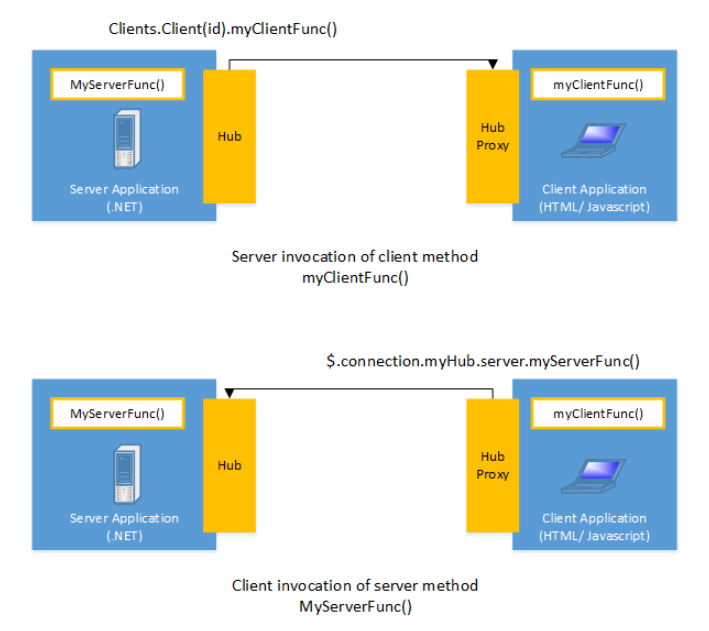
**SignalR**

ASP.NET SignalR is a library for ASP.NET developers that simplifies the process of adding real-time web functionality to applications. Real-time web functionality is the ability to have server code push content to connected clients instantly as it becomes available, rather than having the server wait for a client to request new data.

SignalR provides a simple API for creating server-to-client remote procedure calls (RPC) that call JavaScript functions in client browsers (and other client platforms) from server-side .NET code.



SignalR handles connection management automatically, and lets you broadcast messages to all connected clients simultaneously, like a chat room. You can also send messages to specific clients. The connection between the client and server is persistent, unlike a classic HTTP connection, which is re-established for each communication.

SignalR supports "server push" functionality, in which server code can call out to client code in the browser using Remote Procedure Calls (RPC), rather than the request-response model common on the web today.

SignalR applications can scale out to thousands of clients using Service Bus, SQL Server or [Redis](http://redis.io/).

**SignalR and WebSocket**

SignalR uses the new WebSocket transport where available and falls back to older transports where necessary. While you could certainly write your app using WebSocket directly, using SignalR means that a lot of the extra functionality you would need to implement is already done for you.

**Transports and fallbacks**

SignalR is an abstraction over some of the transports that are required to do real-time work between client and server. A SignalR connection starts as HTTP, and is then promoted to a WebSocket connection if it is available. WebSocket is the ideal transport for SignalR, since it makes the most efficient use of server memory, has the lowest latency, and has the most underlying features (such as full duplex communication between client and server), but it also has the most stringent requirements.

**HTML 5 transports**

**WebSocket**

WebSocket is the only transport that establishes a true persistent, two-way connection between client and server. However, WebSocket also has the most stringent requirements.

**Server Sent Events**, also known as EventSource

**Transport selection process**

1. If the browser is Internet Explorer 8 or earlier, Long Polling is used.
2. If JSONP is configured (that is, the jsonp parameter is set to true when the connection is started), Long Polling is used.
3. If a cross-domain connection is being made (that is, if the SignalR endpoint is not in the same domain as the hosting page), then WebSocket will be used if the following criteria are met:
   * The client supports CORS (Cross-Origin Resource Sharing).
   * The client supports WebSocket
   * The server supports WebSocket

If any of these criteria are not met, Long Polling will be used..

1. If JSONP is not configured and the connection is not cross-domain, WebSocket will be used if both the client and server support it.
2. If either the client or server do not support WebSocket, Server Sent Events is used if it is available.

