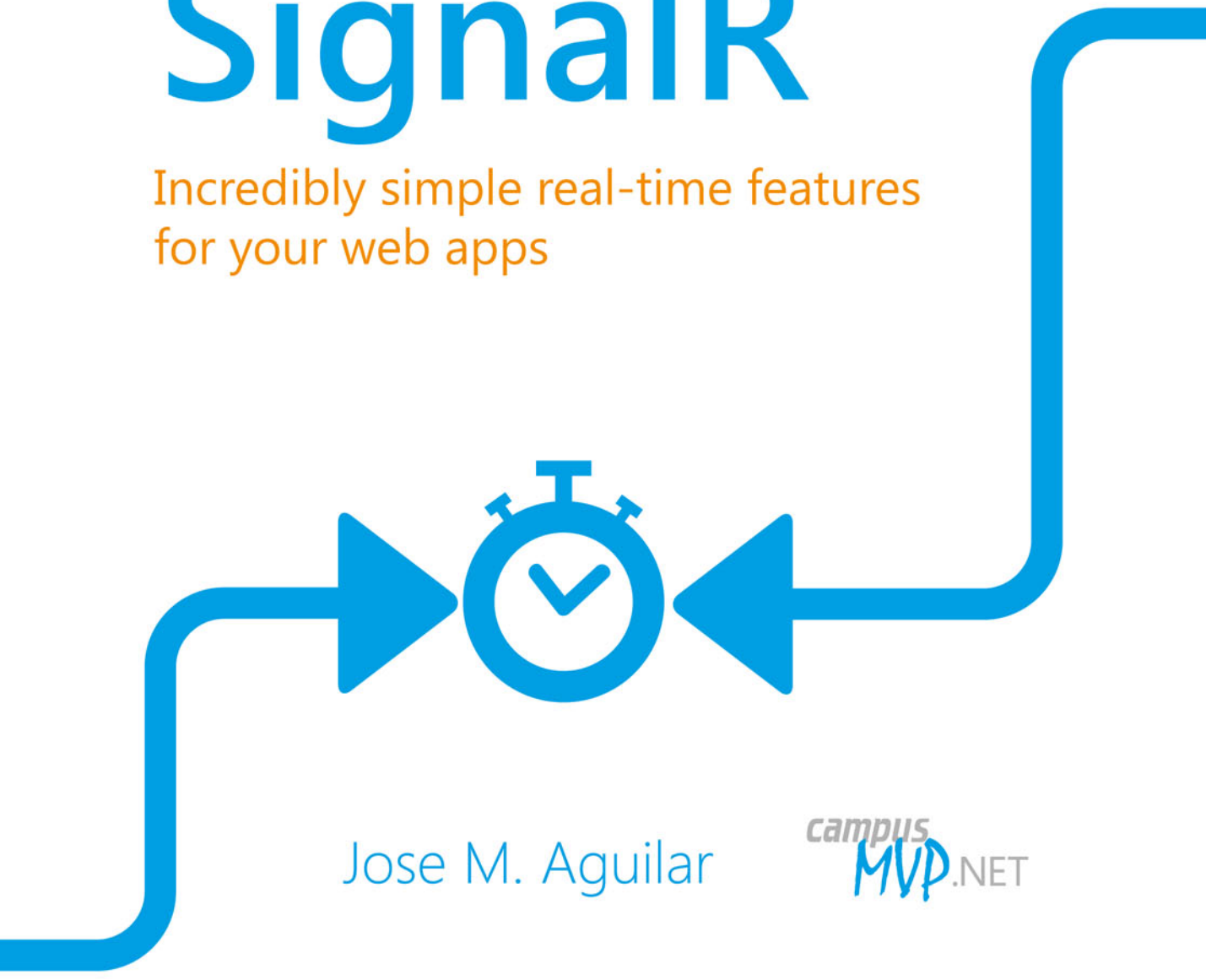
ASP.NET SignalR

Incredibly simple real-time features for your web apps



Jose M. Aguilar

campus MVP.NET

An application that combines internet, asynchrony, and multiple users cooperating and interacting at the same time always deserves a "wow!".

At some point, we have all doubtlessly been amazed by the interactivity which some modern web systems can offer. For instance, when we are editing a document online using Google Docs and another user also accesses it, we can see that they have entered and follow the changes that they are making on the document in real time. Even in a more everyday scenario such as a simple web chat, the messages being typed by our friend just appear as if by magic. Both systems use the same type of solution: **asynchronous data exchange** between the server and the clients **in real time**.

Enter the amazing world of SignalR, a free Open Source library that you can use to add features like these to your own apps.

Throughout its pages, you'll learn how to implement impressive features of this kind using SignalR. It first presents a brief review of the problems that arise when developing real-time multiuser applications and the HTTP protocol limitations to support these types of systems. Next, the book introduces SignalR, describing its main features and the different abstraction levels which it allows above the underlying protocols. By implementing different real-world examples, you'll understand its bases and receive insight into how to use this framework in your own projects. Finally, you'll learn how to host SignalR in any kind of web application (not tied to any specific technology) and how to consume its real-time services from practically any type of system (including desktop apps).

Your apps will never be the same!

ISBN 978-84-939659-7-6

campus MVP.NET

ASP.NET SignalR

Incredibly simple real-time features for your web apps

Jose M. Aguilar

ASP.NET SIGNALR - INCREDIBLY SIMPLE REAL-TIME FEATURES FOR YOUR WEB APPS

ISBN: 978-84-939659-7-6

Contents

CHAPTER 1

Introduction

1.- INTERNET, ASYNCHRONY, MULTIUSER... WOW!

An application that combines internet, asynchrony, and multiple users cooperating and interacting at the same time always deserves a "wow!". At some point, we have all doubtlessly been amazed by the interactivity which some modern web systems can offer, such as Facebook, Twitter, Gmail, Google Docs or many others, where we receive updates almost in real time, without having to reload the page.

For instance, when we are editing a document online using Google Docs and another user also accesses it, we can see that they have entered and follow the changes that they are making on the document. Even in a more everyday scenario such as a simple web chat, the messages being typed by our friend just appear as if by magic. Both systems use the same type of solution: **asynchronous data transfer between the server and the clients in real time**.

Throughout these pages, we will learn how to implement impressive features of this kind using SignalR, a framework which will facilitate our task to the point of making it trivial.

For this, we will first present a brief review of the problems that we find when developing real-time multiuser applications. We will quickly look at HTTP operation and its limitations for supporting this type of systems, and we will introduce the *Push* concept. We will also describe the standards that are being prepared by W3C and IETF, as well as techniques that we can currently use for implementing Push on HTTP.

Next, we will introduce SignalR, describing its main features and the different levels of abstraction which it allows over the underlying protocols. We will implement different examples which will help us understand its bases and give us insight on how we can use this framework in our projects.

Finally, we will describe how SignalR is independent of Web environments, which means that it can be hosted in any type of application and its real-time services can be consumed from practically any type of system.

The Author

JM Aguilar

Jose M. Aguilar (ASP.NET MVP)currently works as an independent consultant and developer, helping companies and institutions reach their goals using software. He also works with company developer teams providing consultancy services and support in several fields. You can follow him on Twitter.

CHAPTER 2

HTTP: You are the client, you are the boss

1.- HTTP OPERATION

HTTP (HyperText Transfer Protocol) is the language in which the client and the server of a web application speak to each other. It was initially defined in 1996[1], and the simplicity and versatility of its design are, to an extent, responsible for the success and expansion of the Web and the Internet as a whole.

Its operation is based on a **request-response schema which is always started by the client**. This procedure is often referred to as the **Pull model**: When a client needs to access a resource hosted by a server, it purposely initiates a connection to it and requests the desired information using the "language" defined by the HTTP protocol. The server processes this request, returns the resource that was asked for (which may be the contents of an existing file or the result of running a process) and the connection is instantly closed.

If the client needs to obtain a new resource, the process starts again from the beginning: A connection to the server is opened, the request for the resource is sent, the server processes it, it returns the result and then closes the connection. This happens every time we access a webpage, images or other resources that are downloaded by the browser, to give a few examples.

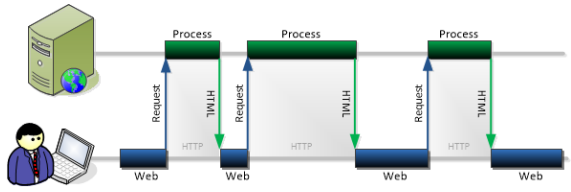


Figure 1. HTTP communication between a browser and a web server

[1] Specification of HTTP 1.0: **http://www.w3.org/Protocols/HTTP/1.0/spec.html**

As you can guess by looking at the image above, **it is a synchronous process**: After sending the request to the server, the client is left to wait, doing nothing until the response is available.

Although this operation is a classic in web systems, the HTTP protocol itself can support the needs for asynchrony of modern applications, owing to the techniques generally known as AJAX (Asynchronous JavaScript And XML).

Using AJAX techniques, the exchange of information between the client and the server can be done without leaving the current page. At any given moment, the client may initiate a connection to the server using JavaScript, request a resource and process it (for example, updating part of the page).

What is truly advantageous and has contributed to the emergence of very dynamic and interactive services, such as Facebook or Gmail, is that these operations are carried out asynchronously, that is, the user may keep using the system while the latter communicates with the server in the background to send or receive information.

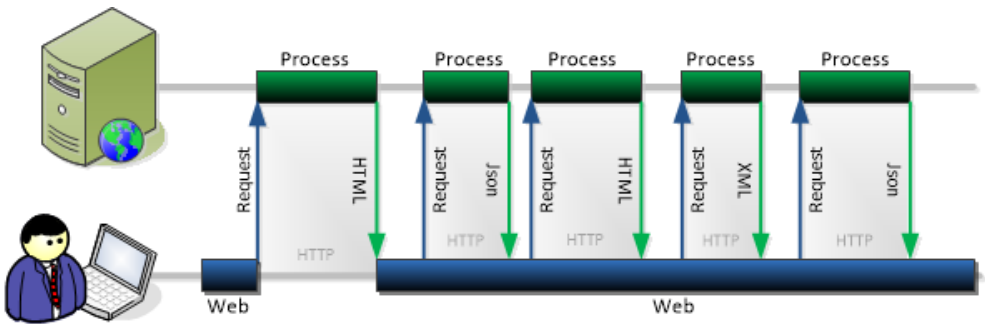


Figure 2. Ajax in a webpage

This operating schema continues to use and abide by the HTTP protocol and the client-driven request/response model. **The client is always the one to take the initiative**, deciding when to connect to the server.

However, there are scenarios where HTTP is not very efficient. With this protocol it is not easy to implement instant-messaging applications or chat-rooms, collaboration tools, multiuser online games or real-time information services, even when using asynchrony.

The reason is simple: HTTP is not oriented to real time. There are other protocols, such as the popular IRC, which are indeed focused on achieving swifter communication in order to offer more dynamic and interactive services than the ones we can obtain using Pull. In them, the server can take the initiative and send information to the client at any time, without waiting for it to request it expressly.

2.- POLLING: THE ANSWER?

As web developers, when we face a scenario where we need the server to be the one sending information to the client on its own initiative, the first solution that intuitively

comes to our minds is to use the technique known as Polling. Polling basically consists in **making periodic connections from the client to check whether there is any relevant update at the server**.

