ASP.NET SignalR Hubs API Guide - Server (C#)

By Tom Dykstra and Patrick Fletcher | June 10, 2014 | Level 100 : Beginner

This document provides an introduction to programming the server side of the ASP.NET SignalR Hubs API for SignalR version 2, with code samples demonstrating common options.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, see **Introduction to SignalR 2 (/signalr/overview/signalr-20/getting-started-with-signalr-20/introduction-to-signalr)** .

Software versions used in this topic

Topic versions

Questions and comments

Overview

This document contains the following sections:

- **How to register SignalR middleware (#route)**
  - **The /signalr URL (#signalrurl)**
  - **Configuring SignalR options (#options)**
- **How to create and use Hub classes (#hubclass)**
  - **Hub object lifetime (#transience)**
  - **Camel-casing of Hub names in JavaScript clients (#hubnames)**
  - **Multiple Hubs (#multiplehubs)**
  - **Strongly-Typed Hubs (#stronglytypedhubs)**
- **How to define methods in the Hub class that clients can call (#hubmethods)**
  - **Camel-casing of method names in JavaScript clients (#methodnames)**
  - **When to execute asynchronously (#asyncmethods)**
  - **Defining overloads (#overloads)**
  - **Reporting progress from hub method invocations (#progress)**
- **How to call client methods from the Hub class (#callfromhub)**
  - **Selecting which clients will receive the RPC (#selectingclients)**
  - **No compile-time validation for method names (#dynamicmethodnames)**

    - Case-insensitive method name matching (#caseinsensitive)
    - Asynchronous execution (#asyncclient)
- How to manage group membership from the Hub class (#groupsfromhub)
    - Asynchronous execution of Add and Remove methods (#asyncgroupmethods)
    - Group membership persistence (#grouppersistence)
    - Single-user groups (#singleusergroups)
- How to handle connection lifetime events in the Hub class (#connectionlifetime)
    - When OnConnected, OnDisconnected, and OnReconnected are called (#onreconnected)
    - Caller state not populated (#nocallerstate)
- How to get information about the client from the Context property (#contextproperty)
- How to pass state between clients and the Hub class (#passstate)
- How to handle errors in the Hub class (#handleErrors)
- How to call client methods and manage groups from outside the Hub class (#callfromoutsidehub)
    - Calling client methods (#callingclientsoutsidehub)
    - Managing group membership (#managinggroupsoutsidehub)
- How to enable tracing (#tracing)
- How to customize the Hubs pipeline (#hubpipeline)

For documentation on how to program clients, see the following resources:

- SignalR Hubs API Guide - JavaScript Client (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client)
- SignalR Hubs API Guide - .NET Client (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-net-client)

The server components for SignalR 2 are only available in .NET 4.5. Servers running .NET 4.0 must use SignalR v1.x.

How to register SignalR middleware

To define the route that clients will use to connect to your Hub, call the `MapSignalR` method when the application starts. `MapSignalR` is an **extension method (http://msdn.microsoft.com/en-us/library/vstudio/bb383977.aspx)** for the `OwinExtensions` class. The following example shows how to define the SignalR Hubs route using an OWIN startup class.

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MyApplication.Startup))]
namespace MyApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }

    }
}
```

If you are adding SignalR functionality to an ASP.NET MVC application, make sure that the SignalR route is added before the other routes. For more information, see **Tutorial: Getting Started with SignalR 2 and MVC 5 (/signalr/overview/signalr-20/getting-started-with-signalr-20/tutorial-getting-started-with-signalr-20-and-mvc-5)** .

The /signalr URL

By default, the route URL which clients will use to connect to your Hub is "/signalr". (Don't confuse this URL with the "/signalr/hubs" URL, which is for the automatically generated JavaScript file. For more information about the generated proxy, see **SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client#genproxy)** .)

There might be extraordinary circumstances that make this base URL not usable for SignalR; for example, you have a folder in your project named *signalr* and you don't want to change the name. In that case, you can change the base URL, as shown in the following examples (replace "/signalr" in the sample code with your desired URL).

Server code that specifies the URL

```
app.MapSignalR("/signalr", new HubConfiguration (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubconfiguration(v=vs.111).aspx) ());
```

JavaScript client code that specifies the URL (with the generated proxy)

```
$.connection.hub.url = "/signalr"
$.connection.hub.start().done(init);
```

JavaScript client code that specifies the URL (without the generated proxy)

```
var connection = $.hubConnection("/signalr", { useDefaultPath: false });
```

.NET client code that specifies the URL

