wrapper implementation around WebSockets and a host of other mechanisms that allow real time peer to peer applications. SignalR supports a number of protocols in addition to WebSockets in order to support environments where WebSockets are not available. This means SignalR’s feature set is available even to legacy browsers and HTTP clients that have no native support for WebSockets.

## Real Time Web Features

WebSockets and Peer To Peer technology really enables a few interesting scenarios that are essentially based on event driven principles. Rather than requesting data and receiving it, data can be **pushed** out to one or more clients, which essentially is an asynchronous, **event driven** **Publish and Subscribe** approach.

The two killer features of this technology are:

* Server Push
* Peer to Peer
* Multi-Casting

### Server Push

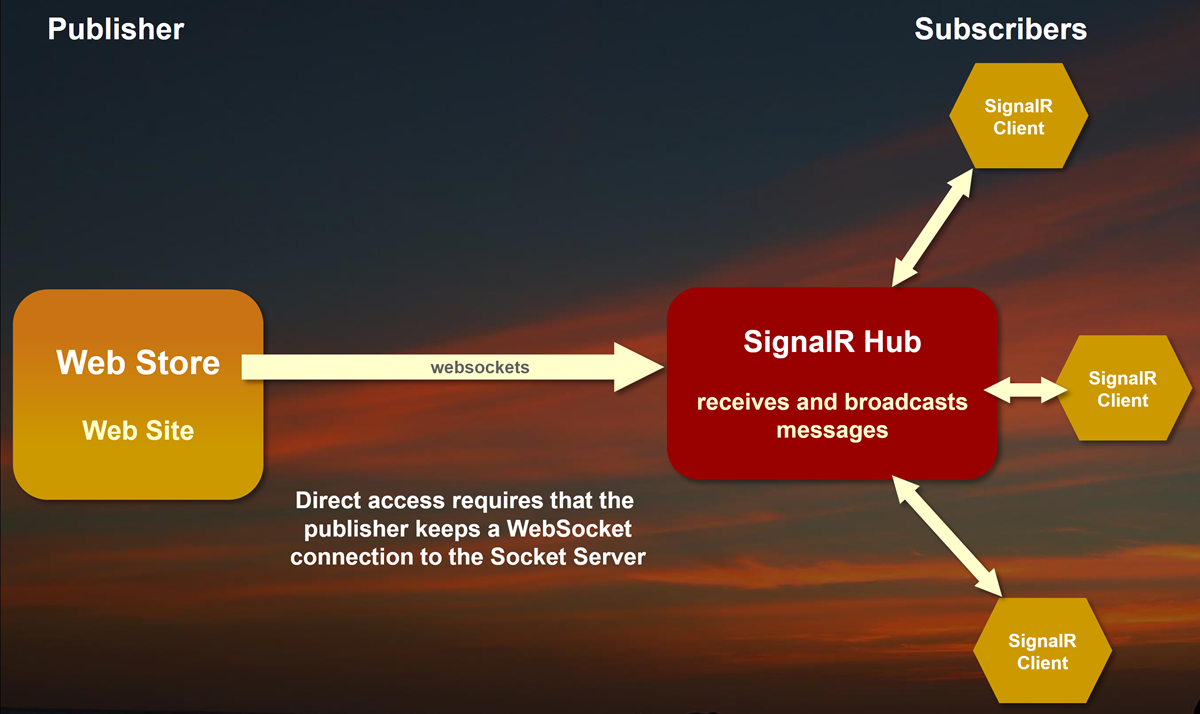
Server Push is the ability of the server to push messages to the client. This addresses a number of the shortcomings of standard HTTP that I discussed earlier. Rather than clients having to poll the server to check for status updates, the server can push that data directly to connected clients.

This is a critical piece for PubSub (publish and subscribe) scenarios where you have different actors that can publish information, and one or many clients that can subscribe and receive the published information. Server Push is really what enables all the rest of the features you gain by peer to peer technology based on a lasting connection that’s established over HTTP and kept alive.

Server push is used by many of today’s messaging solutions – you see it in tools like various messenger tools for mobile devices (iMessage, Facebook messenger, Skype, Twitter etc.). You also see it many Web mail clients that immediately notify you when a new message comes in. IOT devices also rely heavily on this technology to notify you of state changes in devices – typically there’s a background process that’s connected to the device and gets notified when there changes.

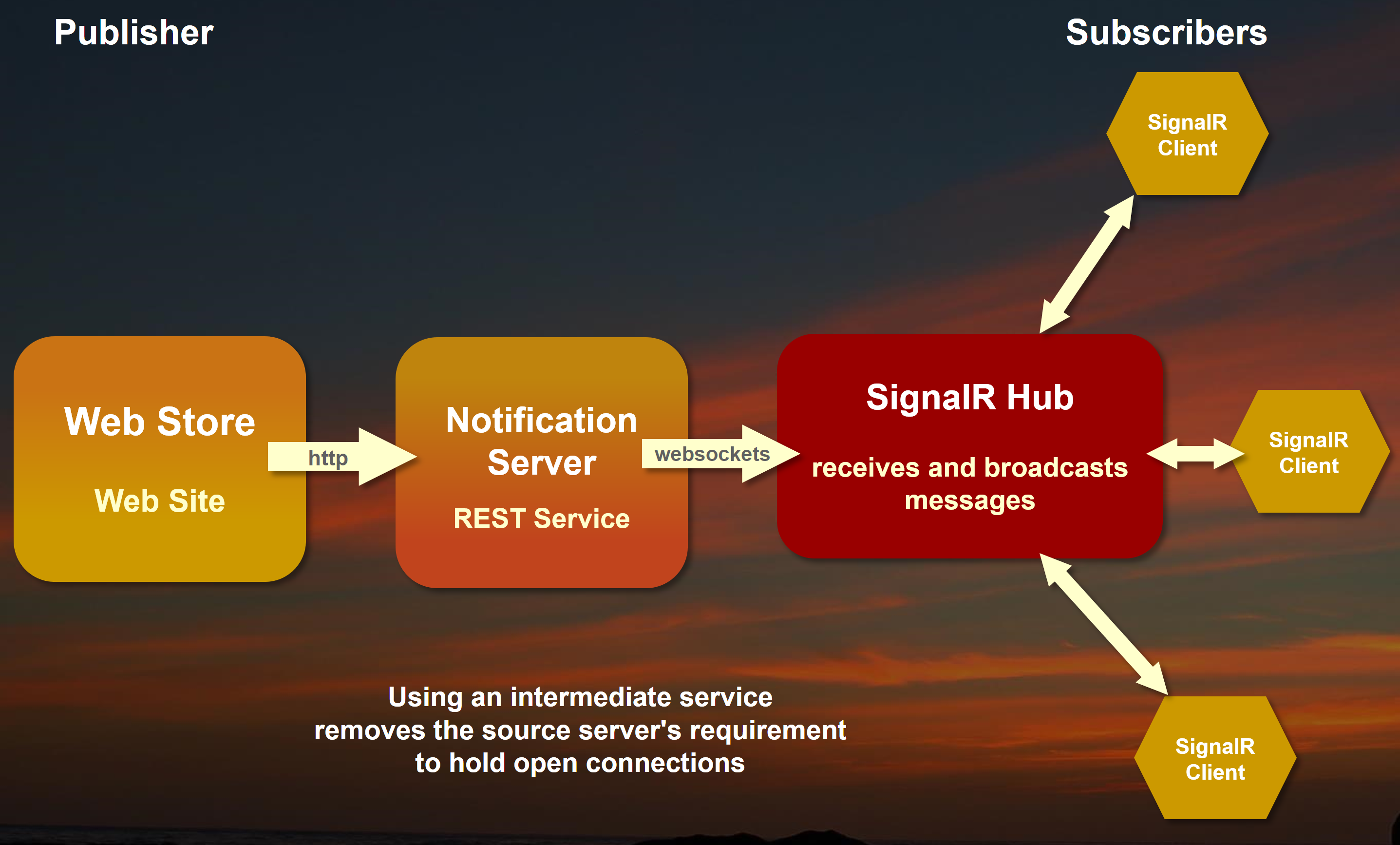
The cool thing about this technology is that you can have any kind of application be both the Publisher and Subscriber. It could be a Web page, or a FoxPro desktop application – it makes no difference as long as there’s a client that exists.

In the example below we have a Web application pushing a message to many clients, say when an order is placed. Any number of clients listening can be notified.



Server push is often complimented by an intermediate service used by Publishers that can be used to send notifications to using a simpler protocol like a plain HTTP REST call. Rather than having to keep a server-side connection open the notification process can often be performed by an intermediate service that in turn forwards messages to the real-time connection service.

For example, if you want to have a Web Store notify any applications listening of a new order that just came in, it’s easy for that store to post a simple REST service request to an intermediate server rather than maintaining a full WebSocket connection to all clients. The service then can forward that message to a dedicated WebSocket server that in turn can send out messages to the connected clients.



By using an intermediate server you can mitigate the permanent connection requirements and minimize the complexity introduced into the host application which now has to only make a simple HTTP call. The Notification server in the middle is the one that uses a WebSockets client and keeps the permanent connection open to the server. Rather than 100s of users keeping a connection open that may or may not be even used, the notification service only keeps one or very few connections open to push messages to all clients.

### Peer to Peer

SignalR’s implementation of WebSockets allows you to implement peer to peer applications where one client can talk to any other connected client. Although there’s a server in the middle to forward requests each client is essentially addressable as peer.

This is very powerful because by way of this server connection in the middle you can set up easy two-way communication between two machines. It doesn’t matter whether the two machines are clients or servers – they are both as they can both send and receive on the same connection.