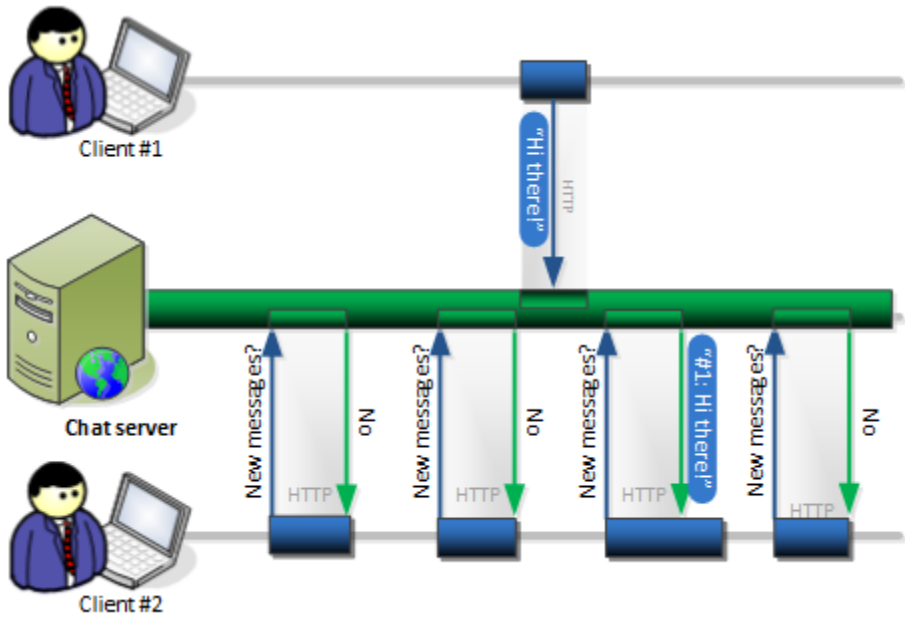


Figure 3. Polling in a chat room service

The main advantages of this solution are, first, its **easy implementation**, and second, its **universal application**: it works in every case, with all browsers and with all servers, since it does nothing more than use the standard features of HTTP. And, of course, we still use the Pull model.

However, **sometimes the price of polling is too high**. Constant connections and disconnections have a high cost in terms of bandwidth and processing at both ends of communication. And the worst part is that this cost increases proportionally to our need for faster updates. In an application providing real-time updates, it is easy to imagine the load that a server has to bear when it has thousands of users connected requesting several updates per second.

There are techniques to mitigate these problems insofar as possible. One of them is to use **adaptive periodicity**, so that the interval between queries regularly adapts to the current system load or to the probability of new updates. It is quite easy to implement and can significantly improve resource consumption in some scenarios.

There is a more conservative variant of polling, but it degrades user experience. It is the technique called *piggy backing*, which consists in not making deliberate queries from the client and, instead, taking advantage of any interaction between the user and the system to update any necessary information. To illustrate this, consider a web mail service: Instead of making periodic queries to check for the arrival of new messages,

those checks would be performed each time the user accessed a page, an email or any other feature. This may be useful in scenarios which do not require great immediacy and where the features of the system itself mean that we can be sure that the user will interact with the application frequently.

In conclusion, polling is a reasonable option despite its disadvantages when we want a solution that is easy to implement and able to be used universally and in scenarios where a very high update frequency is not required. A real-life example of its application is found in the Web version of Twitter, where polling is used to update the timeline every thirty seconds.

3.- PUSH: THE SERVER TAKES THE INITIATIVE

We already said that there are applications where the use of Pull is not very efficient. Among them, we can name instant-messaging systems, real-time collaboration toolsets, multiuser online games, information services and any kind of system where it is necessary to send information to the client right when it is generated.

For such applications, **we need the server to take the initiative** and be capable of sending information to the client exactly when a relevant event occurs, instead of waiting for the client to request it.

And this is precisely the idea behind the **Push, or Server Push, concept**. This name does not make reference to a component, a technology or a protocol: it is a concept, a communication model between the client and the server where the latter is the one taking the initiative in communications.

This concept is not new. There are indeed protocols which are Push in concept, such as IRC, the protocol which rules the operation of classic chat-room services; or SMTP, the one in charge of coordinating email sending. These were created before the term that identifies this type of communication was coined.

For the server to be able to notify events in real time to a set of clients interested in receiving them, the ideal situation would be to be able to initiate a direct point-to-point connection with them. For example, a chat-room server would keep a list with the IP addresses of the connected clients and open a socket type connection to each of them to inform them of the arrival of a new message.

However, that is technically impossible. For security reasons, it is not normally possible to make a direct connection to a client computer due to the existence of multiple intermediate levels that would reject it, such as firewalls, routes or proxies. For this reason, the customary practice is for clients to be the ones to initiate connections and not vice versa.

In order to circumvent this issue and manage to obtain a similar effect, certain techniques emerged which were based on active elements embedded in webpages (Java applets, Flash, Silverlight apps…). These components normally used sockets to open a persistent connection to the server. That is, a connection which would stay open for as long as the client was connected to the service, listening for anything that the server had to notify. When events occurred which were relevant to the client connected, the server would use this open channel to send the updates in real time.

Although this approach has been used in many Push solutions, it is tending to disappear. Active components embedded in pages are being eliminated from the Web at a dramatic speed and are being substituted for more modern, reliable and universal alternatives such as HTML5. Furthermore, long-term persistent connections based on sockets are problematic when there are intermediary elements (firewalls, proxies…) which can block these communications or close the connections after a period of inactivity. They may also pose security risks to servers.

Given the need for reliable solutions to cover this type of scenarios, both W3C and IETF—the main organisms promoting and defining protocols, languages and standards for the Internet—began to work on two standards which would allow a more direct and fluent communication from the server to the client. They are known as Websockets and Server-Sent Events, and they both come under the umbrella of the HTML5 "commercial name".

3.1.- Websockets

The Websocket standard consists in a development API, which is being defined by the W3C (World Wide Web Consortium, **http://www.w3.org**); and a communication protocol, on which the IETF (Internet Engineering Task Force, **http://www.ietf.org**) is working.

Basically, it allows the establishment of a persistent connection that the client will initiate whenever necessary and which will remain open. A **two-way channel between the client and the server is thus created**, where either can send information to the other end at any time.

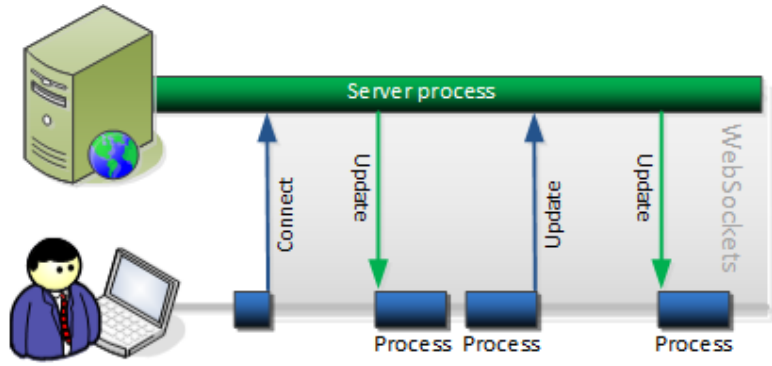


Figure 4. Operation of the Websocket standard

At the moment, the specifications of both the API and the protocol are in draft state, which means that we cannot consider the application of this technology to be universal.

We can find implementations of Websocket in many current browsers, such as IE10, Chrome and Firefox. Some only feature partial implementations (Opera, Safari) and in others, Websocket is simply not available[2].

Aside from the problem of the different implementation levels at the client side, the fact that the standard includes an independent protocol for communication (although initially negotiated on HTTP) means that changes also have to be made on some infrastructural elements, and even on servers, so that connections using Websocket are accepted.

For example, it has not been possible to use Websocket easily on Microsoft technologies up until the very latest wave of developments (IE10, ASP.NET 4.5, WCF, IIS8…), where it is at last supported natively.

From the perspective of a developer, Websocket offers a JavaScript API which is really simple and intuitive to initiate connections, send messages and close the connections when they are not needed anymore, as well as events to capture the messages received:

```
var ws = new WebSocket("ws://localhost:9998/echo");
ws.onopen = function() {
    // Web Socket is connected, send data using send()
    ws.send("Message to send");
    alert("Message is sent...");
};
ws.onmessage = function(evt) {
    var received_msg = evt.data;
    alert("Message is received...");
};
ws.onclose = function () {
    // websocket is closed.
    alert("Connection is closed...");
};
```

As you can see, the connection is opened simply by instantiating a Websocket object pointing to the URL of the service endpoint. The URL uses the ws:// protocol to indicate that it is a Websocket connection.

You can also see how easily we can capture the events produced when we succeed in opening the connection, data are received, or the connection is closed.

Without a doubt, Websocket is the technology of the future for implementing Push services in real time.

3.2.- Server-Sent Events (a.k.a. API Event Source)

Server-Sent Events, also known as API Event Source, are the second standard on which the W3 consortium is working.

[2] Source: **http://caniuse.com/websockets**

Currently, this standard is also in draft state. But this time, since it is a relatively straightforward JavaScript API and no changes are required on underlying protocols, its implementation and adoption is simpler than in the case of the Websocket standard.

In contrast with the latter, Event Source proposes the creation of a **one-directional channel from the server to the client**, but opened by the client. That is, the client "subscribes" to an event source available at the server and receives notifications when data are sent through the channel.

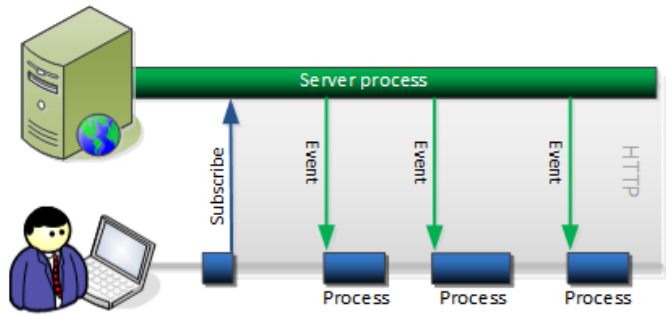


Figure 5. Operation of the Server-Sent Events standard

All communication is performed on HTTP. The only difference with respect to a more traditional connection is the use of the content-type text/event-stream in the response, which indicates that the connection is to be kept open because it will be used to send a continuous stream of events—or messages—from the server.

Implementation at the client is even simpler than the one we saw above for Websockets:

```
var source = new EventSource('/getevents');
source.onmessage = function(event) {
     alert(event.data);
};
```

As you can guess, instantiating the `EventSource` object initiates the subscription of the client to the service whose URL is provided in the constructor, and the messages will be processed in the callback function specified to that effect.

Currently, almost all browsers support this standard except for Internet Explorer, and this limits its use in real applications. Also, if we look at it from an infrastructural point of view, we find that although being based on HTTP greatly simplifies its generalization, it requires the aid of proxies or other types of intermediaries, which must be capable of interpreting the content-type used and not processing the connections in the same way as the traditional ones—avoiding, for example, buffering responses or disconnections due to timeout.

It is also important to highlight the limitations imposed by the fact that the channel established for this protocol is one-directional: **if the client needs to send data to the server, it must do so via a different connection**.

3.3.- Push today

As we have seen, standards are getting prepared to solve the classic Push scenarios, although we currently do not have enough security to use them universally.

Nevertheless, **Push is something that we need right now**. Users demand ever more interactive, agile and collaborative applications. And to develop them we must make use of techniques allowing us to achieve the immediacy of Push, but taking into account current limitations in browsers and infrastructure. And at the moment we can only obtain that by making use of the advantages of HTTP and its prevalence.

Given these premises, it is easy to find multiple conceptual proposals on the Internet, such as Comet, HTTP Push, Reverse Ajax, Ajax Push, etc., each describing solutions (sometimes coinciding) to achieve the goals desired. In the same way, we can find different specific techniques that describe how to implement Push on HTTP more or less efficiently, such as Long Polling, XHR Streaming, or Forever Frame.

We will now study two of them, Long Polling and Forever Frame, for two main reasons. First, they are the most universal ones (they work in all types of client and server systems), and second, they are used natively by SignalR, as we shall see later on, so this way we will begin to approach the objectives of this document.

3.3.1.- Long polling

This Push technique is quite similar to Polling, which we already described, but it introduces certain modifications to improve communication efficiency and immediacy.