```
var hubConnection = new HubConnection("http://contoso.com/signalr", useDefaultUrl: false);
```

Configuring SignalR Options

Overloads of the `MapSignalR` method enable you to specify a custom URL, a custom dependency resolver, and the following options:

- Enable cross-domain calls using CORS or JSONP from browser clients.

  Typically if the browser loads a page from `http://contoso.com`, the SignalR connection is in the same domain, at `http://contoso.com/signalr`. If the page from `http://contoso.com` makes a connection to `http://fabrikam.com/signalr`, that is a cross-domain connection. For security reasons, cross-domain connections are disabled by default. For more information, see **ASP.NET SignalR Hubs API Guide - JavaScript Client - How to establish a cross-domain connection (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client#crossdomain)** .

- Enable detailed error messages.

  When errors occur, the default behavior of SignalR is to send to clients a notification message without details about what happened. Sending detailed error information to clients is not recommended in production, because malicious users might be able to use the information in attacks against your application. For troubleshooting, you can use this option to temporarily enable more informative error reporting.

- Disable automatically generated JavaScript proxy files.

  By default, a JavaScript file with proxies for your Hub classes is generated in response to the URL "/signalr/hubs". If you don't want to use the JavaScript proxies, or if you want to generate this file manually and refer to a physical file in your clients, you can use this option to disable proxy generation. For more information, see **SignalR Hubs API Guide - JavaScript Client - How to create a physical file for the SignalR generated proxy (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client#manualproxy)** .

The following example shows how to specify the SignalR connection URL and these options in a call to the `MapSignalR` method. To specify a custom URL, replace "/signalr" in the example with the URL that you want to use.

```
var hubConfiguration = new HubConfiguration (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubconfiguration(v=vs.111).aspx) ();
hubConfiguration.EnableDetailedErrors = true;
hubConfiguration.EnableJavaScriptProxies = false;
app.MapSignalR("/signalr", hubConfiguration);
```

How to create and use Hub classes

To create a Hub, create a class that derives from **Microsoft.Aspnet.Signalr.Hub (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hub(v=vs.111).aspx)** . The following example shows a simple Hub class for a chat application.

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

In this example, a connected client can call the `NewContosoChatMessage` method, and when it does, the data received is broadcasted to all connected clients.

Hub object lifetime

You don't instantiate the Hub class or call its methods from your own code on the server; all that is done for you by the SignalR Hubs pipeline. SignalR creates a new instance of your Hub class each time it needs to handle a Hub operation such as when a client connects, disconnects, or makes a method call to the server.

Because instances of the Hub class are transient, you can't use them to maintain state from one method call to the next. Each time the server receives a method call from a client, a new instance of your Hub class processes the message. To maintain state through multiple connections and method calls, use some other method such as a database, or a static variable on the

Hub class, or a different class that does not derive from `Hub`. If you persist data in memory, using a method such as a static variable on the Hub class, the data will be lost when the app domain recycles.

If you want to send messages to clients from your own code that runs outside the Hub class, you can't do it by instantiating a Hub class instance, but you can do it by getting a reference to the SignalR context object for your Hub class. For more information, see **How to call client methods and manage groups from outside the Hub class (#callfromoutsidehub)** later in this topic.

Camel-casing of Hub names in JavaScript clients

By default, JavaScript clients refer to Hubs by using a camel-cased version of the class name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions. The previous example would be referred to as `contosoChatHub` in JavaScript code.

Server

```
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
```

If you want to specify a different name for clients to use, add the `HubName` attribute. When you use a `HubName` attribute, there is no name change to camel case on JavaScript clients.

Server

```
[HubName("PascalCaseContosoChatHub")]
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.PascalCaseContosoChatHub;
```

Multiple Hubs

You can define multiple Hub classes in an application. When you do that, the connection is shared but groups are separate:

- All clients will use the same URL to establish a SignalR connection with your service ("/signalr" or your custom URL if you specified one), and that connection is used for all Hubs defined by the service.

  There is no performance difference for multiple Hubs compared to defining all Hub functionality in a single class.

- All Hubs get the same HTTP request information.

  Since all Hubs share the same connection, the only HTTP request information that the server gets is what comes in the original HTTP request that establishes the SignalR connection. If you use the connection request to pass information

from the client to the server by specifying a query string, you can't provide different query strings to different Hubs. All Hubs will receive the same information.

- The generated JavaScript proxies file will contain proxies for all Hubs in one file.

  For information about JavaScript proxies, see **SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client#genproxy)** .

- Groups are defined within Hubs.

  In SignalR you can define named groups to broadcast to subsets of connected clients. Groups are maintained separately for each Hub. For example, a group named "Administrators" would include one set of clients for your `ContosoChatHub` class, and the same group name would refer to a different set of clients for your `StockTickerHub` class.

Strongly-Typed Hubs

To define an interface for your hub methods that your client can reference (and enable Intellisense on your hub methods), derive your hub from `Hub<T>` (introduced in SignalR 2.1) rather than `Hub`:

```
public class StrongHub : Hub<IClient>
{
    public void Send(string message)
    {
        Clients.All.NewMessage(message);
    }
}

public interface IClient
{
    void NewMessage(string message);
}
```

How to define methods in the Hub class that clients can call

To expose a method on the Hub that you want to be callable from the client, declare a public method, as shown in the following examples.

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
```

```
        return _stockTicker.GetAllStocks();
    }
}
```

You can specify a return type and parameters, including complex types and arrays, as you would in any C# method. Any data that you receive in parameters or return to the caller is communicated between the client and the server by using JSON, and SignalR handles the binding of complex objects and arrays of objects automatically.