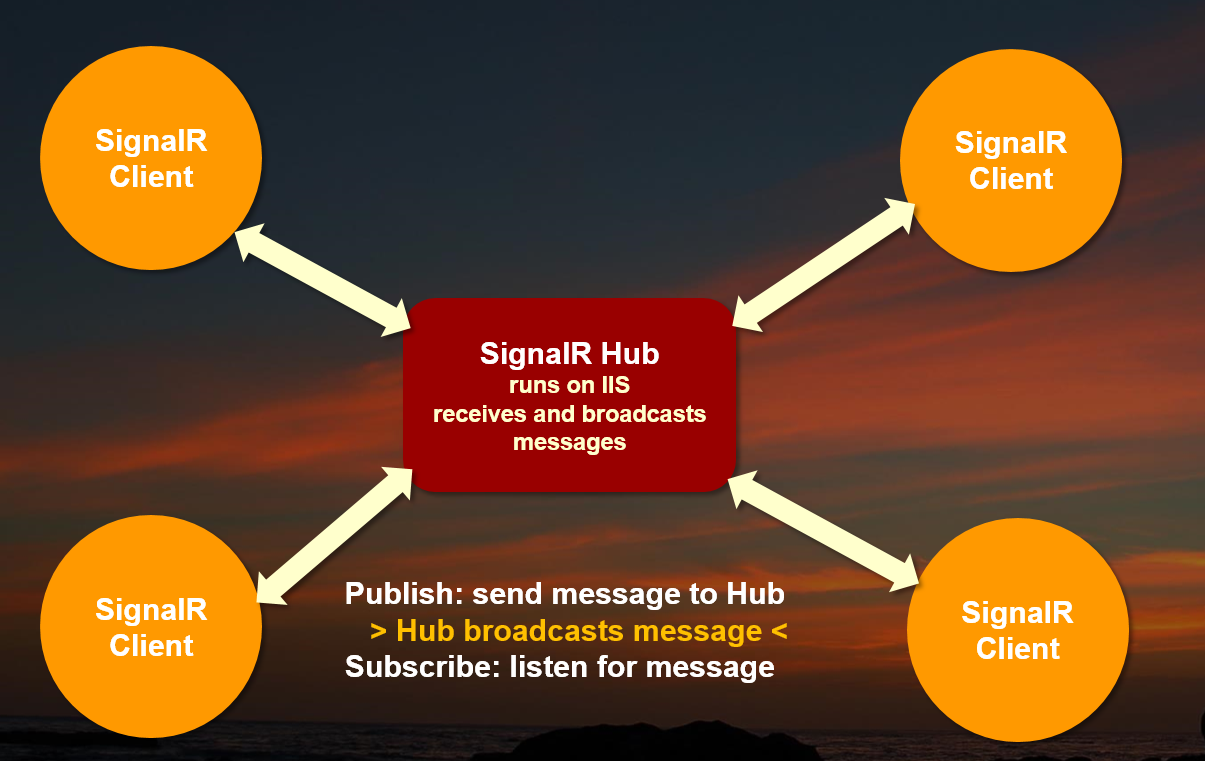
This has been possible via TCP/IP Sockets forever, but WebSockets makes this approach more palatable as it goes over standard ports that are allowed through firewalls and uses HTTP which can automatically handle mapping many users to the same IP address and port – something that is quite difficult to do at the application level with raw TCP/IP. WebSockets and SignalR in particular makes it very easy to set up a peer to peer application.

### Multi-Casting

Multi-casting is the ability to have one client publish a message and push it out to many other clients at the same time. It’s an important feature that allows many listeners to receive updated information virtually in real time. Think about a chat client where a message is sent by one user and immediately is seen by many others at the same time. Or your Twitter feed, when somebody posts a new message, that message is seen by anybody who’s subscribed to that user almost immediately.

The multi-casting works via a **Hub and Spoke** model which uses a **central server** (a **Hub** in SignalR speak) that acts as a store and forward service. All messages go through this server and the server distributes the message to all connected clients that fit the message’s target recipients. The server typically just forwards message, but it can also perform additional logic on each request. The server can also call back to the original client or a hub request can send out many separate messages as part of a request. For example the Hub might send a message to 5 connected clients, and return another message to the caller to notify that the messages have been sent to 5 people.

One nice feature of this model is that a client can both publish messages as well as receive them. So, for example in a Chat application you can have your client Send a new Chat message that is published to all clients. But the sender is also a client and so receives the message that was sent originally just like all other clients, so the same logic used by the other clients can be used to actually display the message.



SignalR (and other technologies like socket.io for JavaScript/Node) provide a software layer on top of the base connection layer to keep many connections to many clients open at the same time all over a single IP Address and port. This software layer manages specific connections to a particular endpoint and can send messages at any time to any of the connected clients.

## Creating SignalR Hubs and Clients

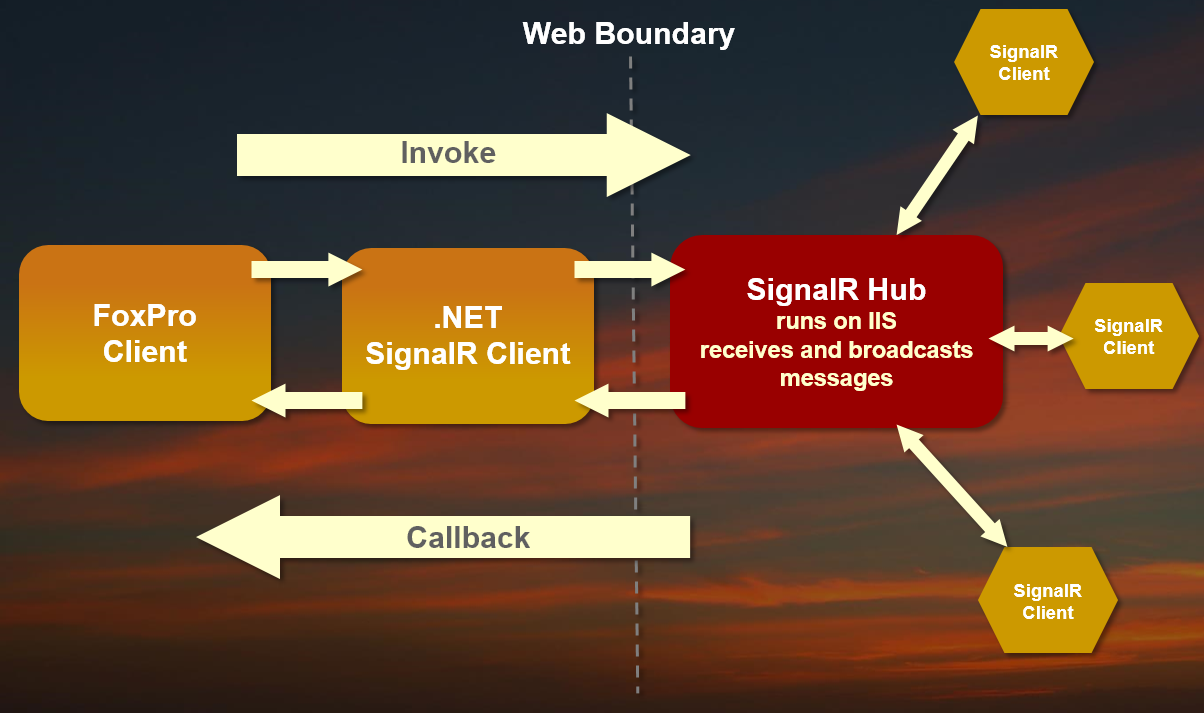
Ok let’s get down to the nuts and bolts of setting up a SignalR Hub and testing it in .NET first. I’ll use a Chat application as an example. Although very simple, a peer to peer chat application demonstrates most of the key features you need to understand with this technology.

First, in order to use this technology **you will need to write some .NET code**. It’s not a lot of code and it’s mostly very plumbing code, but it’s required in order to be able to get the Hub and client up and running since SignalR is at its core a .NET technology.

What we want to do for our FoxPro scenario is:

* Create a SignalR Hub in .NET
* Create a SignalR Client in .NET (Publish and Callback Handling)
* FoxPro Client that uses the .NET SignalR Client

With this setup, here’s what the message flow looks like:



Note that the clients can act both as publisher to send a message and a subscriber to receive messages.

In the context of a Chat application, you might have a publish operation like **SendMessage()** and a subscribe operation like **ReceiveMessage()**.

### Creating a .NET SignalR Hub

Let’s start by creating a new SignalR Hub.

* Open Visual Studio (2017 or later – [Community Edition is free](https://www.visualstudio.com/downloads/))
* Create a new Project and call it SignalRHub
* Choose ASP.NET Web Application (.NET Framework)
* Create an Empty Web application
* Add a NuGet Package (References/Manage NuGet Packages)
* Add **Westwind.Web** Package (for creating a single file REST Service)
* Use Add New **SignalR Hub Class (v2)** to the project
* Call it **ChatHub.cs**
* Use Add New **OWin Startup Class** to the project

Next, you’ll need to fix up the Startup class to load SignalR so it can find and serve your Hub.

using Microsoft.Owin;

using Owin;

[assembly: OwinStartup(typeof(SignalRHub.Startup))]

namespace SignalRHub

{

public class Startup

{

public void Configuration(IAppBuilder app)

{

var signalRConfig = new Microsoft.AspNet.SignalR.HubConfiguration()

{

EnableDetailedErrors = true,

};

**app.MapSignalR("/signalr",signalRConfig);**

}

}

}

This short block of code initializes SignalR and allows it to be served from a standard **/signalr** virtual. SignalR can automatically discover multiple Hub classes so this single URL is all you need even with multiple hubs.

Next let’s look at our ChatHub code and create an initial message.

public class ChatHub : Hub

{

private const string STR\_DEFAULT\_GROUP = "Southwest Fox";

public static Dictionary<string, ChatUser> Users = new Dictionary<string, ChatUser>();

public void SendMessage(string message, string group)

{

if (string.IsNullOrEmpty(group))

group = STR\_DEFAULT\_GROUP;

Users.TryGetValue(Context.ConnectionId, out ChatUser user);

if (user == null)

user = JoinGroup(null, group);

user.LastOn = DateTime.UtcNow;

var msg = new ChatMessage

{

Message = message,

User = user,

IsCurrentUser = user.Id == Context.ConnectionId

};

// Broadcast to a specific Group or (Clients.All)

Clients.Group(group).OnReceiveMessage(msg);

Debug.WriteLine("Server Send: " + user.Group + " " + user.Name);

}

public ChatUser JoinGroup(string name, string groupName)

{

if (string.IsNullOrEmpty(name))

name = StringUtils.RandomString(10);

Groups.Add(Context.ConnectionId, groupName);

Users[Context.ConnectionId] = new ChatUser {Name = name,

Group = groupName,

Id = Context.ConnectionId};

return Users[Context.ConnectionId];

}

public void ExitGroup(string groupName)

{

Groups.Remove(Context.ConnectionId, groupName);

if (Users.ContainsKey(Context.ConnectionId))

Users.Remove(Context.ConnectionId);

}

}

The initial hub is going to have 3 public methods that let us **send a message**, **join** and **exit a group**, which are the basics needed to create a simple chat client. This way we can have multiple areas that messages can be sent to.

In essence, these messages are little more than store and forward handlers that re-send the parameters of the hub method to a client handler on all connected clients. There’s a little bit of additional code to deal default values and tracking users, but most of the time hub methods are going to be fairly light on code just forwarding messages.

In this ChatHub the key method is **SendMessage()** which is used to broadcast a message from a client to any connected clients. Notice that this method is little more than a proxy which simply takes the inbound message, repackages it and sends it to all connected clients as a **OnReceiveMessage()**.

Clients.Group(group).OnReceiveMessage(

new ChatMessage

{

Message = message,

User = user

});

This call means: Broadcast a message to all clients that are in the group of name **group** and call the **OnReceiveMessage()** method on the client that is listening and pass the **ChatMessage**  object parameter to it.

#### Client Types

Above I’m using **Clients.Group()** to specify a specific group to send to. Groups are a very important and common way to ensure that messages reach only a specific subset of users, rather than everyone. Most applications need to group messages and these string based group designators give you all sorts of options for how you separate users.