In this case, the client also polls for updates but, unlike in Polling, if there are no data pending to be received, the connection will not be closed automatically and initiated again later. In Long Polling, the connection **remains open until the server has something to notify**:

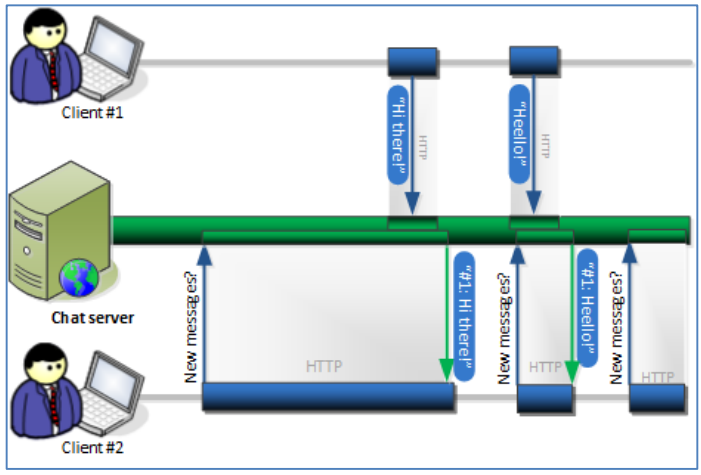


Figure 6. Long polling

The connection, which is always initiated by the client, can only be closed because of two things:

- The server sends data to the client through the connection.
- A timeout error occurs due to lack of activity on the connection.

In both cases, a new connection would be immediately established, which would again remain waiting for updates.

This connection is used exclusively to receive data from the server, so if the client needs to send information upwards it will open an HTTP connection in parallel to be used exclusively for that purpose.

The main advantage of Long Polling is the low delay in updating the client, because as soon as the server has data to update the state of the client, it will be sent through the channel that is already open, so the other end will receive it in real time.

Also, since the number of connection openings and closures is reduced, resource optimization at both ends is much higher than with Polling.

Currently, this is a widely used solution, due to its relatively simple implementation and the fact that it is completely universal. No browser-specific feature is used, just capabilities offered by HTTP.

Resource consumption with Long Polling is somewhat higher than with other techniques where a connection is kept open. The reason is that there are still many connection openings and closures if the rate of updates is high. Also, the time it takes to establish connections means that there may be some delay between notifications.

3.3.2.- Forever frame

The other technique that we are going to look at is called “Forever Frame” and uses the HTML `<iframe>` tag cleverly to obtain a permanently open connection.

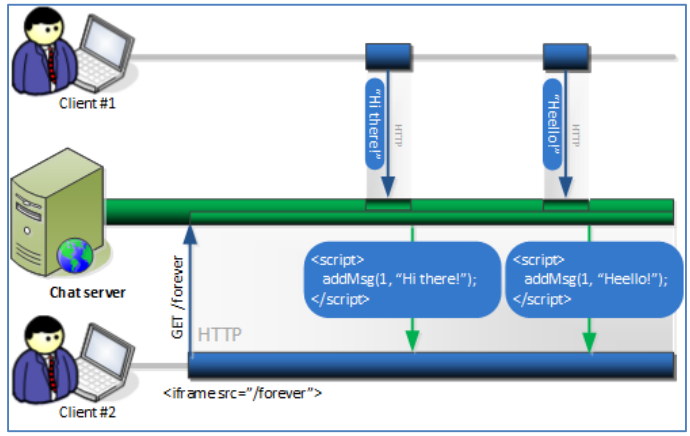


Figure 7. Forever frame

Broadly, it consists in entering an `<iframe>` tag in the page markup of the client. In the source of `<iframe>`, the URL where the server is listening is specified. The server will maintain this connection permanently open (hence the "forever" in its name) and will use it to **send updates in the form of calls to script functions defined at the client**. In a way, we might say that this technique consists in **streaming scripts which are executed at the client as they are received**.

Since the connection is kept open permanently, resources are employed more efficiently because they are not wasted in connection and disconnection processes. Thus we can practically achieve our coveted real time in the server-client direction.

Just like in the previous technique, the use of HTML, JavaScript and HTTP makes the scope of its application virtually universal, although it is obviously very much oriented towards clients that support those technologies, such as web browsers. That is, the implementation of other types of clients such as desktop applications, or other processes acting as consumers of those services, would be quite complex.

The technique is not exempt from disadvantages either. In its implementation it is necessary to take into account that there may be timeouts caused by the client, the server or an intermediary element (such as proxies and firewalls). Also, to obtain the best real-time experience, responses must be sent to the client immediately and not withheld in buffers or caches. And, because the responses would accumulate inside the *iframe*, in client memory, we may end up taking up too much RAM, so we have to "recycle" or eliminate contents periodically.

3.4.- The world needs more than just Push

Up until now, we have seen techniques that allow us to achieve Push, that is, they allow the server to be able to send information to the client asynchronously as it is generated. We have given the initiative to an element which would normally assume a passive role in communications with the client.

However, in the context of asynchronous, multiuser and real-time applications, Push is but one of the aspects that are indispensable. We need many more capabilities in order to create these always surprising systems. Here we list a few of them:

- **Managing connected users**. The server must always know which users are connected to the services, which ones disconnect and, basically, it must control all the aspects associated to monitoring an indeterminate number of clients.

- **Managing subscriptions**. The server must be capable of managing "subscriptions", or grouping clients seeking to receive specific types of messages. For example, in a chat-room service, only the users connected to a specific room should receive the messages sent to that room. This way, the delivery of information is optimized and clients do not receive information that is not relevant to them, minimizing resource waste.

- **Receiving and processing actions**. Not only should the server be capable of sending information to clients in real time, but also of receiving it and processing it on the fly.

- **Monitoring exchanges**, because we cannot guarantee that all clients connect under the same conditions. There may be connections at different speeds, line instability or occasional breakdowns, and this means that it is necessary to provide for mechanisms capable of queuing messages and managing information deliveries individually to ensure that all clients are updated.

- **Offering a flexible API**, capable of being consumed easily by multiple clients. More so nowadays, when there are a wide variety of devices from which we can access online services.

We could surely enumerate many more, but these examples are more than enough to give you an idea of the complexity inherent to developing this type of applications.

Enter SignalR…

Are you enjoying this book?

You will love this superb MVC 4 course, created and tutored by the same autor**:**

Expert Web Development with ASP.NET MVC 4

- ✓ 3 months to learn MVC **from scratch**
- ✓ Jose M. Aguilar **will answer your questions** and doubts
- ✓ Clear learning path:
  - Progressive dificulty, step by step
  - **Detailed milestones** to meet
  - Custom methodology
- ✓ The right amount of theory needed
- ✓ Precise hands-on videos
- ✓ **Downloadable code** examples and related material

The best MVC course you'll find in the market. You can bet!

Your trainer

Jose M. Aguilar (ASP.NET MVP)

Get it now!!!

http://bit.ly/learnMVC4

campus MVP .NET

Premium coached online training for busy developers

CHAPTER 3

SignalR

1.- INTRODUCING SIGNALR

The official website of the project presents SignalR as:

"ASP.NET SignalR is a new library for ASP.NET developer that makes it incredibly simple to add real-time web functionality to your applications"

It is, therefore, a framework that facilitates building interactive, multiuser and real-time web applications (though not only web applications, as we shall see later on), making extensive use of asynchrony techniques to achieve immediacy and maximum performance.

Originally it was a personal project of David Fowler and Damian Edwards, members of the ASP.NET team at Microsoft, but it is now an officially integrated product in the stack of Web technologies. The diagram below will give you a simplified idea of its position within the stack as a framework for the implementation of services:

Figure 1. Conceptual position of SignalR within the ASP.NET technology stack

As is the case of many of these technologies, the product is completely *open source* (Apache 2.0 license), but with the advantages of having the full backing and support of the Redmond giant. Its development can be tracked, and even contributed to, at GitHub[3], where one can find the source code of the framework and related projects.

At the time of writing, the latest version is 1.0.1, published in late February 2013. However, despite its young age, it is currently being used successfully in many real projects, the most visible example probably being Jabbr (**http://www.jabbr.net**), a Web-based chat-room service.

2.- WHAT DOES SIGNALR OFFER?

Basically, SignalR isolates us from low-level details, giving us the *impression* of working on a **permanently open persistent connection**. To achieve this, SignalR includes components specific to both ends of communication, which will facilitate message delivery and reception in real time between the two.

In a way that is transparent to the developer, SignalR is in charge of determining which is the best technique available both at the client and at the server (Long Polling, Forever Frame, Websockets…) and uses it to create a underlying connection and keep it continuously open, also automatically managing disconnections and reconnections when necessary. We will only see and use one permanently open connection, and SignalR will make sure that everything works correctly in the backstage. Thus, with this framework, we frequently say that we work on a **virtual persistent connection**.

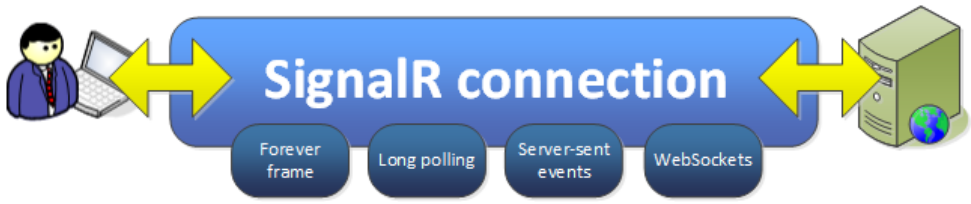


Figure 2. SignalR virtual connection

SignalR includes a set of transports—or techniques to keep the underlying connection to the server open—"as standard", and it determines which one it should use based on certain factors, such as the availability of the technology at both ends. SignalR will always try to **use the most efficient transport**, and will keep falling back until it finds one that is compatible with the context.

This decision is made automatically during an **initial stage in the communication between the client and the server, known as "negotiation"**. It is also possible to force the use of a specific transport using the client libraries of the framework.

Through the proposed abstraction, SignalR offers a **unified programming model** which is independent of the technique used in the underlying connection. As developers, we will implement our services on the virtual connection established by the

[3] SignalR code repository: **http://www.github.com/signalr**

framework. Thus we will have a unified programming model. That is, it is irrelevant to us whether Long Polling or Websockets are being used underneath to maintain the connection: **we will always use the same API**, very powerful, flexible and optimized for creating real-time applications.

Besides its ease, there is a subsequent advantage: We can also isolate ourselves from the particular aspects of the technologies and their evolution. As developers, we will focus on programming our services, and SignalR will be in charge of managing the connections and the specifics of client and server software throughout. We will not have to worry about when Websocket will be made universally available: our services will keep adapting and will begin to use Websocket when that happens.

But there is more. In fact, the features mentioned so far would just help us for a limited period—until Websocket, the most powerful technology of those proposed, was available.

SignalR also **includes a messaging bus** capable of managing data transmission and reception between the server and the clients connected to the service. That is, the server is able to keep track of its clients and detect their connections and disconnections, and it will also have mechanisms to easily send messages to all clients connected or part of them, automatically managing all issues concerning communications (different speeds, latency, errors…) and ensuring the delivery of messages.

Moreover, it includes powerful libraries on the client side that allow the consumption of services from virtually any kind of application, allowing us to manage our end of the virtual connection and send or receive data asynchronously.

In short, **in SignalR we find everything we might need to create multiuser real-time applications**.

3.- INSTALLING SIGNALR

The easiest way to include SignalR in a project is by using the NuGet package manager. This great tool makes it easy to carry out the formerly lengthy process of downloading the components, copying the binaries to the project and adding the references.