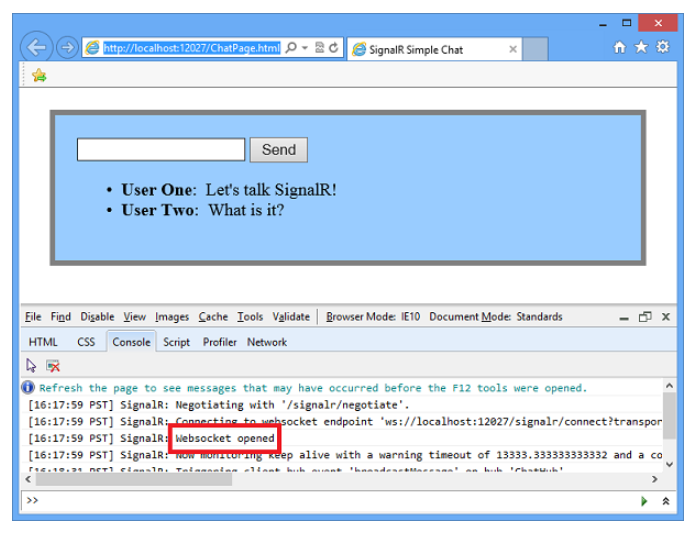
**Monitoring transports**

You can determine what transport your application is using by enabling logging on your hub, and opening the console window in your browser.

To enable logging for your hub's events in a browser, add the following command to your client application:

$.connection.hub.logging = true;

With the console open and logging enabled, you'll be able to see which transport is being used by SignalR.



### **Specifying a transport**

The following code snippet demonstrates trying WebSocket, and failing that, going directly to Long Polling.

connection.start({ transport: ['webSockets','longPolling'] });

The string constants for specifying transports are defined as follows:

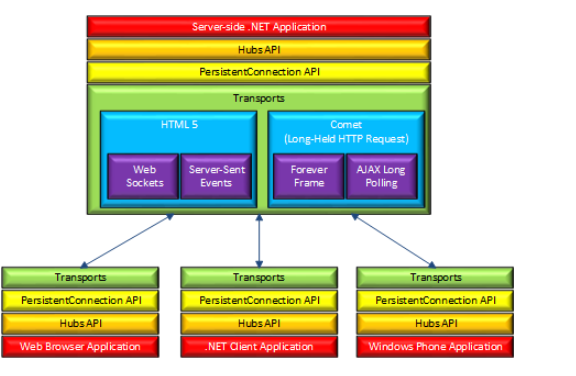
* webSockets
* foreverFrame
* serverSentEvents
* longPolling

### **Connections and Hubs**

The SignalR API contains two models for communicating between clients and servers: Persistent Connections and Hubs.

A Connection represents a simple endpoint for sending single-recipient, grouped, or broadcast messages. The Persistent Connection API (represented in .NET code by the PersistentConnection class) gives the developer direct access to the low-level communication protocol that SignalR exposes. Using the Connections communication model will be familiar to developers who have used connection-based APIs such as Windows Communication Foundation.

A Hub is a more high-level pipeline built upon the Connection API that allows your client and server to call methods on each other directly. SignalR handles the dispatching across machine boundaries as if by magic, allowing clients to call methods on the server as easily as local methods, and vice versa.



**SignalR Hubs**

In the code sample the **ChatHub** class derives from the **Microsoft.AspNet.SignalR.Hub** class. Deriving from the **Hub** class is a useful way to build a SignalR application. You can create public methods on your hub class and then access those methods by calling them from scripts in a web page.

In the chat code, clients call the **ChatHub.Send** method to send a new message. The hub in turn sends the message to all clients by calling **Clients.All.addNewMessageToPage**.

### **SignalR and jQuery**

The essential tasks in the code are creating a reference to the auto-generated proxy for the hub, declaring a function that the server can call to push content to clients, and starting a connection to send messages to the hub.

var chat = $.connection.chatHub;

Then a callback function is created