The options available on Clients for target clients are:

* **All**Sends messages to all connected clients without any filters at all. This option is rarely used as you usually need to filter clients in some way.
* **Client** and **Clients**This allows you to specify specific clients to send to by connection id.
* **Caller**This is the current user that is making this call on the Hub. Use this if you need to make a 1 – 1 call between a client and server, which treats this request very much like a simple REST call.
* **Others**All clients except for the caller.
* **AllExcept()**Allows you to send messages tClo all users except those whose connection ids you exclude. Useful for excluding the current user typically.
* **Group(groupName, excludeIds)**Group lets you filter by an arbitrary string that you create as part of your application. Group is the most common way to filter clients. Clients can belong to multiple groups at the same time. Use **Client.Groups.Add()** and **Client.Groups.Remove()** in Hub code to add or remove users. Usually you need to add methods to explicitly let clients add themselved to groups.

As you can see there quite a few choices that should suit just about any scenario easily.

#### Message Sending via JSON

When the hub publishes a message, the message is encoded as JSON and sends this message to all connected clients that match the client filter (ie. a group name) and this causes the **OnReceiveMessage()** handler to be fired on every client that is currently connected and matches the filter.

This example uses a **ChatMessage** class and you’ll need two additional classes – **ChatMessage** and **ChatUser** – to make this work.

public class ChatMessage

{

public string Message { get; set; }

public ChatUser User { get; set; }

}

public class ChatUser

{

public string Name { get; set; }

public string Group { get; set; }

public DateTime LastOn { get; set; } = DateTime.UtcNow;

}

#### Group Management

Most application will need a way to add and remove users to groups. Rarely do you want to send messages to the **All** group, and groups provide a way to either segrate users by application specific groups or operational scope.

In the Chat application, groups are used to separate the different chat areas that messages are sent to – so each group can be thought of as a chat room.

The **JoinGroup()** and **RemoveGroup()** methods internally use a dictionary to keep track of users connected based on a connection ID so the user’s name doesn’t have to be sent with every message call. Instead the dictionary looks up the name by connection id. Each connection has a unique ID (a GUID) associated with it and I use that key to keep track of the user’s name and which group she’s currently associated with.

The user list is a static dictionary instance that lives in memory which is not persistent – when the application is shut down users are lost. In a real application, you’d store this data in a database or other persistent storage of some sort so it can survive a server restart/update. For our sample, this is sufficient.

Compile this code and make sure it builds.

There’s no easy way to test this Hub on its own – we need to build a client that can call the Hub. Let’s do that next.

### Creating a SignalR .NET Client

To access the server Hub, we have to use a SignalR client to connect to it. Signal has many clients available to it, including JavaScript, .NET, Java, Swift, Objective C etc.

For FoxPro, the best way to access a SignalR Hub is by using .NET to provide the SignalR client using wwDotnetBridge. In this step, I’ll create the .NET ChatClient class which provides the basic interface to publish and subscribe to messages through the Hub and receive result messages. I’ll start with the .NET SignalR client implementation.

To do this create another new project in the existing .NET Solution in Visual Studio we created earlier.

* Create a new Class Library Project
* Name it **SignalRClient**
* Add a new Class and call it **ChatClient.cs**
* Add a NuGet Packages for:
* Microsoft.AspNet.SignalR.Client

Modify ChatClient to something like this:

public class ChatClient : IDisposable

{

public string SignalRUrl { get; set; } = "http://localhost/signalrhub/";

public string CurrentGroup { get; set; }

public string CurrentName { get; set; }

// internal instances of the client and proxy

private HubConnection Server;

public IHubProxy Proxy;

// FoxPro instance

public dynamic Fox;

#region Lifetime management

public void Start(dynamic foxHandler)

{

// assign the FoxPro Class to call back to

Fox = foxHandler;

// Attach to server and specify name of the server Hub

Server = new HubConnection(SignalRUrl);

Proxy = Server.CreateHubProxy("ChatHub");

// Handle any messages the server lets clients handle

Proxy.On<ChatMessage>("OnReceiveMessage", OnReceiveMessage);

//Proxy.On("OnClientConnected", OnClientConnected);

// Start the connection

Server.Start().Wait();

}

/// <summary>

/// Shutdown the SignalR connection

/// </summary>

public void Stop()

{

Server?.Stop();

Server = null;

}

#endregion

#region Invoke Methods on the Proxy

public void SendMessage(string message, string group = null)

{

if (string.IsNullOrEmpty(message))

return;

Debug.WriteLine("Proxy Send: " + group);

Proxy.Invoke("sendmessage", message, group);

}

/// <summary>

/// Adds a user to a group. In this application user can only belong

/// to a single group but you can actually belong to many groups.

/// </summary>

/// <param name="userName"></param>

/// <param name="group"></param>

public void JoinGroup(string userName, string group)

{

if (!string.IsNullOrEmpty(CurrentGroup))

ExitGroup(CurrentGroup);

CurrentGroup = group;

CurrentName = userName;

Proxy.Invoke("joingroup", userName, group);

SendMessage(userName + " joined group " + group, group);

}

public void ExitGroup(string group)

{

CurrentGroup = null;

Proxy.Invoke("exitgroup", group);

}

#endregion

#region Handle callbacks from the Hub Server (ie. handle broadcasts)

public void OnReceiveMessage(ChatMessage message)

{

Debug.WriteLine("Client Receive: " + message.User.Group + " " + message.User.Name);

if (Fox != null)

Fox.ReceiveMessage(message);

}

#endregion

public void Dispose()

{

Stop();

}

}

Let’s break it down.

#### Starting and Stopping the Client

The **Start()** methods makes a connection to the Hub and keeps a connection open. This creates a Hub server connection and an instance of a proxy that the Hub can call methods on. Essentially you map proxy messages (like the OnReceiveMessage() call the Hub makes) to a method in the client.

Note that this class takes a parameter in the constructor that is an object that acts as a handler for Hub messages. In our case here this object is going to be a FoxPro COM object that is passed as a parameter and gets notified when a message is received.

Important: **The instance of the Client class must stay alive** in order to continue to have an open connection and be able to process messages from the hub. This means you have to ensure the Hub connection is stored on a persistent global or scoped object reference in your application. In FoxPro this means in a global variable or something like a FoxPro form for example.

The **Stop()** method simply cut the connection with the server.

#### Hub Invocation Methods

Next you should implement methods that let you **invoke Hub methods**. The ChatHub has three methods that push messages to the server and/or other clients.

**SendMessage()** sends a message to the Hub which forwards it to other clients as **OnReceiveMessage()**. The **JoinGroup()** and **ExitGroup()** methods only send a message to the Hub, which takes actions but doesn’t directly forward the call to connected clients. **JoinGroup()** internally calls **SendMessage()** to push out a new message into the Chat stream that a user has joined a chat room, but there’s no specific new message that is sent to clients.

In theory, you don’t need individual methods that do this mapping, but I like to map each hub proxy method to an explicit method on the client to have a clear interface on what’s available. While you can easily call the Proxy directly and use the ‘magic’ string values to map methods on the Hub, this requires you to remember the method names and parameters at all times. With a client wrapper Intellisense provides you the name of the method and correct parameter interface. If a method name changes on the server, a single method can be changed to adjust all client code. I highly recommend to do this extra step.

As to implementation of the Hub Methods on the client for the most part these are passthrough methods that just call the **Proxy.Invoke(methodName, parm1, parm2..parmN)**:

Proxy.Invoke("sendmessage", message, group);

All of these ‘method’ Ids are dynamically executed on the server – there’s no static binding. "sendmessage", here means call the SendMessage() method **on the Hub**.

#### Callback Handlers

Finally, we need to handle messages that the Hub sends to the client. Remember in the **Start()** method I mapped **Proxy.On()** hub messages to methods on the client. By convention I implement handler methods using **OnXXXX()** names, so it’s clear that these methods are not invoked directly but called by the Hub.

The ChatClient only has a single Handler method **OnReceiveMessage()** which receives a **ChatMessage** object parameter. Like the Hub server, we need to implement this class in order to get it deserialized as an object. In the client project create two classes:

public class ChatMessage

{

public string Message { get; set; }

public ChatUser User { get; set; }

}

public class ChatUser

{

public string Name { get; set; }

public string Group { get; set; }

public DateTime LastOn { get; set; } = DateTime.UtcNow;

}

Now when the Hub sends messages to clients, our client can capture the **ChatMessage** as an object.

#### Passing to FoxPro

The method really does nothing more than forward the method call to a handler object which in this case is ultimately supposed to be a FoxPro object:

if (Fox != null)

Fox.OnReceiveMessage(message);

This calls a method on the FoxPro object that was passed into the **Start()** method of the client. The object has to have a method **OnReceiveMessage()** and accept the single object parameter.

#### Testing the client in .NET

This step is not absolutely necessary but for me I prefer to first test the client in .NET and take advantage of .NET’s intellisense and test client to let me see things work first. From there I can create the FoxPro interface code.

To do this:

* Create a new **Unit Test Project** in our VS Solution
* Add a new **Unit Test Class**

Change the generated unit test class to look like this:

[TestClass]

public class ChatClientTest

{

//const string STR\_SignalRUrl = "http://signalrswf.west-wind.com/";

//const string STR\_SignalRUrl = "http://localhost:2662/"; // IIS Express

const string STR\_SignalRUrl = "http://localhost/signalRHub/"; // IIS

[TestMethod]

public void CallSendMessage()

{

var proxy = new ChatClient();

proxy.SignalRUrl = STR\_SignalRUrl;

// Simulate Fox object

var handler = new ChatHandler();

proxy.Start(handler);

// Join Group so we can see messages

proxy.JoinGroup("Rick", "Web Connection");

// this calls the server and calls back into the ChatHandler.ReceiveMessage()

proxy.SendMessage("First Message: " + StringUtils.RandomString(10), "Web Connection");

Thread.Sleep(1500);

}

}

**Web Server Note:**

This code uses local IIS to run the local samples to ensure that the Web Server is always available. You can also use IIS Express (*http://localhost:XXXX/*), but make sure you have IIS Express running when testing and using these samples. You can start IIS express by using *View in browser* on *index.html*, when IIS Express is set as the startup process. When started this way IIS Express stays running in the task tray where you can check for its running status.

This isn’t much code and it handles both sending a message to the hub and handling the response in a **ChatHandler** class (shown below). The code creates the ChatClient, ssets the URL and passes in the object that will handle the Hub callbacks via the **Start()** method.

I then join a group (forum) and send a message. Then the code sits and waits for a bit to allow the result message to be sent back to the handler.

Note that the client SendMessage() call and callback are asynchronous. This means that code continues executing after the SendMessage call, but we need to keep the client alive in order to receive the response message. Hence the **Sleep()** call to wait out the connection.