SignalR is distributed in diverse packages, where we can find server-side components and various client implementations of this type of services.

```
PM> Get-Package microsoft.aspnet.signalr -ListAvailable

Id                              Version    Description/Release Notes
--                              -------    -------------------------
Microsoft.AspNet.SignalR        1.0.1      Incredibly simple real-time
                                           web for .NET.
Microsoft.AspNet.SignalR.Cl... 1.0.1       .NET client for ASP.NET
                                           SignalR.
Microsoft.AspNet.SignalR.Core  1.0.1       Core server components for
                                           ASP.NET SignalR.
Microsoft.AspNet.SignalR.JS    1.0.1       JavaScript client for
                                           ASP.NET SignalR.
Microsoft.AspNet.SignalR.Owin  1.0.1       OWIN components for ASP.NET
                                           SignalR.
```

```
Microsoft.AspNet.SignalR.Sy... 1.0.1      Components for using
                                          ASP.NET SignalR in
                                          applications hosted on
                                          System.Web.
Microsoft.AspNet.SignalR.Utils 1.0.1      Command line utilities for
                                          ASP.NET SignalR, including
                                          performance counter insta_
                                          llation and Hub JavaScript.
Microsoft.AspNet.SignalR.Sa... 1.0.1      A simple fake stock ticker
                                          sample for ASP.NET SignalR.
```

Thus, **in a web application we will normally install the `Microsoft.AspNet.SignalR` package**, which **includes both server components and client libraries based on JavaScript**. If we want to consume our services from any type of .NET application (including WinRT, Windows Phone 8 or Silverlight 5), we must only install the package named `Microsoft.AspNet.SignalR.Client`.

It is also possible to create SignalR components directly by using the new item templates introduced in the latest update of Visual Studio, denominated "ASP.NET and Web Frameworks 2012.2", which, apart from entering basic code on which we can work, will also download and install the necessary libraries in our project.

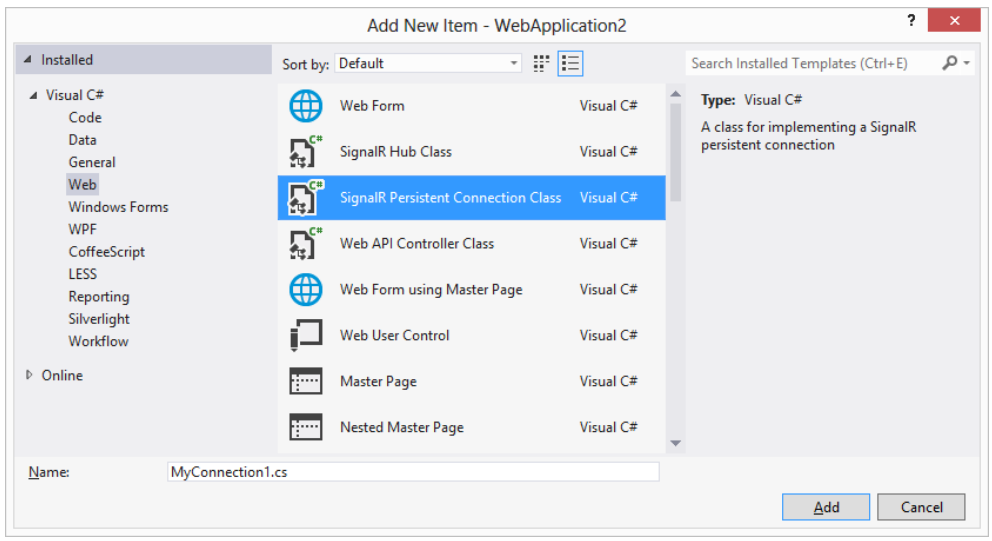


Figure 3. Adding SignalR components to a project

4.- TWO LEVELS OF ABSTRACTION

We have already described SignalR's ability to isolate us from the particularities of the connection, thus providing a smooth homogeneous development surface. However, this is not completely accurate: Actually, SignalR offers **two different levels of abstraction above the transports** used to maintain the connection with the server. In

fact, they constitute two APIs or formulas to work on the virtual connection established.

The first one, called *persistent connections*, is the lower level, closer to the reality of the connections. In fact, it creates a development surface which is quite similar to programming with sockets, although here it is done on the virtual connection established by SignalR.

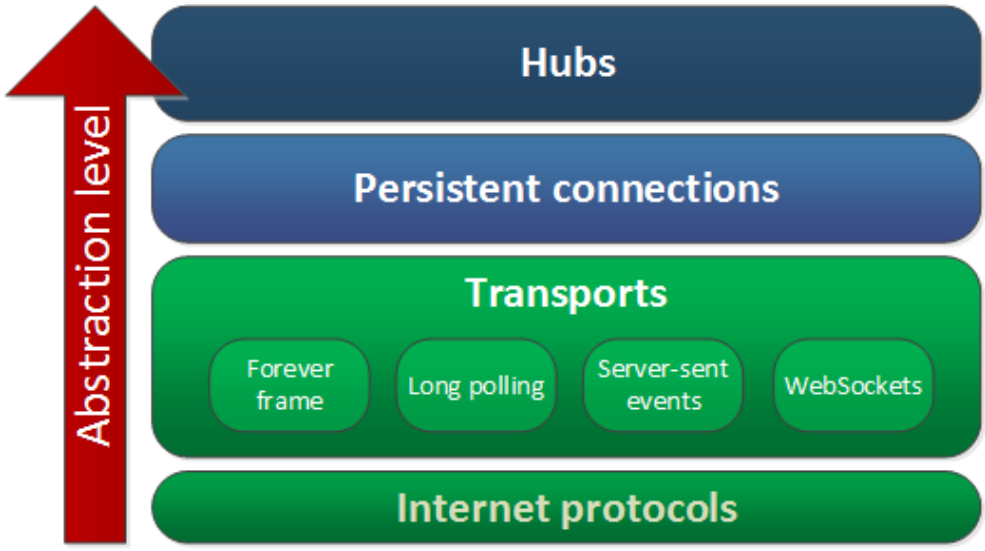


Figure 4. Abstraction levels in SignalR

The second level of abstraction, based on components called Hubs, is quite further removed from the underlying connections and protocols, offering a very imperative programming model, where the boundaries traditionally separating the client and the server melt away as if by magic.

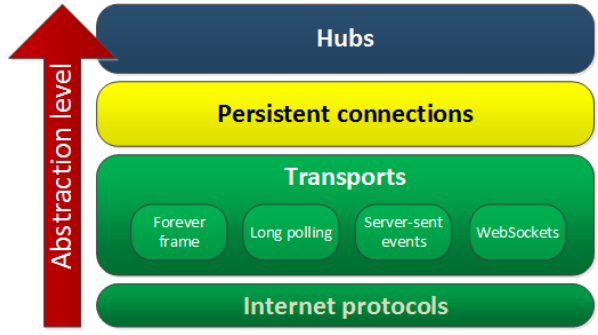
Both levels have their own scopes and will be studied in depth in the following chapters.

CHAPTER 4

Persistent connections

1.- INTRODUCTION

This is the lower-level API with which we can approach the persistent connection that SignalR offers to isolate us from the transports and complexities inherent to keeping a permanent connection open between the client and the server.



In practice, it provides us an access to the communication channel which is quite similar to the one used traditionally when working at a low abstraction level with sockets: On the server side, we can be notified when connections are opened and closed and when data are received, as well as sending information to clients. On the client side, we can open and close connections, as well as sending and receiving arbitrary data. Also, just like with sockets, messages have no format, that is, they are *raw* data—normally text strings—that we will have to know how to interpret correctly at both ends.

From the point of view of the client, its operation is very easy. We just have to initiate a connection to the server, and we will be able to use it to send data right away. We will perform the reception of information using a callback function which is invoked by the framework after its reception.

On the server side things are not very complex either. Persistent connections are classes that inherit from `PersistentConnection` and override some of the methods

that allow taking control when a relevant event occurs, such as the connection or disconnection of new clients or the reception of data. From any of them, we will be able to send information to the client that has caused the event, to all clients connected to the service, or to a group of them.

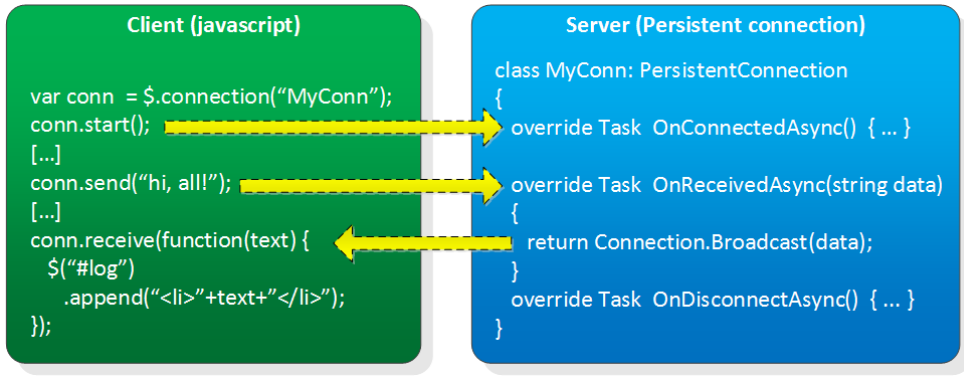


Figure 1. Conceptual implementation of persistent connections

Now we will delve into all these aspects.

2.- IMPLEMENTATION ON THE SERVER SIDE

For the clients to be able to connect to a SignalR service, first of all it is necessary to **register that service in the routing table** of the server, in a very similar way to what we do when using ASP.NET MVC or any technology based on the ASP.NET platform, that is, associating a URL to the class or component that will process the requests directed to it.

As usual, it is a process that has to be performed during application startup. The `Application_Start()` method of the global.asax file is normally a good place to do it:

```
RouteTable.Routes.MapConnection<EchoConnection>(
    name: "chatservice",
    url: "/chat"
);
```

Obviously, if we are using the file location conventions introduced by ASP.NET MVC 4, it will probably be more appropriate to register these routes in the App_Start/RouteConfig.cs file.

With this instruction, we are "maping" the URLs of the type "/chat/anything" to the class where we will implement the persistent connection, which we will call `EchoConnection` in this case. The text indicated by the `name` parameter of the route is an arbitrary name, unique in the routing tabe, and `url` identifies the *endpoint* to which the clients must connect to consume the services.

Once this first step is completed, we are in position to implement the SignalR service, which will only consist in writing a class inheriting from `PersistentConnection` (defined in the `Microsoft.AspNet.SignalR` namespace):

```
public class EchoConnection: PersistentConnection
{
    // ...
}
```

This class will be instantiated by SignalR each time an HTTP connection is opened from a client to the server, which may depend on the transport selected each time. For example, if *Websockets* is used as a transport, once the connection is established, the instance of `PersistentConnection` will remain active until the client disconnects, since it will be used both to send and receive data from the server. Contrariwise, if we use Forever Frame, an object will also be instanced each time the client sends data, since those data are transmitted using a different request from the one used to obtain "Push".

Therefore, it is generally **not a good idea to use instance members on this class to maintain system state** because the instances are created and destroyed depending on the transport used to keep the virtual connection open. For this, static members are normally used, although always appropriately protected from concurrent accesses that could corrupt their content or cause problems inherent to multithreaded systems. We also have to take into account that **using the memory for storing shared data limits the scale-out capabilities of SignalR**, since it will not be possible to distribute the load among several servers.

2.1.- Events of a Persistent Connection

The `PersistentConnection` class offers virtual methods which are invoked when certain events occur which are related to the service and the connections associated to the class, such as the arrival of a new connection, the disconnection of a client, or the reception of data. In order to take control and enter logic where we want, it will suffice to override the relevant methods.

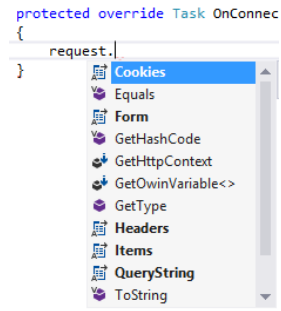
The most frequently used methods, which correspond to the events mentioned, are the following:

```
protected Task OnConnected(IRequest request, string connectionId)
protected Task OnDisconnected(IRequest request,string connectionId)
protected Task OnReceived(IRequest request, string connectionId,
                          string data)
```

First of all, note that all of them return a `Task` type object. This already gives us a clear idea of the extensive use of asynchrony capabilities present in the latest versions of the .NET platform and languages, inside which the goal is to always implement code which can be quickly and asynchronously executed, or a background task represented by a `Task` object which performs it asynchronously.

We can also observe that two parameters arrive at the methods: an `IRequest` object and a text string called `connectionId`.

The first one, `request`, allows accessing specific information on the request received, such as cookies, information about the authenticated user, parameters, server variables, etc. The `IRequest` interface is a specific SignalR abstraction which allows separating the implementation of services from ASP.NET. As we shall see, this will allow them to be hosted in any .NET application.