Camel-casing of method names in JavaScript clients

By default, JavaScript clients refer to Hub methods by using a camel-cased version of the method name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

Server

```
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.newContosoChatMessage(userName, message);
```

If you want to specify a different name for clients to use, add the `HubMethodName` attribute.

Server

```
[HubMethodName("PascalCaseNewContosoChatMessage")]
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.PascalCaseNewContosoChatMessage(userName, message);
```

When to execute asynchronously

If the method will be long-running or has to do work that would involve waiting, such as a database lookup or a web service call, make the Hub method asynchronous by returning a **Task (http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx)** (in place of `void` return) or **Task<T> (http://msdn.microsoft.com/en-us/library/dd321424.aspx)** object (in place of `T` return type). When you return a `Task` object from the method, SignalR waits for the `Task` to complete, and then it sends the unwrapped result back to the client, so there is no difference in how you code the method call in the client.

Making a Hub method asynchronous avoids blocking the connection when it uses the WebSocket transport. When a Hub method executes synchronously and the transport is WebSocket, subsequent invocations of methods on the Hub from the same client are blocked until the Hub method completes.

The following example shows the same method coded to run synchronously or asynchronously, followed by JavaScript client code that works for calling either version.

Synchronous

```
public IEnumerable<Stock> GetAllStocks()
{
        // Returns data from memory.
        return _stockTicker.GetAllStocks();
}
```

Asynchronous

```
public async Task<IEnumerable<Stock>> GetAllStocks()
{
    // Returns data from a web service.
    var uri = Util.getServiceUri("Stocks");
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<IEnumerable<Stock>>());
    }
}
```

JavaScript client using generated proxy

```
stockTickerHubProxy.server.getAllStocks().done(function (stocks) {
    $.each(stocks, function () {
        alert(this.Symbol + ' ' + this.Price);
    });
});
```

For more information about how to use asynchronous methods in ASP.NET 4.5, see **Using Asynchronous Methods in ASP.NET MVC 4 (/mvc/tutorials/mvc-4/using-asynchronous-methods-in-aspnet-mvc-4)** .

Defining Overloads

If you want to define overloads for a method, the number of parameters in each overload must be different. If you differentiate an overload just by specifying different parameter types, your Hub class will compile but the SignalR service will throw an exception at run time when clients try to call one of the overloads.

Reporting progress from hub method invocations

SignalR 2.1 adds support for the **progress reporting pattern (http://blogs.msdn.com/b/dotnet/archive/2012/06/06/async-in-4-5-enabling-progress-and-cancellation-in-async-apis.aspx)** introduced in .NET 4.5. To implement progress reporting, define an `IProgress<T>` parameter for your hub method that your client can access:

```
public class ProgressHub : Hub
{
    public async Task<string> DoLongRunningThing(IProgress<int> progress)
    {
        for (int i = 0; i <= 100; i+=5)
        {
            await Task.Delay(200);
            progress.Report(i);
        }
        return "Job complete!";
    }
}
```

When writing a long-running server method, it is important to use an asynchronous programming pattern like Async/ Await rather than blocking the hub thread.

How to call client methods from the Hub class

To call client methods from the server, use the `Clients` property in a method in your Hub class. The following example shows server code that calls `addNewMessageToPage` on all connected clients, and client code that defines the method in a JavaScript client.

Server

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

JavaScript client using generated proxy

```
contosoChatHubProxy.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '<li>');
};
```

You can't get a return value from a client method; syntax such as `int x = Clients.All.add(1,1)` does not work.

You can specify complex types and arrays for the parameters. The following example passes a complex type to the client in a method parameter.

Server code that calls a client method using a complex object

```
public void SendMessage(string name, string message)
{
    Clients.All.addContosoChatMessageToPage(new ContosoChatMessage() { UserName = name, Message =
message });
}
```

Server code that defines the complex object

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Selecting which clients will receive the RPC

The Clients property returns a **HubConnectionContext (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext(v=vs.111).aspx)** object that provides several options for specifying which clients will receive the RPC:

- All connected clients.

```
Clients.All (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.all(v=vs.111).aspx)
.addContosoChatMessageToPage(name, message);
```

- Only the calling client.

```
Clients.Caller (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.caller(v=vs.111).aspx)
.addContosoChatMessageToPage(name, message);
```

- All clients except the calling client.

```
Clients.Others (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.others(v=vs.111).aspx)
.addContosoChatMessageToPage(name, message);
```

- A specific client identified by connection ID.

```
Clients.Client (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.client(v=vs.111).aspx)(Context.ConnectionId).addContosoChatMessageToPage(name, message);
```

This example calls `addContosoChatMessageToPage` on the calling client and has the same effect as using `Clients.Caller`.