To simulate our FoxPro server in all this and the verify that the handler is called, I need to implement a ChatHandler class. The class doesn’t have to be a FoxPro or COM object – it just has to be an object that has an **OnReceiveMessage()** implementation:

public class ChatHandler

{

public void OnReceiveMessage(ChatMessage message)

{

string output = $@"{DateTime.Now:HH:mm:ss} [{message.User.Group}

{message.User.Name}] - {message.Message}";

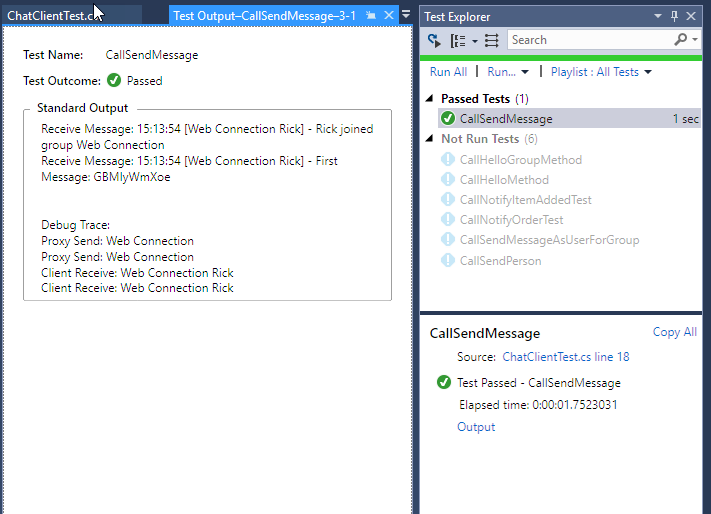
Console.WriteLine("Receive Message: " + output);

}

}

For the tests purpose I just write out some information from the message to the Console which we can see in the test runner.

You can run the test with the test runner. While in the method header press **Ctrl-R,T** or Context Menu *Run Tests* to run the test which gets you this in Visual Studio if all goes well:



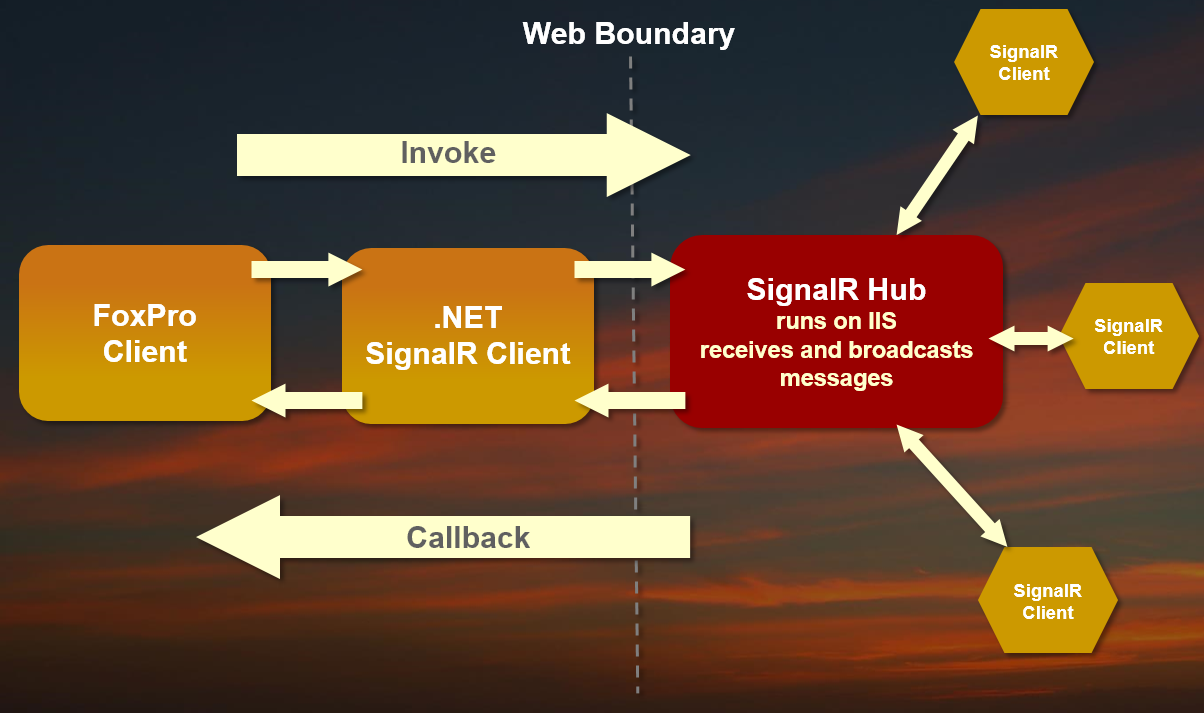
You should see two messages send – one for the join and one for the send message – which is what we want in this case. You can check out this and another test in the sample project for more detail.

### Creating a FoxPro Client

The .NET Test project is nice just to verify things work and have a reference with access to Intellisense that makes it easier to see how the client should be called. But it’s not required.

To create a FoxPro client I’m going to use [wwDotnetBridge](https://github.com/RickStrahl/wwDotnetBridge). wwDotnetBridge lets you call .NET objects from FoxPro without requiring that objects are registered with COM first. You can access objects directly and with the help of a number of support helpers you can access many features that COM interop on its own does not allow for from FoxPro.

As a refresher, here’s the work flow again of how FoxPro interacts with a SignalR Hub:



We have all the pieces in place at this point except the FoxPro client on the left. So lets set that up.

The FoxPro client essentially will be a wrapper around the .NET client so we want to implement **Start()** and **Stop()** methods, plus any of the

DO wwDotnetBridge

SET PROCEDURE TO ChatClient ADDITIVE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DEFINE CLASS ChatClient AS Custom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

oBridge = null

oChat = null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Init()

this.oBridge = GetwwDotnetBridge()

IF !this.oBridge.LoadAssembly( FULLPATH("SignalRClient.dll"))

ERROR "Couldn't load SignalR client library." + + this.oBridge.cErrorMsg

ENDIF

THIS.oChat = this.oBridge.CreateInstance("SignalRClient.Chat.ChatClient")

IF ISNULL(this.oChat)

ERROR "Unable to load Chat Client " + this.oBridge.cErrorMsg

ENDIF

ENDFUNC

\* Init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Start

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Start()

THIS.oChat.Start(this)

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Stop

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Stop()

this.oChat.Stop()

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SendMessage

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION SendMessage(lcMsg as String, lcGroup as string)

this.oChat.SendMessage(lcMsg, lcGroup)

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JoinGroup

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION JoinGroup(lcName, lcGroup)

this.oChat.JoinGroup(lcName, lcGroup)

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* OnReceiveMessage

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION OnReceiveMessage(loMessage)

ACTIVATE SCREEN

? loMessage.Message

? " - " + loMessage.User.Group + " " + loMessage.User.Name

ENDFUNC

ENDDEFINE

This should all look very familiar by now, right? This code basically just forwards all the methods of the .NET client into a FoxPro interface. The **Init()** method handles hooking up the .NET instance via wwDotnetBridge code that loads the .NET assembly (dll) and then loads an instance of the class.

All other methods except for **OnReceiveMessage()** simply forward calls to the .**oChat** instance of the .NET Client. **OnReceiveMessage()** is the callback method that gets called when the hub pushes messages out to clients and it receives a single **ChatMessage** parameter. For testing I’m writing out some values to the screen, but normally you are likely to leave this method blank and rather use **BINDEVENT()**  to forward the method call to a function in a form or other user interface action.

It’s a bit of busy body code, but with this in place we now have a very simple interface we can work with on a FoxPro form.

#### Testing this code in FoxPro

Just as I did in .NET, I can now write a simple test to see if the client works. I’m going to use a small PRG file to do so:

DO ChatClient

IF VARTYPE(loChat) = "O"

loChat.Stop()

WAIT WINDOW "" TIMEOUT 0.2

ENDIF

PUBLIC loChat

loChat = CREATEOBJECT("ChatClient")

loChat.Start()

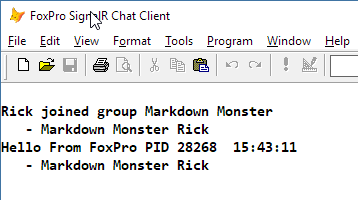
lcGroup = "Markdown Monster"

loChat.JoinGroup("Rick", lcGroup)

loChat.SendMessage("Hello From FoxPro PID " + TRANSFORM(\_vfp.ProcessId) + ;

" " + TIME(), lcGroup)

If all went well, you should now see two messages when you run this code:



To make this more interesting, try running multiple instances of FoxPro. As you do you’ll see messages from the other instances of FoxPro show up in each of these instances.

Even more interesting try adding this at the bottom of the file:

FOR lnX = 1 TO 100

goChat.SendMessage("Hello From FoxPro PID " + TRANSFORM(\_vfp.ProcessId) + " " + TIME(), lcGroup)

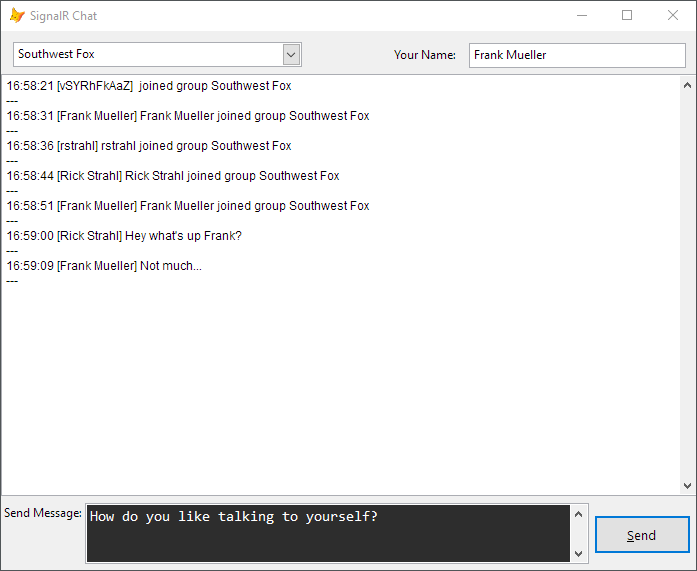
WAIT WINDOW "" TIMEOUT 0.1

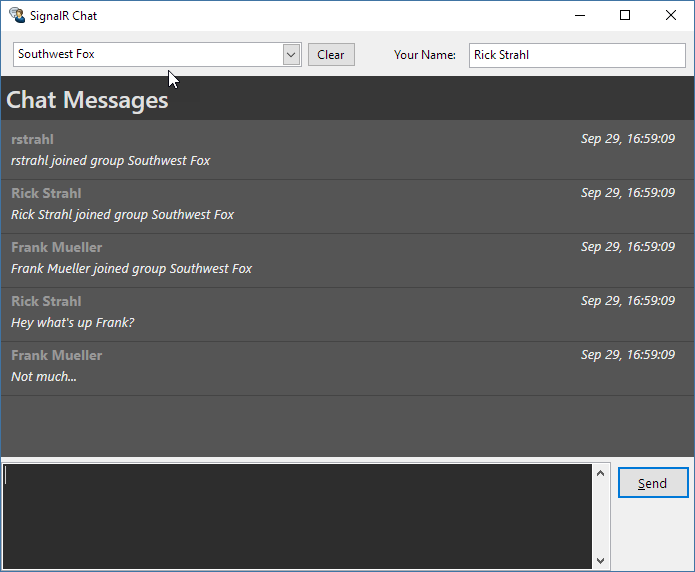
ENDFOR

Then run this on multiple instances simultaneously and you can start to get a feel of the ‘real time’ nature of this technology as messages fire in rapid fire succession in both instances of Visual FoxPro.