The second parameter, `connectionId`, is a unique identifier associated to the connection which is generated by SignalR automatically during the initial negotiation process. Although the behavior can be modified, the framework will use a GUID[4] by default:

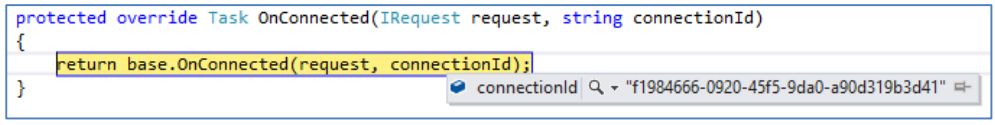


Figure 2. Value of a connectionId

We can use the connection identifier to send direct messages to specific clients or to perform any type of personalized monitoring on them.

The following code shows the implementation of a simple service, which just counts the number of users connected to it and internally displays said number on the debug console. Note the use of *thread-safe* constructions to avoid problems associated with concurrent access from different execution threads to the static variable where the value is stored. These are precautions that we must take mandatorily when implementing this kind of services:

```
public class VisitorsCountConnection: PersistentConnection
{
    private static int connections = 0;
    protected override Task OnConnected(IRequest request,
                                        string connectionId)
    {
        Interlocked.Increment(ref connections);
        Debug.WriteLine("Visitors: " + connections);
        return base.OnConnected(request, connectionId);
    }
    protected override Task OnDisconnected(IRequest request,
                                           string connectionId)
    {
        Interlocked.Decrement(ref connections);
        Debug.WriteLine("Visitors: " + connections);
        return base.OnDisconnected(request, connectionId);
    }
}
```

[4] Globally Unique IDentifier: **http://en.wikipedia.org/wiki/Globally_unique_identifier**

Other less utilized methods also exist in the `PersistentConnection` class, such as `OnReconnected()` and `OnRejoiningGroups()`. The former can be useful to take control when there is a reconnection, that is, when the client has connected to the server again after the physical connection of the transport has closed due to a timeout or any other incident. From the point of view of the virtual connection, it is still the same client and has the same identifier, thus the invocation of `OnReconnected()` instead of treating it as a new connection. The `OnRejoiningGroups()` method allows taking control when a connection is reopened after a timeout and determining which groups the connection should be reassigned to.

We can implement the treatment of the reception of data sent by a client by overriding the `OnReceived()` method of the persistent connection, where we receive a text string with the information submitted:

```
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
    // Do something interesting here
}
```

Of course, if a JSON serialized object came in this `string`, we could deserialize it directly using the Json.NET library, which will be available in our project because it is required by SignalR:

```
using Newtonsoft.Json;
// ...
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
    var message = JsonConvert.DeserializeObject<ChatMessage>(data);
    if (message.MessageType == MessageType.Private)
    {
        var text = message.Text;
        // ...
    }
    // ...
}
```

2.2.- Sending messages to clients

There are tools available to the classes that inherit from `PersistentConnection`, which allow sending information to all connected clients, to specific clients identified by their `connectionId`, or to groups of clients.

To **send a message asynchronously to all clients connected** to the service, we will use the `Connection` property to invoke the `Broadcast()` method as follows:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    // Notify all connected clients
    return this.Connection.Broadcast(
                "New connection: " + connectionId);
}

protected override Task OnDisconnect(IRequest request,
                                     string connectionId)
{
    // Notify all connected clients
    return this.Connection.Broadcast("Bye bye, " + connectionId);
}
```

In this example, each time a new client connects to the server, the notification text is sent to all connected clients (including the newcomer) through the virtual connection. And, in the same way, we make use of the `OnDisconnected()` method, by which we are informed of the disconnection of a client, to notify the rest of the users.

The parameter that we send to `BroadCast()` is an `object` type, which means that we can send any type of object, which **will be serialized to JSON automatically**:

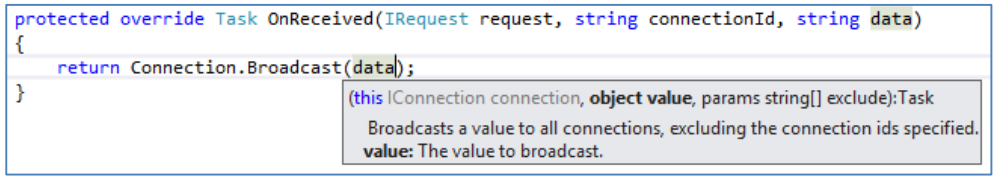


Figure 3. Broadcast receives an object type parameter

The `Broadcast()` method also accepts an optional parameter where we can specify a collection of `connectionId`'s to which the message will not be sent. The following example shows how to use this feature:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    return this.Connection.Broadcast(
             "A new user is online! Let's give them a warm welcome!",
             connectionId  // Do not notify the current user

    );
}
```

The list of identifiers excluded from the return is given in a parameter of the `params string[]` type, so they can be specified directly as parameters separated by commas or as an array of text strings:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    return this.Connection.Broadcast(
             "A new user is online! Let's give them a warm welcome!",
             new[] { connectionId }
    );
}
```

To **send messages to a specific client**, we need to know its `connectionId`. Normally, this is not a problem, because this information will be available in the methods from which we will use these calls. The following example displays a welcome message to the client initiating a connection only:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    return this.Connection.Send(connectionId,
                                "Welcome, " + connectionId);
}
```

A very interesting aspect that we already anticipated is the fact that both in the `Send()` method and in `BroadCast()` **the message to be sent to the clients is an `object` type**. This means that messages are not limited to text; it is entirely possible to send any type of object, which will be automatically serialized in JSON format before being returned to the client. This allows sending messages with a structure beyond the mere character string:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    var message = new {
                        type = MessageType.NewUser,
                        id = connectionId
                      };
    return this.Connection.Broadcast(message);
}
```

2.3.- Asynchronous event processing

As you can imagine, calls to the `Send()` or `Broadcast()` methods that we have already used in some of our examples could take too long to execute if communications between both ends are very slow, or if the number of connections is large.

For this reason, those commands are executed asynchronously, returning a `Task` object representing the task that will take care of sending in the background. Consequently, we can return the result of the call from the body of the method, as we have been doing with the code shown up until now:

```
return this.Connection.Broadcast(message);
```

We must use this very technique inside the methods of the persistent connection whenever we are to perform long tasks, and especially those requiring the use of external elements such as the access to web services or external APIs. The following example shows how to return a task that waits five seconds before sending the message received to all the users:

```
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    return
        this.Connection.Broadcast("A new user is online!")
            .ContinueWith(_ =>
                this.Connection.Send(
                           connectionId,
                          "Welcome, " + connectionId
                )
        );
}
```

2.4.- Connection groups

We have seen how SignalR allows sending messages to individual clients, identified by their `connectionId`, or to all clients connected to a service. Although these capabilities cover multiple scenarios, it would still be difficult to undertake certain tasks which require selective communication with a specific group of connections.

Imagine a chat service with different rooms. When a user enters a specific room and writes a text, ideally it would be sent only to the users present in said room. However, with the functionalities studied up to this point, the implementation of this feature would not be easy.

For this reason, **SignalR offers the possibility of grouping connections** based on whatever criteria we deem relevant. For example, in a chat, we might create a group for each room; in an online game, we could group the users competing in the same match; in a multiuser document editor similar to Google Docs, we could create a group for every document being edited.

To manage those groups, we use the `Groups` property, available in the `PersistentConnection` class and thus in all its descendants. This property is of the `IConnectionGroupManager` type and, among other things, it provides methods to add a connection identified by its `connectionId` to a group and, likewise, remove it.

The following example shows how we might interpret the commands `join <groupname>` and `leave <groupname>` coming from a client, to respectively add the connection to the group specified and remove it from it:

```
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
    var pars = data.Split(new[] {" "},
                          StringSplitOptions.RemoveEmptyEntries);

    if(pars.Length == 2 && pars[0].ToLower()=="join")
    {
        return this.Groups.Add(connectionId, pars[1]);
    }
    if (pars.Length == 2 && pars[0].ToLower() == "leave")
    {
        return this.Groups.Remove(connectionId, pars[1]);
    }
    // ...
}
```

The groups do not need to exist previously nor do they require any kind of additional managing. They are simply created when the first connection is added to them and are automatically eliminated when they become empty. And, of course, one connection can be included in as many groups as necessary.

To send information through the connections included in a group, we can use the `Send()` method as follows:

```
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
    int i;
    if ((i = data.IndexOf(":")) > -1)
    {
        var groupName = data.Substring(0, i);
        var message = data.Substring(i + 1);
        return this.Groups.Send(groupName, message);
    }
    // ...
}
```

As you can see, the above code would interpret a message such as “signalr:hello!” by sending the text “hello!” to all clients connected to the “signalr” group.

Unfortunately, for reasons related to the structural scalability of SignalR, we cannot obtain information about the groups, such as the list of connections included in them, not even how many there are. Neither can we know, a priori, what groups are active at a given moment. If we want to include these aspects in our applications, we must implement them ourselves, outside SignalR.

3.- CLIENT-SIDE IMPLEMENTATION

SignalR offers many client libraries with the purpose that practically any kind of application may use the virtual connection provided by the framework. Although later on we will see examples of implementations of other types of clients, for the moment

we will deal with creating clients from the Web using JavaScript, mainly because it is easy and widely done.

In any case, the concepts and operating philosophy that we are going to see are common to all clients and project types.

3.1.- Initiating the connection using the Javascript client

An important aspect to underline is that this client is completely and solely based on JavaScript, so it can be used in any kind of Web project: Webforms, ASP.NET MVC, Webpages, PHP, or even from pure HTML pages.

In fact, in order to access services in real time from an HTML page, it will suffice to reference jQuery—since the client is implemented as a plug-in of this renowned framework—and then the jquery.signalR-1.0.1.js library (or its corresponding minimized version). We can find both of them in the /Scripts folder when we install the Microsoft.AspNet.SignalR.JS package from NuGet, or the complete Microsoft.AspNet.SignalR package, which includes both the client and server components for web systems.

```
<script src="Scripts/jquery-1.9.1.min.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
```

Once they are referenced, we can begin to work with SignalR from the client side, although first we must open a connection to the server. For this, it is necessary for the client to know the URL by which the persistent connection is accessible. As we said before, it is registered in the routing table during application startup. Here you can see an example:

```
RouteTable.Routes.MapConnection<ChatConnection>(
    name: "chatservice",
    url: "realtime/chat"
);
```

Given the definition of the previous route, which maps the URL /realtime/chat to the persistent connection implemented in the `ChatConnection` class, the following code shows how to create and open a connection using JavaScript:

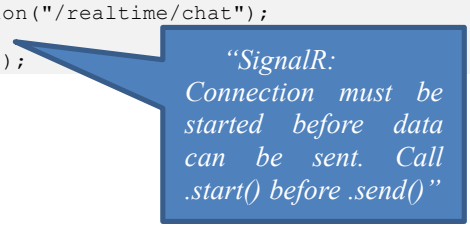
```
<script type="text/javascript">
    $(function() {
        var connection = $.connection("/realtime/chat");
        connection.start();
        // ...
    });
</script>
```

As you can see, this call is being entered in the page initialization, following the customary pattern used to develop with jQuery. Although it does not necessarily have

to be this way, this technique ensures that the code will be executed when the page has loaded completely and the DOM is ready to be operated on.

The call to the `start()` method is asynchronous, so the execution of the script will continue on the next line even if the connection has not been opened yet. This particular detail is very important, because we will not be able to use the connection until it has been established. If this does not happen, an exception will be thrown with a very clear message:

```
var connection = $.connection("/realtime/chat");
connection.start();
connection.send("Hi there!");
```



Fortunately, this method has overloads which allow us to specify a callback function that will be executed once the connection is open and the process of transport negotiation with the server has been completed:

```
var connection = $.connection("/realtime/chat");
connection.start(function() {
    // Connection established!
    connection.send("Hi there!");
});
```

It is also possible to use the well-known Promise[5] pattern and the implementation of jQuery based on `Deferred`[6] objects to take control when the connection has been successful, as well as in the event that an error has occurred:

```
var connection = $.connection("/realtime/chat");
connection.start()
    .done(function() {
        connection.send("Hi there!"); // Notify other clients
    })
    .fail(function() {
        alert("Error connecting to realtime service");
    });
```

Once the connection is established, we can begin to send and receive information using the mechanisms that are described below.

From this point onwards, we can also close the connection explicitly, by invoking the `connection.close()` method, or obtain the identifier of the current connection, generated by the server during the negotiation phase, through the `connection.id` property.

[5] The Promise pattern: **http://wiki.commonjs.org/wiki/Promises**
[6] The Deferred object in jQuery: **http://api.jquery.com/category/deferred-object/**

3.2.- Support for cross-domain connections

As standard, SignalR includes support for connections with a different server from the one that has served the script currently being executed, that is normally not allowed for security reasons. This type of requests, called *cross-domain*, requires the use of some kind of special technique to avoid this restriction which is usual in browsers. Examples of those techniques are JSONP[7] or, only in some browsers, the use of the CORS[8] specification.

In any case, cross-domain requests must be explicitly enabled from the server for each persistent connection. This can be done in the definition of the access URL, sending a `ConnectionConfiguration` in which we have to set the `EnableCrossDomain` property to true:

```
RouteTable.Routes.MapConnection<EchoConnection>(
       name: "chatservice",
       url: "/chat",
       configuration: new ConnectionConfiguration {
                              EnableCrossDomain = true
       }
);
```

SignalR will automatically detect when we are making a connection to an external domain, and will try to use the best possible mechanism to establish communication with it. In the worst case scenario, Long Polling with JSONP will be used, a configuration which we may also expressly indicate, should we want to, when initiating the connection from the client. At that moment, it is possible to send an object with settings that allow refining this process:

```
var connection = $.connection("http://localhost:3701/realtime/chat");
connection.start({ jsonp: true }, function() {
    alert(connection.transport.name);
});
```

3.3.- Sending messages

As you can probably guess by looking at the previous code, to send information to the server from the JavaScript client we will use the `send()` method available in the `connection` object, which represents the connection created before.

This method accepts the information to be sent in the form of a text string as a parameter. This same text will be received in the `data` parameter of the `OnReceived()` method of the server, as we previously saw:

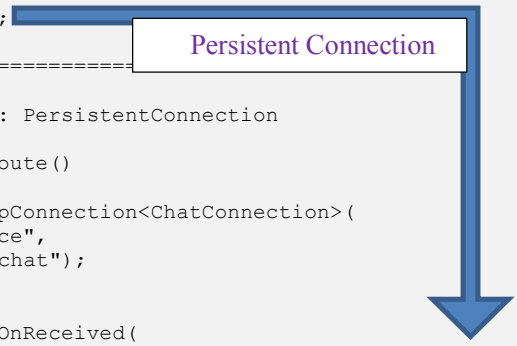
[7] JSONP (JSON with Padding). **http://en.wikipedia.org/wiki/JSONP**

[8] CORS (Cross-Origin Resource Sharing): **http://en.wikipedia.org/wiki/Cross-origin_resource_sharing**

```
// ======================================================
// Client code (Javascript)
connection.send("Hi there!");

// ======================================
// Server code
public class ChatConnection : PersistentConnection
{
    public static void InitRoute()
    {
        RouteTable.Routes.MapConnection<ChatConnection>(
            name: "chatservice",
            url: "/realtime/chat");
    }

    protected override Task OnReceived(
                  IRequest request, string connectionId, string data)
    {
        // Broadcasts the message to all connected clients
        return this.Connection.Broadcast(data);
    }
}
```



If we want to send an object instead of a text string, we need to serialize it previously as a JSON *string*:

```
$("#buttonSend").click(function () {
    var obj = {
        messageType: 1, // Broadcast message, type = 1
        text:        $("#text").val(),
        from:        $("#currentUser").val(),
    };
    var str = JSON.stringify(obj);
    connection.send(str);
});
```

Since what would arrive at the server would be a text string with the JSON representation of the object, in order to manipulate the data comfortably it would be necessary to deserialize the *string* and turn it into a CLR object, as we have already seen:

```
// Server code
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
    var message = JsonConvert.DeserializeObject<ChatMessage>(data);
    if (message.MessageType == MessageType.Broadcast)
    {
        return this.Connection.Broadcast(
                "Message from "+message.From +
                ": " + message.Text);
    }
    // ...
    return null;
}
```

Although the `send()` method is expected to adhere to the Promise pattern in the future, this is currently not so, and there is no direct way of knowing when the transfer process has ended or if there has been an error in its execution. Nevertheless, we could cover this scenario by using the `error()` method of the connection, where we can set the callback function to be executed when any error occurs on it:

```
var connection = $.connection("/realtime/chat");
connection.start();
connection.error(function (err) {
    alert("Uups! It seems there is a problem " +
          "contacting with the server");
});
```

3.4.- Receiving messages

The reception of data sent from the server is performed at the JavaScript client by registering the callback that will be executed each time information is received. This registration is performed by calling the `received()` method of the object that represents the connection with the server and supplying it the function for data handling:

```
connection.received(function (msg) {
    $("#contents").append("<li>" + msg + "</li>");
});
```

As you can see, as a parameter, this function receives the data sent from the server in the form of a directly usable object. In contrast with sending, where we were the ones responsible for serialization and deserialization of data at both ends, here SignalR will be in charge of that task.

Thus, if a character string has been sent from the server, we can obtain it and process it directly at the client, as in the previous example. Conversely, if structured data are sent from the server, they are automatically serialized to JSON, and in the input parameter of our callback function we will directly receive the JavaScript object ready for use:

```
// ======================================================
// Server code
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    var message = new
    {
        type = MessageType.NewUser,
        id = connectionId,
        text = "New user!"
    };
    return this.Connection.Broadcast(message);
}

// ======================================================
```

```
// Client code (Javascript)
connection.received(function (msg) {
    $("#contents").append(msg.text + ". Id: " + msg.id);
});
```

3.5.- Other events available at the client

The `connection` object has a large number of events that allow us to take control at certain moments in the life cycle of the connection if we register the callback methods that we want to be invoked when the time comes. The most interesting ones are:

- `received()` and `error()`, which we already saw and which allow us to specify the function to be executed when data are received or when an error occurs in the connection.
- `connectionSlow()`, which allows entering logic when the connection is detected to be slow or unstable.
- `stateChanged()`, invoked when the state of the connection changes.
- `reconnected()`, when there is a reconnection of the client after its connection has been closed due to timeout or any other cause.

The documentation on methods, events and properties that the connections of the JavaScript client offer us can be found in the repository of SignalR at GitHub [9].

4.- TRANSPORT NEGOTIATION

We have seen that, once the reference to the connection is obtained, the `start()` method really initiates communication with the server, thus beginning the negotiation phase in which the technologies or techniques to be used to maintain the persistent connection will be selected.

First, there will be an attempt to establish the connection using Websocket, which is the only transport that really supports full-duplex on the same channel and is therefore the most efficient one. If this is not possible, Server-Sent Events will be tried, since this transport at least offers a standard mechanism to obtain Push. If it is not possible either, the fallback process will lead to trying Forever Frame (only in Internet Explorer) and, finally, Long Polling.

This process is easily traceable using Fiddler[10]-[11] or the development tools available in our browsers, and thus we can view the main terms of the "agreement" reached by the client and the server.

[9] Documentation of Javascript client: **https://github.com/SignalR/SignalR/wiki/SignalR-JS-Client**

[10] Fiddler: **http://www.fiddler2.com**

The following screenshot shows the content negotiation process using Internet Explorer 10 as the client and IIS 8 Express as the server. They end up with a Websocket connection:

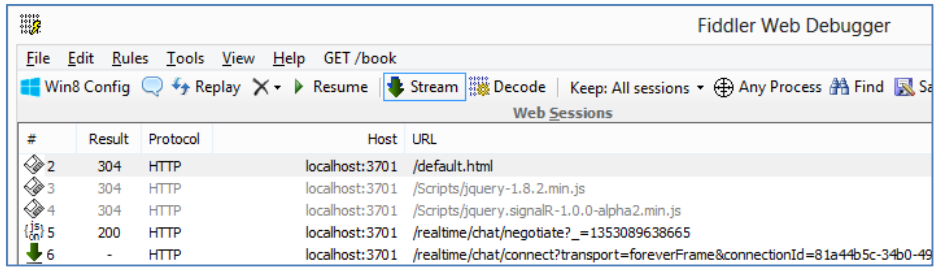


Figure 4. Internet Explorer connecting to IIS 8

Finally, we have Google Chrome trying to connect to IIS 8 on ASP.NET 4, which does not support Websocket, and so the transport selected is Server-Sent Events:

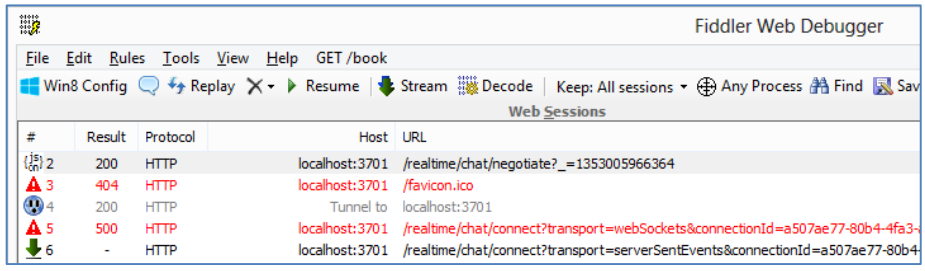


Figure 5. Google Chrome trying to connect to IIS 8 on ASP.NET 4

5.- COMPLETE EXAMPLE: TRACKING VISITORS

We will now see the code of a complete example, both on the client and server sides, with the purpose of consolidating some of the concepts addressed throughout this chapter.

Specifically, we will track the mouse of the visitors of a page and send this information to the rest of the users in real time. Thus every visitor will be able to see the position of other users' cursors on their own screen and follow their movement across it.

The following screenshot shows the system being executed on a busy page:

[11] Notes on using Fiddler with SignalR:
https://github.com/SignalR/SignalR/wiki/Using-fiddler-with-signalr

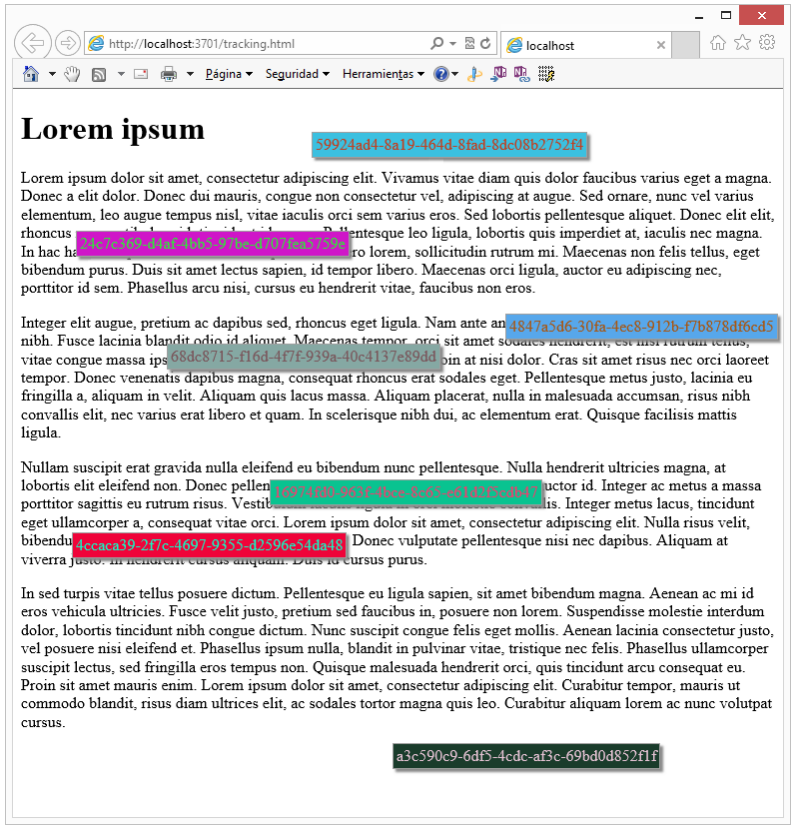


Figure 6. System for tracking users in real time operating

5.1.- Implementation on the client side

5.1.1.- HTML markup

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/jquery-1.9.1.min.js"></script>
    <script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
    <script src="Scripts/tracking.js"></script>
    <style>
        .client {
            position: absolute;
            background-color: white;
            -moz-box-shadow: 10px 10px 5px #888;
            -webkit-box-shadow: 10px 10px 5px #888;
            box-shadow: 3px 3px 3px #888;
            border: 1px solid #a0a0a0;
```

```
                padding: 3px;
            }
        </style>
    </head>
        <body>
            <h1>Lorem ipsum</h1>
            <p>Lorem ipsum dolor sit amet, […]</p>
            <p>Integer elit augue, […] </p>
        </body>
    </html>
```