- All connected clients except the specified clients, identified by connection ID.

```
Clients.AllExcept (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.allexcept(v=vs.111).aspx) (connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group.

```
Clients.Group (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.group(v=vs.111).aspx)(groupName).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except the specified clients, identified by connection ID.

```
Clients.Group (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.group(v=vs.111).aspx) (groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except the calling client.

```
Clients.OthersInGroup (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.othersingroup(v=vs.111).aspx)(groupName).addContosoChatMessageToPage(name, message);
```

- A specific user, identified by userId.

```
Clients.User(userid).addContosoChatMessageToPage(name, message);
```

By default, this is `IPrincipal.Identity.Name`, but this can be changed by **registering an implementation of IUserIdProvider with the global host (/signalr/overview/signalr-20/hubs-api/mapping-users-to-connections#IUserIdProvider)** .

- All clients and groups in a list of connection IDs.

```
Clients.Clients(ConnectionIds).broadcastMessage(name, message);
```

- A list of groups.

```
Clients.Groups(GroupIds).broadcastMessage(name, message);
```

- A user by name.

```
Clients.Client(username).broadcastMessage(name, message);
```

- A list of user names (introduced in SignalR 2.1).

```
Clients.Users(new string[] { "myUser", "myUser2" }).broadcastMessage(name, message);
```

No compile-time validation for method names

The method name that you specify is interpreted as a dynamic object, which means there is no IntelliSense or compile-time validation for it. The expression is evaluated at run time. When the method call executes, SignalR sends the method name and the parameter values to the client, and if the client has a method that matches the name, that method is called and the parameter values are passed to it. If no matching method is found on the client, no error is raised. For information about the format of the data that SignalR transmits to the client behind the scenes when you call a client method, see **Introduction to SignalR (/signalr/overview/signalr-20/getting-started-with-signalr-20/introduction-to-signalr)** .

Case-insensitive method name matching

Method name matching is case-insensitive. For example, `Clients.All.addContosoChatMessageToPage` on the server will execute `AddContosoChatMessageToPage`, `addcontosochatmessagetopage`, or `addContosoChatMessageToPage` on the client.

Asynchronous execution

The method that you call executes asynchronously. Any code that comes after a method call to a client will execute immediately without waiting for SignalR to finish transmitting data to clients unless you specify that the subsequent lines of code should wait for method completion. The following code example shows how to execute two client methods sequentially.

Using Await (.NET 4.5)

```
public async Task NewContosoChatMessage(string name, string message)
{
    await Clients.Others.addContosoChatMessageToPage(data);
    Clients.Caller.notifyMessageSent();
}
```

If you use `await` to wait until a client method finishes before the next line of code executes, that does not mean that clients will actually receive the message before the next line of code executes. "Completion" of a client method call only means that SignalR has done everything necessary to send the message. If you need verification that clients received the message, you have to program that mechanism yourself. For example, you could code a `MessageReceived` method on the Hub, and in the `addContosoChatMessageToPage` method on the client you could call `MessageReceived` after you do whatever work you

need to do on the client. In `MessageReceived` in the Hub you can do whatever work depends on actual client reception and processing of the original method call.

How to use a string variable as the method name

If you want to invoke a client method by using a string variable as the method name, cast `Clients.All` (or `Clients.Others`, `Clients.Caller`, etc.) to `IClientProxy` and then call **Invoke(methodName, args...) (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.iclientproxy.invoke(v=vs.111).aspx)** .

```
public void NewContosoChatMessage(string name, string message)
{
    string methodToCall = "addContosoChatMessageToPage";
    IClientProxy proxy = Clients.All;
    proxy.Invoke(methodToCall, name, message);
}
```

How to manage group membership from the Hub class

Groups in SignalR provide a method for broadcasting messages to specified subsets of connected clients. A group can have any number of clients, and a client can be a member of any number of groups.

To manage group membership, use the **Add (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.igroupmanager.add(v=vs.111).aspx)** and **Remove (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.igroupmanager.remove(v=vs.111).aspx)** methods provided by the `Groups` property of the Hub class. The following example shows the `Groups.Add` and `Groups.Remove` methods used in Hub methods that are called by client code, followed by JavaScript client code that calls them.

Server

```
public class ContosoChatHub : Hub
{
    public Task JoinGroup(string groupName)
    {
        return Groups.Add(Context.ConnectionId, groupName);
    }

    public Task LeaveGroup(string groupName)
    {
        return Groups.Remove(Context.ConnectionId, groupName);
    }
}
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.joinGroup(groupName);
```