#### Creating a Chat Client Form

If you’ve made it this far, it’s time to do something more interesting so let’s add some UI to this to the chat client. The samples include two versions of this form – a simple version that uses a text box for message display and another version that uses a Web Browser control:



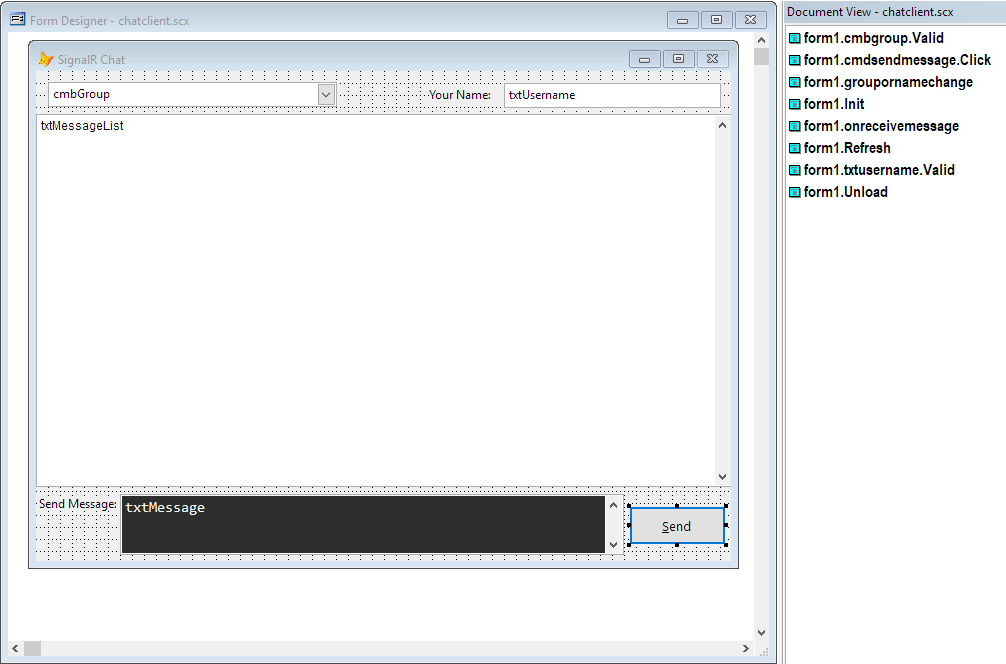


The latter of the forms has bit more logic in it that deals with the Web Browser display, so the former is a bit simpler to read and discuss here. You can check out either of them by running the sample application with **DO signalrsamples.prg.**

### Setting up

The first thing to do is create a new form and add the ChatClient to it. Create a form called ChatClient and add a property called **oChat** to it that will hold the chatclient instance for the duration of the form.

Set up the form to look something like this:



In the **Init**  of the form do this to initialize the ChatClient:

DO ChatClient

DO wwutils

IF TYPE("gcSignalRUrl") # "C"

PUBLIC gcSignalUrl

gcSignalRUrl = "http://signalrswf.west-wind.com/"

ENDIF

thisform.cmbGroup.RowSourceType = 1

thisform.cmbGroup.RowSource = "Foyer,Web Connection,Markown Monster,Southwest Fox"

thisform.cmbGroup.Value = "Southwest Fox"

thisform.txtMessageList.Value = ""

this.txtMessage.SetFocus()

**loChat = CREATEOBJECT("ChatClient")**

**thisform.oChat = loChat**

**BINDEVENT(thisform.oChat,"OnReceiveMessage",thisform,"OnReceiveMessage")**

**loChat.oChat.SignalRUrl = gcSignalRUrl**

**loChat.Start()**

**loChat.JoinGroup(this.txtUserName.Value, thisform.cmbGroup.Value)**

The key code is the last bit that creates the chat client, sets the SignalRUrl, starts the connection and then Joins the initial group which also immediately forces a message to be sent and displayed.

Note that I’m using **BINDEVENT()** to map the **OnReceiveMessage** handler from the ChatClient to the form’s **OnReceiveMessage** method. This allows me to write the message code inside of the context of the form rather than having to deal with form specific code inside of the generic class.

The two key operations related to signal are to send messages to any connected clients and to receive messages from any client that’s sending a new message.

#### Receiving Messages

To receive messages implement an **OnReceiveMessage()** method on the form and add this code:

LPARAMETERS loMessage

lcText = FormatValue(DATETIME(), "HH:mm:ss") + ' [' + loMessage.User.Name + '] ' + ;

loMessage.Message

lcList = this.cMessageList

lcList = lcList + lcText +;

CHR(13) + CHR(10) +;

"---" + CHR(13) + CHR(10)

IF LEN(lcList) > 10000

lcList = SUBSTR(lcList,5000)

ENDIF

THISFORM.cMessagelist = lcList

thisform.txtMessageList.Value = lcList

DoEvents

thisform.txtMessageList.SelStart = 20000

thisform.txtMessageList.SelLength = 1

DoEvents

The method receives an **loMessage** parameter of type **ChatMessage** which consists of the Message property and a User object property. In this example I simply take that value and write it out into a string and then append that to the editbox on the page. Note that this object is a COM object that is passed from .NET into FoxPro.

When the ChatClient is created we pass an instance of it to the .NET ChatClient and when a message comes in the .NET ChatClient checks to see whether an object was passed, and if it was tries to call the **OnReceiveMessage()** on the client. This is how the FoxPro **ChatClient.OnReceiveMessage()** method is called. The **BINDEVENTS()** code further forwards this to the form which now receives the .NET COM object that can access **loMessage.Message** and **loMessage.User.Name** for example.

#### Sending Messages

Sending messages is even easier since it’s essentially a simple method call on the ChatClient. Put the following into the **cmdSendMessage.Click()** handler:

**thisform.oChat.SendMessage(thisform.txtMessage.Value, TRIM(thisform.cmbGroup.Text))**

thisform.txtMessage.Value = ""

thisform.txtMessage.SetFocus()

Simple nothing to it right? But powerful – this simple line of code is pushed out to all connected clients – and to ourselves because we are also part of this group when the message is sent.

#### Changing Name and Group

Finally you can also change the group that you are listening on. The ChatClient is always connected to a single group at a time and you can call the **JoinGroup()** and **ExitGroup()** methods to get on or off a specific chat group.

Remember some of the extra logic I mentioned in the .NET Code in regards to group management and the mapping of users to groups – by doing that little bit of extra code I can now simply call this in **GroupOrNameChange()** which is hooked up to the **Valid()** of the **cmbGroup** and **txtUsername** controls:

thisform.oChat.JoinGroup(TRIM(thisform.txtUsername.Value), TRIM(thisform.cmbGroup.Value))

to join another group. This code removes me from the previous group and adds me to a new group.

Note, it is possible to belong to multiple groups at the same time so this logic was handled this way by choice. If you wanted to listen to multiple groups you can just join groups without leaving them which requires a small change in the logic in the .NET client.

The **BrowserChatClient** example, uses very similar code but adds some additional logic for passing messages into a live Web browser control. Overall the Initialization and send code is identical, but the **OnReceiveMessage** code is a little different and it uses rendering inside of the Web browser control to create the HTML layout.

### Multi-Casting Summary

The previous example primarily demonstrates the peer to peer features of this technology where a client sends a message to one or more other clients. This is extremely useful for certain kinds of distributed and connected applications, but it requires a specialized use case. If you’re building games or any sort of group activity that needs real time stats, then multi-casting is a great choice for those scenarios.

But, it’s kind of a very specialized scenario that’s useful only for very specific situations. If you find one that matches though multi-casting with SignalR can be a powerful feature that opens up functionality that previously would have been very difficult to implement with FoxPro. Now this is yet another potential feature you can handle with FoxPro.

## Push Notifications from Web Applications

Multi-casting is very cool and makes for the showcase scenario of SignalR. However, I’ve found the more realistic and more common use case for SignalR is the Push technology that allows a server application to notify a client application.

In the previous chat example, I highlighted the peer to peer or client to client features. In my work with this technology though I’ve found the most common use case to be one of bridging technologies. A common use case I see is this:

You have a Web site or Web site that performs transactional tasks, and when something happens on that site or service, you want to be notified **as soon as it happens**.

For example, you might have an e-commerce Web site and perhaps you want to have a running desktop application that your users are working with immediately be notified that an order was placed.

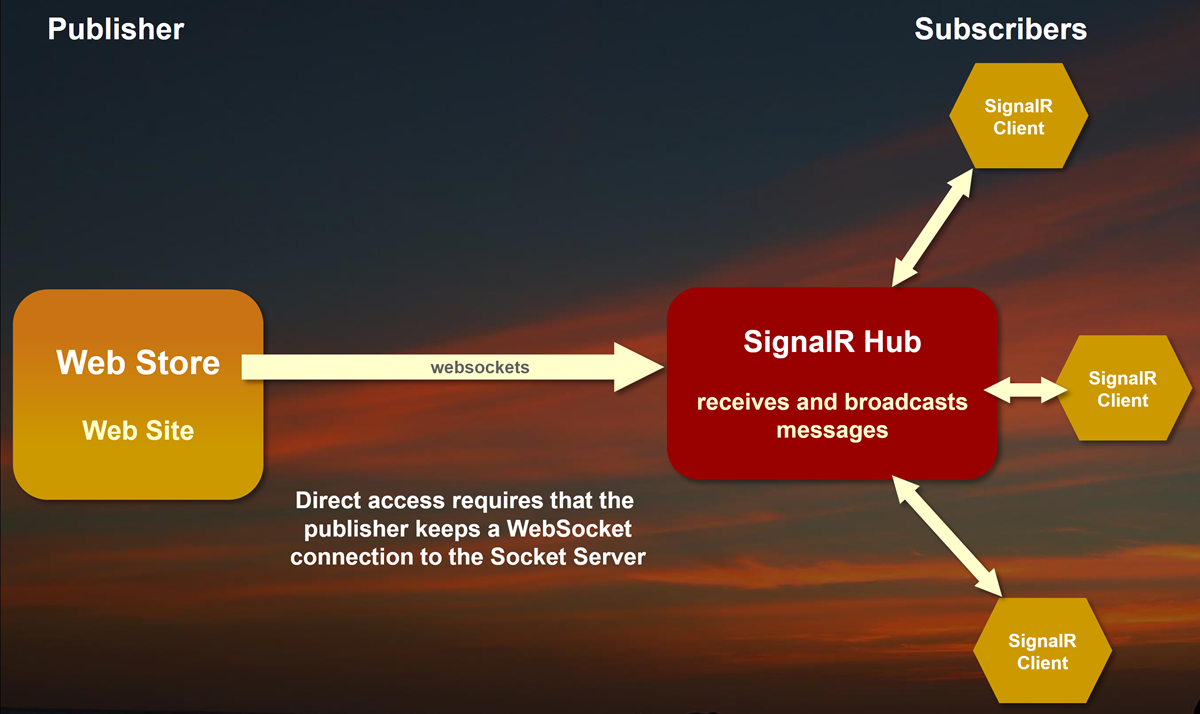
Or maybe you’re working in an IT department that’s been struggling with fraudulent Credit Card transactions and you want to be notified as soon as an order is in progress or when an order comes back with invalid credit card data.