```
contosoChatHubProxy.server.leaveGroup(groupName);
```

You don't have to explicitly create groups. In effect a group is automatically created the first time you specify its name in a call to `Groups.Add`, and it is deleted when you remove the last connection from membership in it.

There is no API for getting a group membership list or a list of groups. SignalR sends messages to clients and groups based on a **pub/sub model (http://en.wikipedia.org/wiki/Publish/subscribe)** , and the server does not maintain lists of groups or group memberships. This helps maximize scalability, because whenever you add a node to a web farm, any state that SignalR maintains has to be propagated to the new node.

Asynchronous execution of Add and Remove methods

The `Groups.Add` and `Groups.Remove` methods execute asynchronously. If you want to add a client to a group and immediately send a message to the client by using the group, you have to make sure that the `Groups.Add` method finishes first. The following code example shows how to do that.

Adding a client to a group and then messaging that client

```
public async Task JoinGroup(string groupName)
{
    await Groups.Add(Context.ConnectionId, groupName);
    Clients.Group(groupname).addContosoChatMessageToPage(Context.ConnectionId + " added to
group");
}
```

Group membership persistence

SignalR tracks connections, not users, so if you want a user to be in the same group every time the user establishes a connection, you have to call `Groups.Add` every time the user establishes a new connection.

After a temporary loss of connectivity, sometimes SignalR can restore the connection automatically. In that case, SignalR is restoring the same connection, not establishing a new connection, and so the client's group membership is automatically restored. This is possible even when the temporary break is the result of a server reboot or failure, because connection state for each client, including group memberships, is round-tripped to the client. If a server goes down and is replaced by a new server before the connection times out, a client can reconnect automatically to the new server and re-enroll in groups it is a member of.

When a connection can't be restored automatically after a loss of connectivity, or when the connection times out, or when the client disconnects (for example, when a browser navigates to a new page), group memberships are lost. The next time the user connects will be a new connection. To maintain group memberships when the same user establishes a new connection, your application has to track the associations between users and groups, and restore group memberships each time a user establishes a new connection.

For more information about connections and reconnections, see **How to handle connection lifetime events in the Hub class (#connectionlifetime)** later in this topic.

Single-user groups

Applications that use SignalR typically have to keep track of the associations between users and connections in order to know which user has sent a message and which user(s) should be receiving a message. Groups are used in one of the two commonly used patterns for doing that.

- Single-user groups.

  You can specify the user name as the group name, and add the current connection ID to the group every time the user connects or reconnects. To send messages to the user you send to the group. A disadvantage of this method is that the group doesn't provide you with a way to find out if the user is online or offline.

- Track associations between user names and connection IDs.

  You can store an association between each user name and one or more connection IDs in a dictionary or database, and update the stored data each time the user connects or disconnects. To send messages to the user you specify the connection IDs. A disadvantage of this method is that it takes more memory.

How to handle connection lifetime events in the Hub class

Typical reasons for handling connection lifetime events are to keep track of whether a user is connected or not, and to keep track of the association between user names and connection IDs. To run your own code when clients connect or disconnect, override the `OnConnected`, `OnDisconnected`, and `OnReconnected` virtual methods of the Hub class, as shown in the following example.

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        // Add your own code here.
        // For example: in a chat application, record the association between
        // the current connection ID and user name, and mark the user as online.
        // After the code in this method completes, the client is informed that
        // the connection is established; for example, in a JavaScript client,
        // the start().done callback is executed.
        return base.OnConnected();
    }

    public override Task OnDisconnected()
    {
        // Add your own code here.
        // For example: in a chat application, mark the user as offline,
        // delete the association between the current connection id and user name.
        return base.OnDisconnected();
    }

    public override Task OnReconnected()
    {
        // Add your own code here.
        // For example: in a chat application, you might have marked the
        // user as offline after a period of inactivity; in that case
        // mark the user as online again.
        return base.OnReconnected();
    }
}
```

When OnConnected, OnDisconnected, and OnReconnected are called

Each time a browser navigates to a new page, a new connection has to be established, which means SignalR will execute the `OnDisconnected` method followed by the `OnConnected` method. SignalR always creates a new connection ID when a new connection is established.

The `OnReconnected` method is called when there has been a temporary break in connectivity that SignalR can automatically recover from, such as when a cable is temporarily disconnected and reconnected before the connection times out. The `OnDisconnected` method is called when the client is disconnected and SignalR can't automatically reconnect, such as when a browser navigates to a new page. Therefore, a possible sequence of events for a given client is `OnConnected`, `OnReconnected`, `OnDisconnected`; or `OnConnected`, `OnDisconnected`. You won't see the sequence `OnConnected`, `OnDisconnected`, `OnReconnected` for a given connection.

The `OnDisconnected` method doesn't get called in some scenarios, such as when a server goes down or the App Domain gets recycled. When another server comes on line or the App Domain completes its recycle, some clients may be able to reconnect and fire the `OnReconnected` event.

For more information, see **Understanding and Handling Connection Lifetime Events in SignalR (/signalr/overview/signalr-20/hubs-api/handling-connection-lifetime-events)** .

Caller state not populated

The connection lifetime event handler methods are called from the server, which means that any state that you put in the `state` object on the client will not be populated in the `Caller` property on the server. For information about the `state` object and the `Caller` property, see **How to pass state between clients and the Hub class (#passstate)** later in this topic.

How to get information about the client from the Context property

To get information about the client, use the `Context` property of the Hub class. The `Context` property returns a **HubCallerContext (http://msdn.microsoft.com/en-us/library/jj890883(v=vs.111).aspx)** object which provides access to the following information:

- The connection ID of the calling client.

```
string connectionID = Context.ConnectionId;
```

  The connection ID is a GUID that is assigned by SignalR (you can't specify the value in your own code). There is one connection ID for each connection, and the same connection ID is used by all Hubs if you have multiple Hubs in your application.

- HTTP header data.

```
System.Collections.Specialized.NameValueCollection headers = Context.Request.Headers;
```

  You can also get HTTP headers from `Context.Headers`. The reason for multiple references to the same thing is that `Context.Headers` was created first, the `Context.Request` property was added later, and `Context.Headers` was retained for backward compatibility.

- Query string data.

```
System.Collections.Specialized.NameValueCollection queryString = Context.Request.QueryString;
string parameterValue = queryString["parametername"]
```

You can also get query string data from `Context.QueryString`.

The query string that you get in this property is the one that was used with the HTTP request that established the SignalR connection. You can add query string parameters in the client by configuring the connection, which is a convenient way to pass data about the client from the client to the server. The following example shows one way to add a query string in a JavaScript client when you use the generated proxy.

```
$.connection.hub.qs = { "version" : "1.0" };
```

For more information about setting query string parameters, see the API guides for the **JavaScript (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-javascript-client)** and **.NET (/signalr/overview/signalr-20/hubs-api/hubs-api-guide-net-client)** clients.

You can find the transport method used for the connection in the query string data, along with some other values used internally by SignalR:

```
string transportMethod = queryString["transport"];
```

The value of `transportMethod` will be "webSockets", "serverSentEvents", "foreverFrame" or "longPolling". Note that if you check this value in the `OnConnected` event handler method, in some scenarios you might initially get a transport value that is not the final negotiated transport method for the connection. In that case the method will throw an exception and will be called again later when the final transport method is established.

- Cookies.

```
System.Collections.Generic.IDictionary<string, Cookie> cookies = Context.Request.Cookies;
```

You can also get cookies from `Context.RequestCookies`.

- User information.

```
System.Security.Principal.IPrincipal user = Context.User;
```

- The HttpContext object for the request :

```
System.Web.HttpContextBase httpContext = Context.Request.GetHttpContext();
```

Use this method instead of getting `HttpContext.Current` to get the `HttpContext` object for the SignalR connection.

How to pass state between clients and the Hub class

The client proxy provides a `state` object in which you can store data that you want to be transmitted to the server with each method call. On the server you can access this data in the `Clients.Caller` property in Hub methods that are called by clients. The `Clients.Caller` property is not populated for the connection lifetime event handler methods `OnConnected`, `OnDisconnected`, and `OnReconnected`.

Creating or updating data in the `state` object and the `Clients.Caller` property works in both directions. You can update values in the server and they are passed back to the client.

The following example shows JavaScript client code that stores state for transmission to the server with every method call.

```
contosoChatHubProxy.state.userName = "Fadi Fakhouri";
contosoChatHubProxy.state.computerName = "fadivm1";
```

The following example shows the equivalent code in a .NET client.

```
contosoChatHubProxy["userName"] = "Fadi Fakhouri";
chatHubProxy["computerName"] = "fadivm1";
```

In your Hub class, you can access this data in the `Clients.Caller` property. The following example shows code that retrieves the state referred to in the previous example.

```
public void NewContosoChatMessage(string data)
{
    string userName = Clients.Caller.userName;
    string computerName = Clients.Caller.computerName;
    Clients.Others.addContosoChatMessageToPage(message, userName, computerName);
}
```

Note: This mechanism for persisting state is not intended for large amounts of data, since everything you put in the `state` or `Clients.Caller` property is round-tripped with every method invocation. It's useful for smaller items such as user names or counters.

In VB.NET or in a strongly-typed hub, the caller state object can't be accessed through `Clients.Caller`; instead, use `Clients.CallerState` (introduced in SignalR 2.1):

Using CallerState in C#

```
public void NewContosoChatMessage(string data)
{
    string userName = Clients.CallerState.userName;
    string computerName = Clients.CallerState.computerName;
    Clients.Others.addContosoChatMessageToPage(data, userName, computerName);
}
```

Using CallerState in Visual Basic

```
Public Sub NewContosoChatMessage(message As String)
    Dim userName As String = Clients.CallerState.userName
    Dim computerName As String = Clients.CallerState.computerName
    Clients.Others.addContosoChatMessageToPage(message, userName, computerName)
End Sub
```

How to handle errors in the Hub class

To handle errors that occur in your Hub class methods, use one or more of the following methods:

- Wrap your method code in try-catch blocks and log the exception object. For debugging purposes you can send the exception to the client, but for security reasons sending detailed information to clients in production is not recommended.