### Web Store Orders and Items Added Sample

To demonstrate I’m going to set up another example that hooks into my online Web store and sends notifications when an item is added to the shopping cart or an order is placed.

Before we jump in and build this however, I want to discuss architecture in this scenario. At first blush you might think that setting the application so that SignalR runs inside of my Web Store application would be the way to go.

Like this:



In this scenario, the Web Store would directly integrate with the WebSockets technology and use SignalR directly to communicate with the SignalR Hub which in turn would push notifications out to any listening clients.

This would be possible in this case because I own the Web Store and I could directly integrate a SignalR client into the application.

However – this is really not such a great idea for several reasons.

* I have to own the application
* It has to be capable of running SignalR (not guaranteed)
* It requires that the Web Store uses WebSockets which are resource intensive

#### WebSocket Connections are Expensive

The latter point is actually very important – SignalR and WebSockets create permanent connections to the Web Server so WebSocket connections are what is considered resource expensive. A tied up connection is one less connection that the Web server can use to serve requests. While SignalR can handle thousands of simultaneous connections it’s much easier to overload a Site with a 1000 WebSocket connections than it is with a 1000 standard HTTP visitors.

#### SignalR Integration into a Host Application

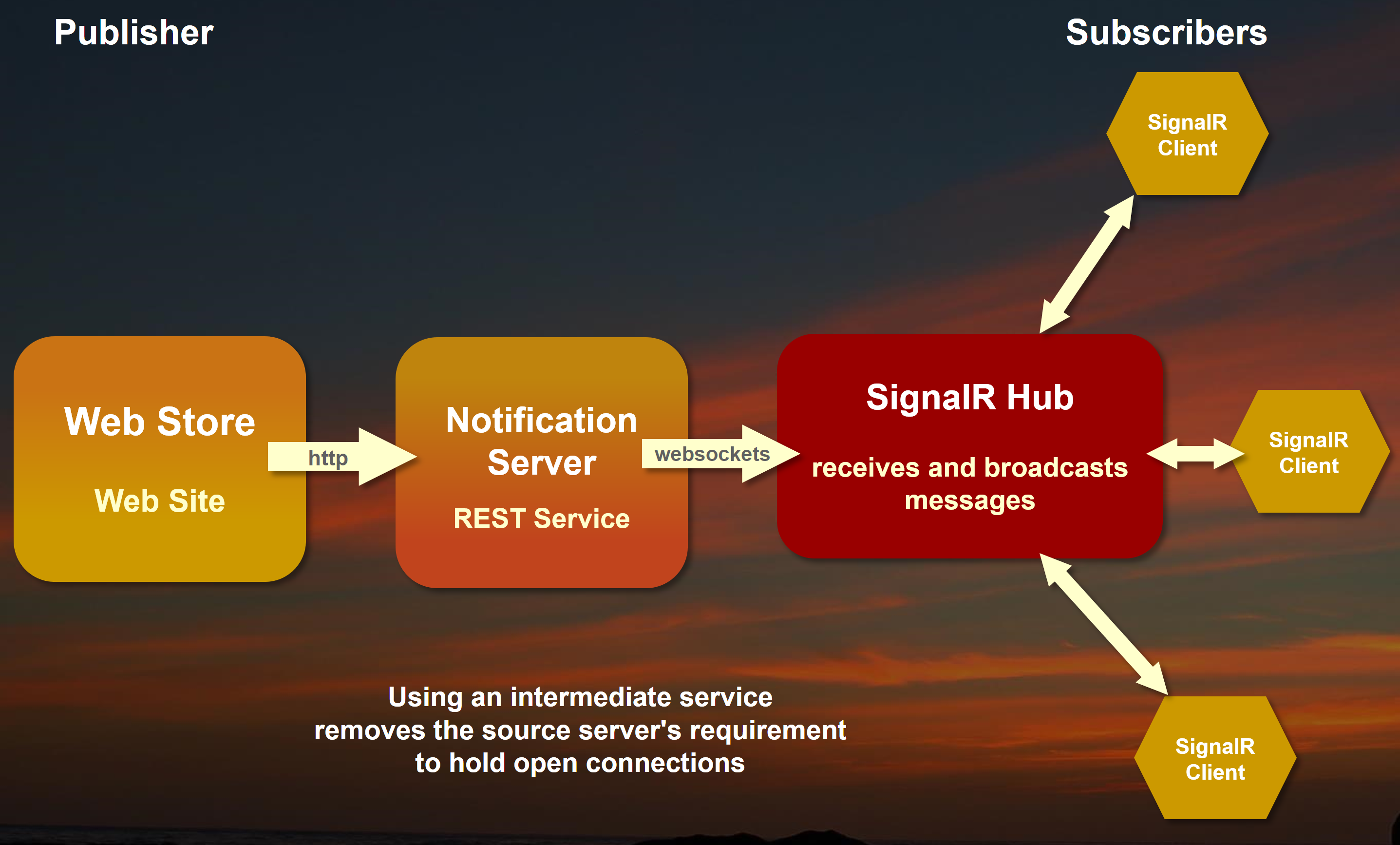
Integrating a SignalR application into an existing application may end up being a bit of a bolt-on scenario. Most likely the application will not have a good fit for running a SignalR client connection to a server and as mentioned above it requires a persistent connection which is not really all that easy to manage from a Web application since Web servers deal with connections somewhat differently.

#### A Better Approach: Using an intermediate Service

As it turns out when building notification gateways from one application to another, it’s ofen much cleaner to build a dedicated small service that sits between the SignalR hub and the Web application. The service can often sit on the same server or the same nework as the SignalR server and so provide a very efficient connection between the two.

But the biggest benefit is that the host pplication only has to make a simple HTTP call to push its data out to be published rather than having to deal with the implementation details of SignalR in an existing application.

Here’s what that looks like:



This is more realistic for most real world scenarios because it allows the host service on the left to do nothing more than send an HTTP request which is low overhead and doesn’t add any special knowledge to the original application. And… you can do that pretty easily regardless of whether the application was written in .NET, Java, Node, or PHP.

I highly recommend taking the above approach for server to client or server to server messaging.

### Implementing a Web Store Notification System

I’ll use my online store at <https://store.west-wind.com> as an example here, and hook up two notifications on the site:

* Notification when an item is added to the shopping cart
* Notification when an order is placed

To do this I’ll quickly run through creating a SignalR hub, and .NET Client just as we did in the last example. I’ll then create a very simple HTTP REST service that can take a JSON input that calls into the SignalR Hub and pushes messages out to a client. I then build a very simple FoxPro client to capture the messages and display them on a form.

Most of this is nearly identical to what we did earlier with the chat example so I’m not going to explain much except where there are difference but I’ll show the code for all of it. I’m just going to add all the functionality to the existing Solution that I created earlier.

Let’s start again with the Hub:

public class WebStoreNotificationHub : Hub

{

public void NotifyOrder(WebStoreOrderNotification notification)

{

Clients.All.OnNotifyOrder(notification);

}

public void NotifyItemAdded(WebStoreItemAddedNotification notification)

{

Clients.All.OnNotifyItemAdded(notification);

}

}

This hub is even simpler than the previous one in that it only has two methods that act as passthrough. This example doesn’t need to know about groups so it simply sends out messages to **Clients.All**.

We’ll also need classes for the DTOs that hold the message data:

public class WebStoreOrderNotification

{

public string OrderNumber { get; set; }

public decimal OrderAmount { get; set; }

public DateTime OrderDate { get; set; }

public string CustomerName { get; set; }

public int CustomerId { get; set; }

}

public class WebStoreItemAddedNotification

{

public string Sku { get; set; }

public string Description { get; set; }

public decimal Qty { get; set; }

public decimal Price { get; set; }

public decimal Discount { get; set; }

}

Next is the WebStoreNotifications .NET Client:

public class WebStoreNotificationsClient : IDisposable

{

public string SignalRUrl { get; set; } = "http://localhost/signalrhub/";

private HubConnection Server;

public IHubProxy Proxy;

// FoxPro instance

public dynamic Fox;

#region Lifetime management

public void Start(dynamic foxHandler)

{

Server = new HubConnection(SignalRUrl);

// Specify the name of server Hub Class

Proxy = Server.CreateHubProxy("WebStoreNotificationHub");

Proxy.On<WebStoreOrderNotification>("OnNotifyOrder", OnNotifyOrder);

Proxy.On<WebStoreItemAddedNotification>("OnNotifyItemAdded", OnNotifyItemAdded);

Server.Start().Wait();

Fox = foxHandler;

}

/// <summary>

/// Shutdown the SignalR connection

/// </summary>

public void Stop()

{

Server?.Stop();

Server = null;

}

public void Dispose()

{

Stop();

}

#endregion

#region Publish methods

public void NotifyOrder(WebStoreOrderNotification notification)

{

Proxy.Invoke("NotifyOrder", notification);

}

public void NotifyItemAdded(WebStoreItemAddedNotification notification)

{

Proxy.Invoke("NotifyItemAdded", notification);

}

#endregion

#region Subscription Handler Methods

private void OnNotifyOrder(WebStoreOrderNotification notification)

{

Debug.WriteLine("OnOrderReceived Client called: " + notification);

if (Fox != null)

Fox.OnNotifyOrder(notification);

}

private void OnNotifyItemAdded(WebStoreItemAddedNotification notification)

{

Debug.WriteLine("OnItemAdded Client called: " + notification);

if (Fox != null)

Fox.OnNotifyItemAdded(notification);

}

#endregion

}

This code implements Start and Stop with start accepting a handler object as input on which the proxy methods are called. The Hub allows for **NotifyOrder** and **NotifyItemAdded** methods to be fired so those two methods are implemented which simply forward to the hub. I actually won’t be using those methods, but I added them to the client anyway in case the service is actually called through SignalR. More on this in a minute.