- Create a Hubs pipeline module that handles the **OnIncomingError (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubpipelinemodule.onincomingerror(v=vs.111).aspx)** method. The following example shows a pipeline module that logs errors, followed by code in Startup.cs that injects the module into the Hubs pipeline.

  ```
  public class ErrorHandlingPipelineModule : HubPipelineModule
  {
      protected override void OnIncomingError(ExceptionContext exceptionContext,
  IHubIncomingInvokerContext invokerContext)
          {
              Debug.WriteLine("=> Exception " + exceptionContext.Error.Message);
              if (exceptionContext.Error.InnerException != null)
              {
                  Debug.WriteLine("=> Inner Exception " +
  exceptionContext.Error.InnerException.Message);
              }
              base.OnIncomingError(exceptionContext, invokerContext);

          }
  }
  ```

  ```
  public void Configuration(IAppBuilder app)
  {
      // Any connection or hub wire up and configuration should go here
      GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule());
      app.MapSignalR();
  }
  ```

- Use the `HubException` class (introduced in SignalR 2). This error can be thrown from any hub invocation. The `HubError` constructor takes a string message, and an object for storing extra error data. SignalR will auto-serialize the exception and send it to the client, where it will be used to reject or fail the hub method invocation.

  The following code samples demonstrate how to throw a `HubException` during a Hub invocation, and how to handle the exception on JavaScript and .NET clients.

Server code demonstrating the HubException class

```
public class MyHub : Hub
{
    public void Send(string message)
    {
        if(message.Contains("<script>"))
        {
            throw new HubException("This message will flow to the client", new { user =
Context.User.Identity.Name, message = message });
        }

        Clients.All.send(message);
    }
}
```

JavaScript client code demonstrating response to throwing a HubException in a hub

```
myHub.server.send("<script>")
            .fail(function (e) {
                if (e.source === 'HubException') {
                    console.log(e.message + ' : ' + e.data.user);
                }
            });
```

.NET client code demonstrating response to throwing a HubException in a hub

```
try
{
    await myHub.Invoke("Send", "<script>");
}
catch(HubException ex)
{
    Conosle.WriteLine(ex.Message);
}
```

For more information about Hub pipeline modules, see **How to customize the Hubs pipeline (#hubpipeline)** later in this topic.

How to enable tracing

To enable server-side tracing, add a system.diagnostics element to your Web.config file, as shown in this example:

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
```

```
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="SignalRSamples" connectionString="Data Source=(local);Initial
Catalog=SignalRSamples;Integrated Security=SSPI;Asynchronous Processing=True;" />
    <add name="SignalRSamplesWithMARS" connectionString="Data Source=(local);Initial
Catalog=SignalRSamples;Integrated Security=SSPI;Asynchronous
Processing=True;MultipleActiveResultSets=true;" />
  </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true" />
  </system.webServer>
  <system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
     <source name="SignalR.ServiceBusMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
     </source>
     <source name="SignalR.ScaleoutMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.Transports.WebSocketTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.ServerSentEventsTransport">
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.ForeverFrameTransport">
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.LongPollingTransport">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
```

```
      <source name="SignalR.Transports.TransportHeartBeat">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
    </switches>
    <sharedListeners>
      <add name="SignalR-Transports"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="transports.log.txt" />
        <add name="SignalR-Bus"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="bus.log.txt" />
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>
</configuration>
```

When you run the application in Visual Studio, you can view the logs in the **Output** window.

How to call client methods and manage groups from outside the Hub class

To call client methods from a different class than your Hub class, get a reference to the SignalR context object for the Hub and use that to call methods on the client or manage groups.

The following sample `StockTicker` class gets the context object, stores it in an instance of the class, stores the class instance in a static property, and uses the context from the singleton class instance to call the `updateStockPrice` method on clients that are connected to a Hub named `StockTickerHub`.

```
// For the complete example, go to
// http://www.asp.net/signalr/overview/getting-started/tutorial-server-broadcast-with-aspnet-
signalr
// This sample only shows code related to getting and using the SignalR context.
public class StockTicker
{
    // Singleton instance
    private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(
        () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>()));
```

```
    private IHubContext _context;

    private StockTicker(IHubContext context)
    {
        _context = context;
    }

    // This method is invoked by a Timer object.
    private void UpdateStockPrices(object state)
    {
        foreach (var stock in _stocks.Values)
        {
            if (TryUpdateStockPrice(stock))
            {
                _context.Clients.All.updateStockPrice(stock);
            }
        }
    }
```

If you need to use the context multiple-times in a long-lived object, get the reference once and save it rather than getting it again each time. Getting the context once ensures that SignalR sends messages to clients in the same sequence in which your Hub methods make client method invocations. For a tutorial that shows how to use the SignalR context for a Hub, see **Server Broadcast with ASP.NET SignalR (/signalr/overview/signalr-20/getting-started-with-signalr-20/tutorial-server-broadcast-with-signalr-20)** .

Calling client methods

You can specify which clients will receive the RPC, but you have fewer options than when you call from a Hub class. The reason for this is that the context is not associated with a particular call from a client, so any methods that require knowledge of the current connection ID, such as `Clients.Others`, or `Clients.Caller`, or `Clients.OthersInGroup`, are not available. The following options are available:

- All connected clients.

```
context.Clients.All (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.all(v=vs.111).aspx)
.addContosoChatMessageToPage(name, message);
```

- A specific client identified by connection ID.

```
context.Clients.Client (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.client(v=vs.111).aspx)
(connectionID).addContosoChatMessageToPage(name, message);
```

- All connected clients except the specified clients, identified by connection ID.

```
context.Clients.AllExcept (http://msdn.microsoft.com/en-us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.allexcept(v=vs.111).aspx) (connectionId1,
```

```
connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group.

```
context.Clients.Group (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.group(v=vs.111).aspx)
(groupName).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except specified clients, identified by connection ID.

```
Clients.Group (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.hubs.hubconnectioncontext.group(v=vs.111).aspx) (groupName,
connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

If you are calling into your non-Hub class from methods in your Hub class, you can pass in the current connection ID and use that with `Clients.Client`, `Clients.AllExcept`, or `Clients.Group` to simulate `Clients.Caller`, `Clients.Others`, or `Clients.OthersInGroup`. In the following example, the `MoveShapeHub` class passes the connection ID to the `Broadcaster` class so that the `Broadcaster` class can simulate `Clients.Others`.

```
// For the complete example, see
// http://www.asp.net/signalr/overview/signalr-20/getting-started-with-signalr-20/tutorial-server-
broadcast-with-signalr-20
// This sample only shows code that passes connection ID to the non-Hub class,
// in order to simulate Clients.Others.
public class MoveShapeHub : Hub
{
    // Code not shown puts a singleton instance of Broadcaster in this variable.
    private Broadcaster _broadcaster;

    public void UpdateModel(ShapeModel clientModel)
    {
        clientModel.LastUpdatedBy = Context.ConnectionId;
        // Update the shape model within our broadcaster
        _broadcaster.UpdateShape(clientModel);
    }
}

public class Broadcaster
{
    public Broadcaster()
    {
        _hubContext = GlobalHost.ConnectionManager.GetHubContext<MoveShapeHub>();
    }

    public void UpdateShape(ShapeModel clientModel)
    {
        _model = clientModel;
        _modelUpdated = true;
    }
```

```
    // Called by a Timer object.
    public void BroadcastShape(object state)
    {
        if (_modelUpdated)
        {
            _hubContext.Clients.AllExcept(_model.LastUpdatedBy).updateShape(_model);
            _modelUpdated = false;
        }
    }
}
```

Managing group membership

For managing groups you have the same options as you do in a Hub class.

- Add a client to a group

```
context.Groups.Add (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.igroupmanager.add(v=vs.111).aspx) (connectionID, groupName);
```

- Remove a client from a group

```
context.Groups.Remove (http://msdn.microsoft.com/en-
us/library/microsoft.aspnet.signalr.igroupmanager.remove(v=vs.111).aspx) (connectionID, groupName);
```

How to customize the Hubs pipeline

SignalR enables you to inject your own code into the Hub pipeline. The following example shows a custom Hub pipeline module that logs each incoming method call received from the client and outgoing method call invoked on the client:

```
public class LoggingPipelineModule : HubPipelineModule
{
    protected override bool OnBeforeIncoming(IHubIncomingInvokerContext context)
    {
        Debug.WriteLine("=> Invoking " + context.MethodDescriptor.Name + " on hub " +
context.MethodDescriptor.Hub.Name);
        return base.OnBeforeIncoming(context);
    }
    protected override bool OnBeforeOutgoing(IHubOutgoingInvokerContext context)
    {
        Debug.WriteLine("<= Invoking " + context.Invocation.Method + " on client hub " +
context.Invocation.Hub);
        return base.OnBeforeOutgoing(context);
    }
}
```

The following code in the *Startup.cs* file registers the module to run in the Hub pipeline:

```
public void Configuration(IAppBuilder app)
{
    GlobalHost.HubPipeline.AddModule(new LoggingPipelineModule());
    app.MapSignalR();
}
```

There are many different methods that you can override. For a complete list, see **HubPipelineModule Methods (http://msdn.microsoft.com/en-us/library/jj918633(v=vs.111).aspx)** .

This article was originally created on June 10, 2014

Author Information

Tom Dykstra – Tom Dykstra is a Senior Programming Writer on Microsoft's Web Platform & Tools Content Team...

Patrick Fletcher – Patrick Fletcher is a former programmer-writer on the ASP.NET team.

Comments (14)