The Hub publishes two messages **OnNotifyOrder** and **OnNotifyItemAdded** so there are handler methods which forward these messages to the provided handler object – eventually our FoxPro client that will display this information on a form.

#### Creating an ASP.NET REST Service the Easy Way

With the client in place I’m going to create a REST service. To do this I’ll use one of my own tools from the[Westwind.Web toolkit](https://www.west-wind.com/WestwindToolkit/). The [Callback Handler](https://west-wind.com/westwindtoolkit/docs/_3rb0st94w.htm) class allows you to simply drop a class into an ASP.NET Web project and add methods with a **[CallbackHandler]** attribute on it to create a REST service.

To do this:

* Go to the SignalRHub Web Project in the Solution
* Add a new Class and call it **WebStoreOrderService.cs**

In that source file implement the following classes:

/// <summary>

/// REST API Service that can be called with a simple HTTP

/// POST operation and JSON content from a client application.

/// </summary>

public class WebStoreOrderService : CallbackHandler

{

[CallbackMethod(RouteUrl="api/WebStore/OrderNotification")]

public WebStoreOrderNotification NotifyOrder(WebStoreOrderNotification notification)

{

// statically access a hub locally - rather than creating a client

var context = GlobalHost.ConnectionManager.GetHubContext<WebStoreNotificationHub>();

context.Clients.All.OnNotifyOrder(notification);

return notification;

}

[CallbackMethod(RouteUrl = "api/WebStore/ItemAddedNotification")]

public WebStoreItemAddedNotification NotifyItemAdded(WebStoreItemAddedNotification notification)

{

// statically access a hub locally - rather than creating a client

var context = GlobalHost.ConnectionManager.GetHubContext<WebStoreNotificationHub>();

context.Clients.All.OnNotifyItemAdded(notification);

return notification;

}

}

Simple right? The key is the **CallbackHandler** base class and the **[CallbackHandler]** attribute on methods that you want to be reachable via HTTP. You can specify a RouteURL which specifies a site relative path.

Both methods directly connect to the SignalR runtime by using

GlobalHost.ConnectionManager.GetHubContext<WebStoreNotificationHub>();

To get access to a Hub and calling a method on it. This bypasses SignalR/WebSockets and just directly invokes the methods as if they were called externally. This is very efficient and avoids the additional overhead.

If the above service was not running on the same site, you’d then have to use the client we created in the last step to make an actual SignalR call in the service. No big deal, but in this scenario the direct call is much more efficient.

There’s one more thing that needs to be configured for the server in the **Startup.cs** file of the site:

**CallbackHandlerRouteHandler.RegisterRoutes<WebStoreOrderService>(RouteTable.Routes);**

This enables the RouteUrls on the Callback methods. And that’s it for the service.

I also created a .NET Test Project for these requests. You can find those in the sample project.

### Creating the FoxPro Service

Now it’s time to build the FoxPro client. As before the Fox client pretty much wraps the .NET client methods in FoxPro code.

DO wwDotnetBridge

SET PROCEDURE TO WebStoreNotificationsClient ADDITIVE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DEFINE CLASS WebStoreNotificationsClient AS Custom

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

oBridge = null

oNotifications = null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Init()

this.oBridge = GetwwDotnetBridge()

IF !this.oBridge.LoadAssembly( FULLPATH("SignalRClient.dll"))

ERROR "Couldn't load SignalR client library." + + this.oBridge.cErrorMsg

ENDIF

THIS.oNotifications = this.oBridge.CreateInstance("SignalRClient.WebStoreNotifications.WebStoreNotificationsClient")

IF ISNULL(this.oNotifications)

ERROR "Unable to load Notifications Client " + this.oBridge.cErrorMsg

ENDIF

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Start

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Start()

THIS.oNotifications.Start(this)

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Stop

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION Stop()

this.oNotifications.Stop()

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* OnNotifyOrder

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION OnNotifyOrder(loNotification)

ACTIVATE SCREEN

? loNotification.OrderNumber + " " + TRANSFORM(loNotification.OrderAmount) + " " + NVL(loNotification.CustomerName,"")

ENDFUNC

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* OnNotifyItemAdded

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION OnNotifyItemAdded(loNotification)

ACTIVATE SCREEN

? loNotification.Sku + " " + loNotification.Description + " " + TRANSFORM(loNotification.Qty)

ENDFUNC

ENDDEFINE

The code should be pretty self explanatory – it just forwards calls to the .NET client, except for **OnNotifyOrder** and **OnNotifyItemAdded** which just echo some text to the screen.

To test this out you can now call this client with the following code:

DO WebStoreNotificationsClient

IF VARTYPE(loNotifications) = "O"

loNotifications.Stop()

WAIT WINDOW "" TIMEOUT 0.2

ENDIF

PUBLIC loNotifications

loNotifications = CREATEOBJECT("WebStoreNotificationsClient")

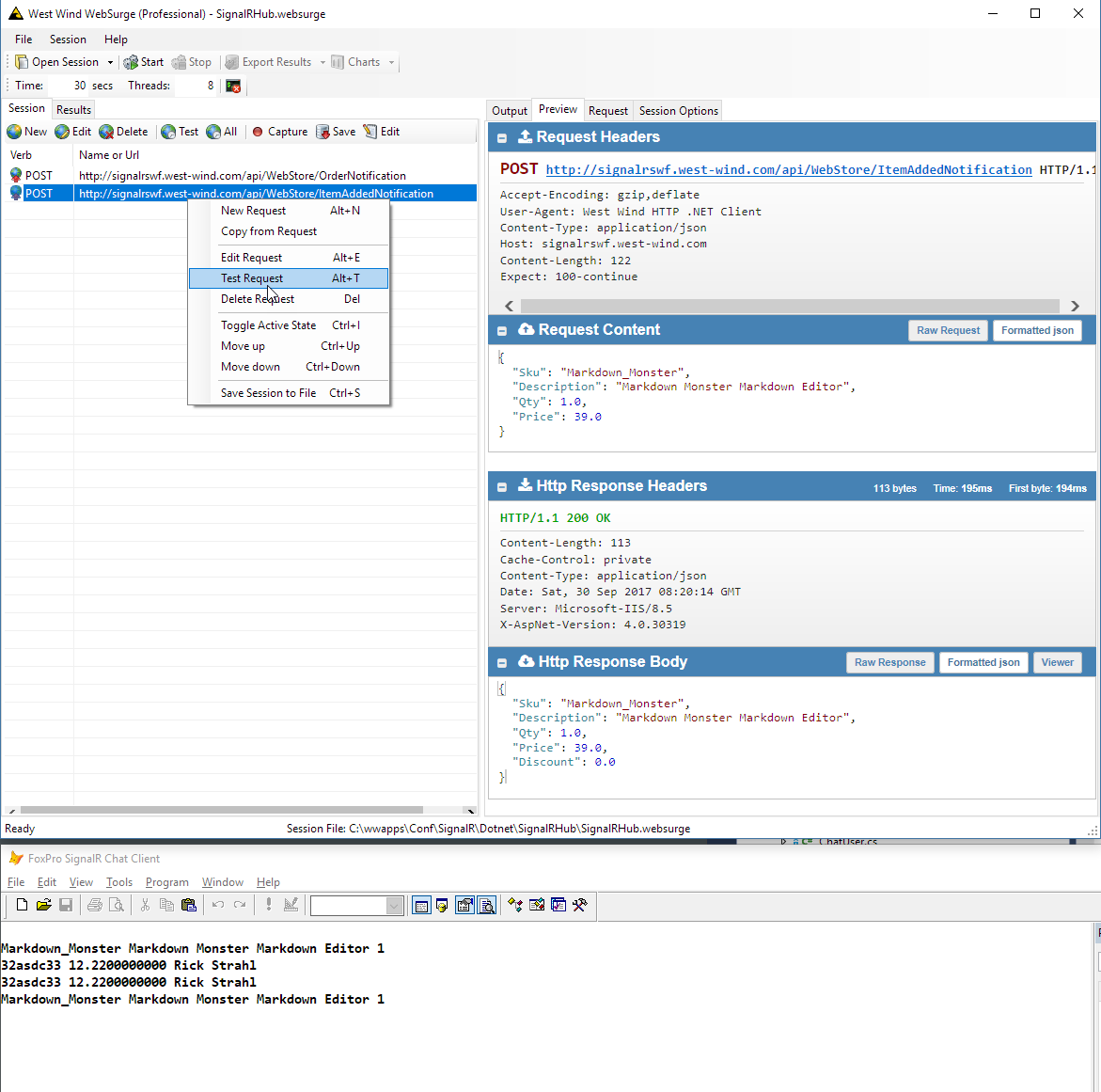
loNotifications.Start()

WAIT WINDOW "Waiting for orders to be placed..."

Ah – but this code won’t do anything – other than wait for something to happen 😊

So let’s make something happen. To test we should now be able to send a message to the Web Service via HTTP. To do this I’m going to use a visual client [West Wind WebSurge](https://websurge.west-wind.com) which lets me simulate Web requests. You can use other tools like Postman or Fiddler, or CURL or wwHttp (in another FoxPro instance!).

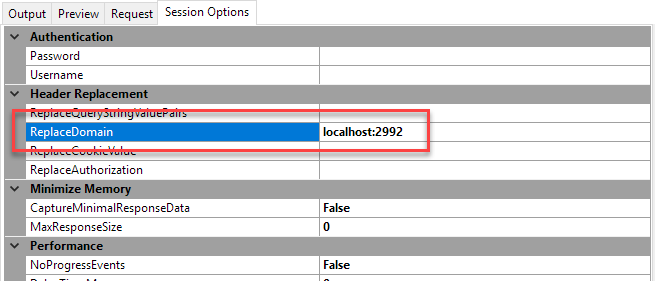
If all is well you should see something like this:



What’s happening here is:

* I’m hitting the Web Service on my Web site
* It’s sending a message to my connected client
* Which displays the order info

The above example uses the live sample. To check the local site the URL should be changed to <http://localhost/signalRHub> or <http://localhost:2992/> as the base url. In WebSurge you can easily change the domain/virtual by using the Domain override:



This allows you to test with a local server.

### Integrating into my Web Store

So this works fine – I know I can hit the Web Service with a JSON order or line item object and it will hit the connected clients. Now I need to integrate this into my Web Store application on my Web site that’s running .NET.

To do this I’m going to first create a class that handles the HTTP calls like this:

public class WebStoreNotifications

{

//private static string NotificationUrl = "http://localhost/signalrhub/";

private const string NotificationUrl = "http://signalrswf.west-wind.com/";

public static void OrderNotification(busInvoice invoice)

{

var cust = invoice.Customer.Entity;

var inv = invoice.Entity;

var request = new WebStoreOrderNotification

{

CustomerId = cust.Pk,

CustomerName = cust.Firstname + " " + cust.Lastname,

OrderAmount = inv.Invtotal,

OrderDate = inv.Invdate,

OrderNumber = inv.Ordercode

};

var settings = new HttpRequestSettings

{

Url = NotificationUrl + "api/WebStore/OrderNotification",

Content = request,

ContentType="application/json",

Timeout = 3000,

HttpVerb = "POST",

};

try

{

var result = HttpUtils.JsonRequest<WebStoreOrderNotification>(settings);

}

catch

{

// ignore the error!

}

}

public static void ItemAddedNotification(busTLineItem lineItem)

{

var li = lineItem.Entity;

var request = new WebStoreItemAddedNotification

{

Description = li.Descript,

Sku = li.Sku,

Price = li.Price,

Discount = li.Discount,

Qty = li.Qty

};

var settings = new HttpRequestSettings

{

Url = NotificationUrl + "api/WebStore/ItemAddedNotification",

Content = request,

ContentType = "application/json",

Timeout = 3000,

HttpVerb = "POST",

};

try

{

var result = HttpUtils.JsonRequest<WebStoreItemAddedNotification>(settings);

}

catch

{

// ignore the error

}

}

}

public class WebStoreOrderNotification

{

public string OrderNumber { get; set; }

public decimal OrderAmount { get; set; }

public DateTime OrderDate { get; set; }

public string CustomerName { get; set; }

public int CustomerId { get; set; }

}

public class WebStoreItemAddedNotification

{

public string Sku { get; set; }

public string Description { get; set; }

public decimal Qty { get; set; }

public decimal Price { get; set; }

public decimal Discount { get; set; }

}

This code uses the [Westwind.Utilities](https://github.com/RickStrahl/Westwind.Utilities) [HttpUtils.JsonRequest()](https://west-wind.com/westwindtoolkit/docs/_4ij04e9r7.htm) function which makes it really easy to make an HTTP call with a JSON object input and result.

Both methods take business objects as parameters. The BO’s hold the actual entity data for each an order and lineitem internally which is pulled out and stuffed into the DTOs. The DTOs are used for the Content data that is serialized and sent to the web service by **JsonRequest().**

I then integrate these static method calls into the actual Web site where it makes sense. For the item item added this goes into the Item page code:

if (!TLine.Save() )

{

DisplayError(TLine.ErrorMessage);

return;

}

WebStoreNotifications.ItemAddedNotification(TLine);

Now whenever an item is added to any shopping cart a notification is sent to the Notifications Web Service and then on to the connected clients.

For the order form:

if (!Invoice.Save( saveFromTempItems ))

{

ErrorDisplay.ShowError(Invoice.ErrorMessage, "An error occurred saving the invoice:");

return;

}

WebStoreNotifications.OrderNotification(Invoice);

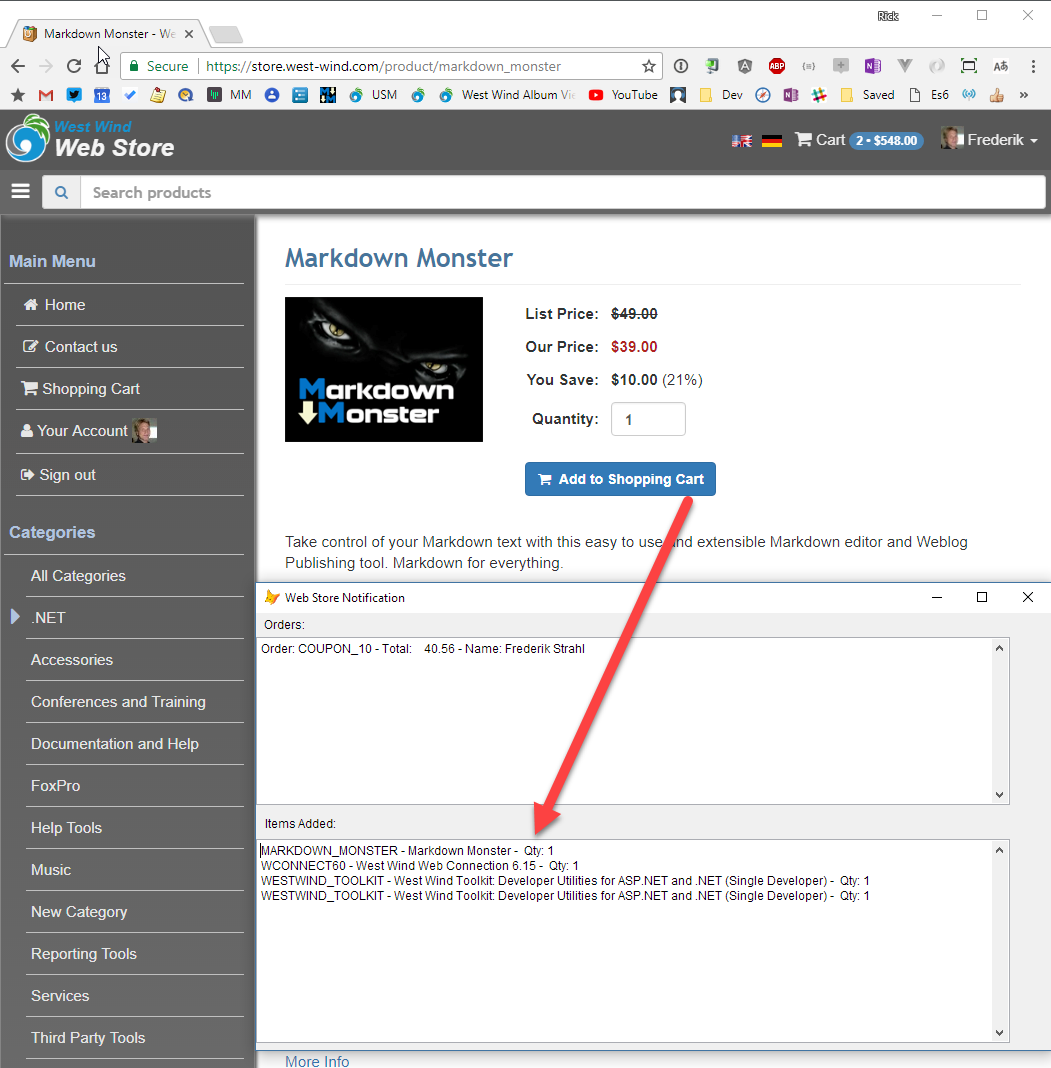
Same deal – when an order is placed I should now see a notification.

### Another FoxPro Form

As with the chat example, it’s nice to actually see this information in a on form and a form makes it nice and easy to keep the connection to the SignalR hub open by attaching the SignalR clint instance to the

form.

Here’s what the form looks like:



I kept this real simple just to get the point across. When you click on a button to add a new item in the shopping cart a notification fires on the FoxPro form. Likewise when an order is placed you’ll see a notification on the form.

The code should look familiar. The form **Init** looks like this:

DO WebStoreNotificationsClient

IF TYPE("gcSignalRUrl") # "C"

PUBLIC gcSignalUrl

gcSignalRUrl = "http://signalrswf.west-wind.com/"

ENDIF

this.oNotifications = CREATEOBJECT("WebStoreNotificationsClient")

this.oNotifications.oNotifications.SignalRUrl = gcSignalRUrl

\*\*\* Forward events to this form

BINDEVENT(this.oNotifications,"OnNotifyOrder",this,"OnNotifyOrder")

BINDEVENT(this.oNotifications,"OnNotifyItemAdded",this,"OnNotifyItemAdded")

this.oNotifications.Start()

OnNotifyItemAdded looks like this:

LPARAMETERS loNotification

lcTextBox = this.txtItemAdded.Value

lcText = loNotification.Sku + " - " + loNotification.Description + " - Qty: " + TRANSFORM(loNotification.Qty) + CHR(13) + CHR(10)

lcTextBox = lcTextBox + lcText

this.txtItemAdded.Value = lcTextBox

this.txtItemAdded.SelStart = this.txtItemAdded.SelLength

OnNotifyOrder is pretty much the same except with order info.

Voila – there you have it a full notifications hub that can notify when anything – anything at all happens on a Web site you can be instantly notified inside of your desktop application.

### In Praxis

I’ve worked on a handful of systems that have used this functionality to great effect.

* **Call Center Forwarding**Voice over IP solution that allows forwarding of phone calls to clients. FoxPro application needed to know when calls are made and pick up those calls based on ids. The VOIP provider had an HTTP callback interface that we could build a .NET base Service for, which then forwarded those requests to the FoxPro application which could then pick up the call using the proprietary VOIP software from the desktop application
* **Web Shop API Using Web Sockets**  
  Worked with a customer who needed to integrated with an E-Commerce provider in the UK whose entire shop platform is hosted online and publishes real time order information via Web Sockets. Essentially the provider provides the E-Commerce front end and the WebSockets interface pushes the data to a number of FoxPro desktop applications which then handle the actual backend order processing. In this case a FoxPro daemon service just sits and listens to incoming orders and routes orders to be handled by the appropriate staff.
* **FoxPro Pushing Real Time Data To Mobile Applications**Using multi-casting or just push technology also allows you to push real time data directly to connected applications that could be running on mobile devices or even IOT devices.

Push technology is still in its infancy and there are many ways that this can be taken advantage of once you really think about how this technology works. There are many opportunities for hosted applications to publish data in real time to other backend or front end applications. This is often much easier to integrate into existing business solutions than traditional Web based technologies because they require heavy investment in server programming (most likely using a different language and environment). With peer to peer services you can push the data directly to the old application and use and process that data inside of the original environment.

### Is it For You?

While it’s exciting to see this technology evolve and just as exciting to see that we can use it within FoxPro, there’s no doubt that it’s not trivial to build applications with it. As you’ve seen there’s a fair bit of busy work involved forwarding information across the various messaging layers and each layer needs an implementation.

The good news is that that the code is mostly boiler plate plumbing that is easy to write and easy to understand and almost copy verbatim. The .NET Proxy, the .NET Hub are mostly forwarding routines that take incoming messages and redistribute them to the connected clients that do the actual work. In other words, the **real** work is left for the handling application – FoxPro in this case.

Looking at all that is involved may feel daunting, but the reality is for what you can do with this technology, it’s surprisingly simple to get this to work if you compared with what was involved in the past to build connected peer to peer applications using TCP/IP for example.

So, if you think that this technology would be helpful to you, don’t discount it because of complexity – it’s easier than it looks especially once you’ve created something simple and you have followed the flow all the way through.

I hope this very long article has done just that and given you some ideas of how you can add real time features to your applications. Try it out on your own, either by creating these samples on your own or by creating some simple examples of your own.

Go ahead – build something amazing!

## Resources

* [Source Code and Samples on BitBucket](https://bitbucket.org/RickStrahl/swfox16_signalr)
* [White Paper](https://bitbucket.org/RickStrahl/swfox16_signalr/raw/master/Documents/Strahl_SignalR.docx) (this one)
* [Slides](https://bitbucket.org/RickStrahl/swfox16_signalr/raw/master/Documents/Strahl_SignalR.pptx)