5.1.2.- Scripts (Scripts/Tracking.js)

```
    /* SignalR client */
    var connection = $.connection("/tracker");
    connection.start(function () {
        startTracking();
    });
    connection.received(function (data) {
        data = JSON.parse(data);
        var domElementId = "id" + data.id;
        var elem = createElementIfNotExists(domElementId);
        $(elem).css({ left: data.x, top: data.y }).text(data.id);
    });

    function startTracking() {
        $("body").mousemove(function (e) {
            var data = { x: e.pageX, y: e.pageY, id: connection.id };
            connection.send(JSON.stringify(data));
        });
    }

    /* Helper functions */
    function createElementIfNotExists(id) {
        var element = $("#" + id);
        if (element.length == 0) {
            element = $("<span class='client' id='" + id + "'></span>");
            var color = getRandomColor();
            element.css({ backgroundColor: getRgb(color),
                          color: getInverseRgb(color) });
            $("body").append(element).show();
        }
        return element;
    }

    function getRgb(rgb) {
        return "rgb(" + rgb.r + "," + rgb.g + "," + rgb.b + ")";
    }

    function getInverseRgb(rgb) {
        return "rgb(" + (255 - rgb.r) + "," +
                        (255 - rgb.g) + "," + (255 - rgb.b) + ")";
    }

    function getRandomColor() {
        return {
```

```
            r: Math.round(Math.random() * 256),
            g: Math.round(Math.random() * 256),
            b: Math.round(Math.random() * 256),
        };
    }
```

5.2.- Implementation on the server side

5.2.1.- Persistent connection (TrackerConnection.cs)

```
    public class TrackerConnection : PersistentConnection
    {

        public static void InitRoute()
        {
            RouteTable.Routes.MapConnection<TrackerConnection>(
                name: "tracker",
                url: "/tracker");
        }

        protected override Task OnReceived(IRequest request,
                                           string connectionId,
                                           string data)
        {
            return Connection.Broadcast(data, connectionId);
        }
    }
```

5.2.2.- Startup code (Global.asax.cs)

```
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            TrackerConnection.InitRoute();

            // Other initialization code
        }
    }
```

Are you enjoying this book?

You will love this superb MVC 4 course, created and tutored by the same autor**:**

Expert Web Development with ASP.NET MVC 4

- ✓ 3 months to learn MVC **from scratch**
- ✓ Jose M. Aguilar **will answer your questions** and doubts
- ✓ Clear learning path:
  - Progressive dificulty, step by step
  - **Detailed milestones** to meet
  - Custom methodology
- ✓ The right amount of theory needed
- ✓ Precise hands-on videos
- ✓ **Downloadable code** examples and related material

The best MVC course you'll find in the market. You can bet!

Your trainer

Jose M. Aguilar (ASP.NET MVP)

Get it now!!!

http://bit.ly/learnMVC4

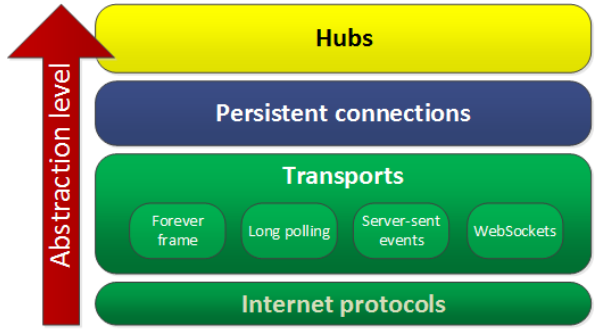
campus MVP .NET

Premium coached online training for busy developers

CHAPTER 5

Hubs

1.- INTRODUCTION



Indubitably, persistent connections provide all that we may need to create real-time multiuser applications through a really simple and intuitive API.

Nevertheless, the creators of SignalR have taken it one step further, offering **a much higher level of abstraction** above Web protocols, transports and persistent connections. Hubs use an imperative development model based on the flexibility of dynamic languages such as Javascript and C# which **creates an illusion of continuity** between these two levels which are physically detached.

Hubs are thus **the higher-level API** that we can use to access the persistent connection created by SignalR. We can create the same applications using persistent connections, but if we choose hubs it will be simpler.

Although there are two different approaches to working with hubs in SignalR, the most common model used in Web environments allows us to make direct calls between client-side and server-side methods transparently. That is, **from the client we will directly invoke methods available at the server, and vice versa**:

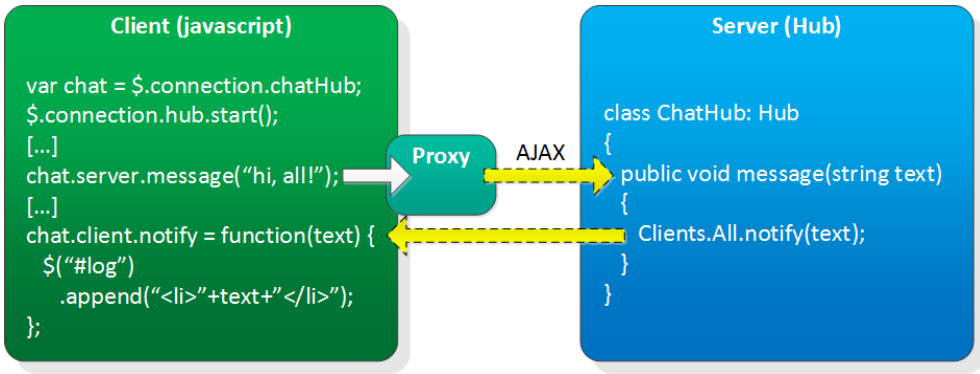


Figure 1. Conceptual implementation of services using hubs

Obviously, there is nothing magical about this. To make it possible, SignalR automatically creates proxy objects with the façade of server Hub classes, and in their methods it enters Ajax style calls to their real methods. Conversely, when the server invokes a method of the client, it is resolved by using dynamic types and a special protocol that "packages" these calls at the server and sends them to the other end through Push using the underlying transport. They then arrive at the client, where they are interpreted and executed.

The use of hubs is recommended when we need to send different types of messages with various structures between the client and the server. As you know, persistent connections operate mainly on text strings, which means that we have to perform the parsing of data manually, which can sometimes be laborious. But if we use hubs, most of the work is done by the SignalR framework itself, in exchange for a very small additional load.

2.- SERVER IMPLEMENTATION

2.1.- Route registration

Unlike persistent connections, services based on hubs **do not require specific individual entries in the routing table** to associate them with the URLs that access them, since they are all accessible through a single base URL that the framework sets by default with the value "/SignalR".

For this reason, it will only be necessary to register one entry in the routing table, for example entering the following code in the global.asax so that it is executed during system initialization:

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapHubs(); // Use the default URL
    // Other initialization code
}
```

Naturally, we can modify the default URL for accessing the services in order to adapt it to our needs or preferences:

```
RouteTable.Routes.MapHubs("/realtime", new HubConfiguration());
```

In this case, the base URL for accessing the hubs will be /realtime. Also, in the second parameter we can modify some configuration options of our hubs, such as whether the generation of dynamic proxies is enabled, whether cross-domain requests are enabled or the dependency resolution component to be used.

2.2.- Creating Hubs

Services are implemented in classes that inherit from the `Hub` class, available in the `Microsoft.AspNet.SignalR` namespace:

```
public class Echo: Hub
{
    // ...
}
```

Hubs are instantiated via calls. That is, each time a client sends a message to the server, an object of this type will be created to process it. Hubs will also be instantiated when new connections or disconnections are detected, so that we can enter customized logic in these cases, as we shall see.

The name selected for the class is an important detail since later, on the client side; it will be used to reference the service. Nevertheless, it is possible to specify a different name for that by using the `[HubName]` attribute:

```
// This service will be referenced as "EchoService" instead of "Echo"
[HubName("EchoService")]
public class Echo: Hub
{
    // ...
}
```

Inside this class we will implement the methods that will be exposed to the clients and will allow us to take control upon the arrival of messages.

2.3.- Receiving messages

When a client needs to send data to a hub, what it does is invoke one of its methods and supply it the required parameters. That is, in this case we do not have a single point for receiving information on the server side like the `OnReceived()` method of persistent connections; here we can implement as many methods as we need in the class, and

each of them will act as a possible point of entry and implementation of processing logic.

The following code shows a hub with the signature of methods which could be useful for a simple chat service:

```
public class ChatHub: Hub
{
    public void Broadcast(string text) { ... }
    public void Message(string to, string text) { ... }
    public void SendToGroup(string group, string text) { ... }
    public void ChangeNick(string newNickname) { ... }
    public void Join(string group) { ... }
    public void Leave(string group) { ... }
    // ...
}
```

Below you can see how one of these methods could be invoked from a JavaScript client using the automatically generated proxy (we will look at this in depth later on):

```
var newNick = prompt("Enter your new nickname");
hubProxy.server.changeNick(newNick); // Calls method ChangeNick()
                                     // on the server
```

Or from a generic .NET client:

```
hubProxy.Invoke("ChangeNick", "Joe"); // Calls method ChangeNick()
                                      // on the server
```

As you can see, the name of the method is used from the client to make the call, although it is possible to specify an alternative one using the `[HubMethodName]` attribute:

```
// This method will be referenced as "SendPrivate"
// instead of "Message"
[HubMethodName("SendPrivate")]
public void Message(string to, string text) { ... }
```

The arguments entered in the call from the client side are directly mapped to the parameters expected by the method on the server side, and the type conversions on both ends are managed automatically. Internally, a mechanism is used which is to a certain extent similar to the one found in the *binder* of ASP.NET MVC or WebAPI: not only will it be capable of converting values to native types used by the parameters, but it will also do the same thing with complex types like the one shown in the following example:

```
public class ChatHub: Hub
{
    // ...
    public void SendPrivate(PrivateMessage msg)
    {
        // ...
    }

}

public class PrivateMessage
{
    public string From { get; set; }
    public string To { get; set; }
    public string Message { get; set; }
    public int MsgId { get; set; }
}
```

Thus, from a JavaScript client, the call to the method might be like this:

```
hubProxy.server.sendPrivate({ from: "john",
                              to: "peter",
                              message: "Hi!",
                              msgId: "18"
});
```

Another interesting feature of these methods is that **they allow the direct return of any type of value**; SignalR will be the one in charge of serializing them to make them available to the client that made the call. In the following example we can see the implementation of a method with a return and its retrieval by a JavaScript client, where we can again observe the use of the Promise pattern to obtain the result of this operation:

```
// ======================================================
// Server code
public class Calc: Hub
{
    // ...
    public int Sum(int a, int b)
    {
        return a + b;
    }
}

// ======================================================
// Client code (Javascript)
hubProxy.server.sum(3, 4)
        .done(function(result) {
            alert(result);
        });
```

Asynchrony is also present in the implementation of the methods of the hub. If, inside a method, the process to be performed is too costly and it depends on external factors—for example, the connection to a Web service or a complex query to a database—we can return a `Task<T>` object representing the process in the background that will return a `T` type object:

```
public Task<int> Sum(int a, int b)
{
    return Task.Factory.StartNew(() =>
              {
                  Thread.Sleep(5000); // Simulate an external call
                  return a + b;
              });
}
```

2.4.- Sending messages to clients

The same concept applied to sending messages from the client to the server is also employed in the opposite direction. Through the `Clients` property, the `Hub` class offers a wide variety of tools to determine the recipients of the message and "invoke their methods" in a very simple way thanks to the flexibility provided by .NET dynamic types.

The following code shows an invocation of the `showAlert()` method in all the clients connected when one of them calls the `Alert()` method of the hub:

```
public class AlertService: Hub
{
    public void Alert(string msg)
    {
        this.Clients.All.showAlert(msg);

    }
}
```

`Clients.All` returns a reference to all connected clients in the form of a dynamic object that we can subsequently use to directly code the call to the method that we want to execute in all of them. Notice that the use of dynamic types is what makes the above code not fail in compilation despite the fact that there is no `showAlert()` method in the object on which we are making the call.

Internally, still at the server, all invocations to methods which are made on this dynamic object are captured and, following the Command[12] pattern, their specifications are entered into a data packet, which is what is really sent to the clients. When the information reaches the other end, the data will be interpreted to execute whatever logic has been implemented.

The structure sent from the server to the clients looks approximately like this:

[12] Command Pattern: **http://en.wikipedia.org/wiki/Command_pattern**

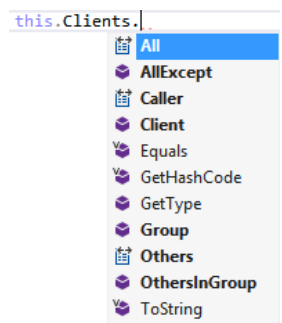
```
{
   "C":"B,3|I,0|J,0|K,0",
   "M":[{
          "H":"AlertService",
          "M":"showAlert",
          "A":["Hi, all!"]
        }]
}
```

Although this data packet has a lot of control information, we can intuitively understand that when a client receives this information it will know that it must execute the "showAlert" method of the "AlertService" hub, supplying it the text "Hi, all!" as an argument. Depending on the type of client (JavaScript, .NET, WP, etc.) it will be executed in one way or another. For example, in a JavaScript client, the packet will be translated into a call to the `client.showAlert()` method of the proxy created for the hub and its parameter will be supplied the text `"Hi, all!"`:

```
hubProxy.client.showAlert = function (msg) {
    alert(msg);
};
```

It is important to take into account that no checks are performed at the server on the name of the method or the coincidence of its signature, parameters or types. The call specification will simply be "packaged"; if we make any mistake no error will be generated: the packet will be sent to the recipients but they will not know how to interpret it, and thus the effects will not be as expected. .

If we go back to the server side, we find that besides `Clients.All` we can use various constructions to select the recipients of the calls:



- `AllExcept(connId1, connId2…)`, which indicates that the call must be sent to all clients except those whose `connectionId` are passed as parameters.
- `Caller`: the message must only be sent to the client that has invoked the method of the hub currently being executed.
- `Client(connectionId)` sends the message to the specified client only.
- `Group(groupName)` sends the message to the clients included in the specified group.
- `Others` represent all the clients connected except the one that has invoked the method being executed.
- `OthersInGroup(groupName)` is for all the clients included in the specified group except the current client—the one which invoked the method of the hub which is being executed.

```
public void MessageToGroup(string group, string text)
{
    Clients.Group(group).message(text);
    Clients.Caller.notify("Your message has been sent");
}
```

An important detail to keep in mind is that **all these methods return a `Task` type object**, which allows employing them as a return of asynchronous methods and using the capabilities offered by this type to coordinate processes executed in parallel:

```
public Task Alert(string msg)
{
    return
       (Clients.All.showAlert(msg) as Task)
       .ContinueWith( _ =>
            Clients.Caller.showAlert("Your alert has been sent")
        );
}
```

2.5.- State maintenance

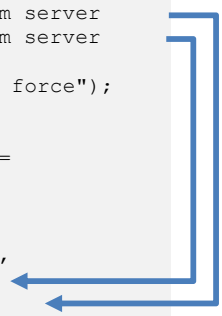
Clients automatically include state information in the calls they make to the methods of a hub. This information consists of a key-value dictionary which contains data that may be of use to either end. From the point of view of the server, this gives us the possibility of accessing these data from the body of the methods, again thanks to .NET dynamic types.

Thus on the client side we can create and use arbitrary properties that could be directly accessed from the server method invoked. In the case of a JavaScript client, these properties are created inside the `state` property of the proxy as you can see in the following code:

```
// ======================================================
// Client code (Javascript)
hubProxy.state.UserName = "Obi Wan";   // Accessible from server
hubProxy.state.MsgId = 1;              // Accessible from server

hubProxy.server.alert("I felt a great disturbance in the force");


// ======================================================
// Server code
public Task Alert(string msg)
{
    var alert = string.Format("#{0} alert from {1}: {2}",
                              Clients.Caller.MsgId,
                              Clients.Caller.UserName,
                              msg);
    return Clients.All.ShowAlert(alert);
}
```



Obviously, we can only access data relative to the client that invoked the current method; for this reason, we use `Clients.Caller` to access such data.

It is also possible to modify the values at the server. The new state will be transferred to the client in the response to the request. Notice that in this case we are applying an auto-increment operator on the `MsgId` property:

```
public Task Alert(string msg)
{
    var alert = string.Format("#{0} alert from {1}: {2}",
                              Clients.Caller.MsgId++,
                              Clients.Caller.UserName,
                              msg);
    return Clients.All.ShowAlert(alert);
}
```

The new value of the `MsgId` property will be returned to the client as part of the response to the call to the method. Upon arrival, the local value of the property will be updated.

This capability may be useful for the purpose of simplifying the signature of the methods of the hub, although we should bear in mind that **the state information that we include at the client will travel in all requests**, so we must use it carefully.

Later on we will study the mechanisms behind this interesting feature.

2.6.- Accessing information about the request context

When using hubs, in method signatures we only include the parameters that the methods need to receive from the client to perform the task assigned to them. Consequently, obtaining information from the context of the request or even the identifier of the connection that makes the call is not as direct as when we used persistent connections, where we received these data as parameters of the methods provided by the `PersistentConnection` base class.

Nevertheless, it is just as easy. To access the context data, the `Hub` class offers the `Context` property, through which it exposes properties which include `ConnectionId`, `Headers`, `QueryString` or `User`:

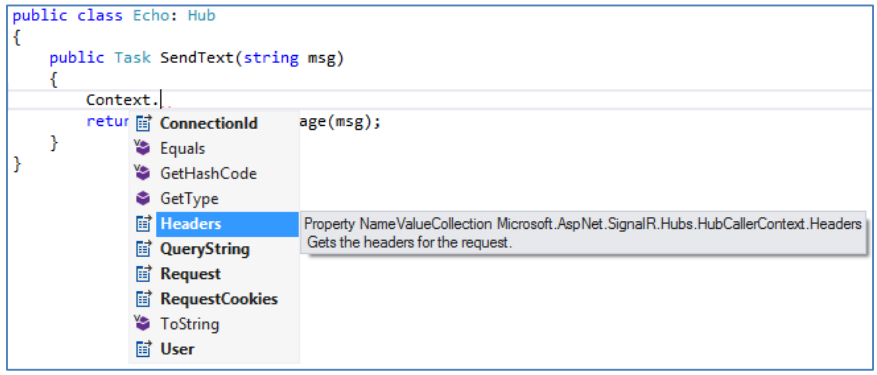


Figure 2. Members of the Context property of the Hub class

2.7.- Notification of connections and disconnections

Hubs also detect when new clients connect, as well as disconnections and reconnections, allowing us to enter logic at these points. In a similar way to persistent connections, we will achieve this by overriding three methods defined in the `Hub` class:

```
public abstract class Hub : IHub, IDisposable
{
    public virtual Task OnDisconnected() { ... }
    public virtual Task OnConnected()    { ... }
    public virtual Task OnReconnected()  { ... }
    ... // Other members of Hub
}
```

Since, as we have seen, the context information is available through the `Context` property of the `Hub`, none of these methods defines input parameters. Also, as with the `PersistentConnection` class, they all return a `Task` type object.

```
public class AlertService : Hub
{
    public override Task OnConnected()
    {
        return Clients.All.ShowAlert(
                "There is a new user ready to send alerts to!"
        );
    }
    ... // Other members of AlertsService
}
```

2.8.- Managing groups

We have already seen how to send messages to clients belonging to a group using the `Clients.Group()` and `Clients.OthersInGroup()` methods, but before this we must obviously find some way to both associate clients to SignalR groups and disassociate them.

Both operations can be performed very easily through the `Groups` property of the hubs, which gives us the `Add()` and `Remove()` properties for asynchronously adding a connection and removing it from a group:

```
public class Chat: Hub
{
    public Task Join(string groupName)
    {
        return Groups.Add(Context.ConnectionId, groupName);
    }
    public Task Leave(string groupName)
    {
        return Groups.Remove(Context.ConnectionId, groupName)
    }
```

```
        ... // Other members of the Chat
    }
```

As with persistent connections, in order not to limit the scale-out possibilities of SignalR applications, we are not given the possibility to ask what groups are available or what connections are included in each of them. If we were to need these functionalities for our applications, we would have to implement them outside the SignalR framework.

3.- CLIENT IMPLEMENTATION

The SignalR framework offers specific client libraries for the different types of systems that can consume services in real time: JavaScript, generic .NET applications, Windows Phone, WinRT, etc. They are all very similar, yet obviously adapted to the peculiarities of their respective execution environments.

Although later on we will see other examples of use, for now we will continue focused on the JavaScript client because it is easy to implement and very useful for a very natural environment for this type of systems: the Web.

3.1 JavaScript clients

There are two different ways of working with SignalR using JavaScript: with or without an automatic proxy. We have already anticipated many aspects of the first way in previous sections, and it is really the most spectacular and easy to use, since SignalR, on its own, takes care of a large portion of the work needed to achieve bidirectional communication with the server. Contrariwise, if we choose not to use a proxy, we will have to make a little more effort to reach the same goals, because we will use a syntax which is more disembodied and aseptic, quite similar to the one used in other types of clients.

We will continue studying the JavaScript client using the dynamic proxy, although later on we will also see some examples of direct use of the API without this aid.

3.2 Generating the proxy

To initiate the connection to the Hub using the JavaScript client with a dynamic proxy, we must reference the following script libraries from the page:

```
<script src="Scripts/jquery-1.9.1.min.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
<script src="/signalr/hubs"></script>
```

The first two, explained in the implementation of clients using persistent connections, are basic. SignalR's client library for JavaScript (jquery.signalR) is a

plug-in for JQuery, and for this reason both must be included in the page, and in this order.

Next we find a new reference to the script located at /Signalr/hubs. This is the URL where SignalR will be expecting connections from the clients wanting to consume the services provided by the hubs. This URL is common to all hubs in the application.

As we saw before, the base URL (without the final /hubs) can be easily modified in the call to the `MapHubs()` method that the server executes during application startup, although normally the one provided by default will be valid and we will not need to change it.

When the first request to this URL is received, SignalR will analyze the classes inheriting from `Hub` and will create a script dynamically. Inside it, we will find an object for every hub implemented at the server, which will act as a proxy of it on the client side. This process will take place only once, remaining cached for the following connections.

It is also possible to create these proxies manually using utilities provided by the developers of SignalR, downloadable through NuGet (microsoft.aspnet.signalr.utils package). It is possible to enter this process as part of the build of the solution, and it generates a single .js static file which we can reference from our pages instead of the dynamic proxy. The result will be the same, although having the file available at design time can make development easier thanks to intellisense, as well as helping us optimize downloads, using the cache more efficiently or compacting and packaging the resulting script into bundles.

In this case, it may be useful to disable the capability for automatic generation of proxies, specifying it in the configuration of the hub that we can supply during system initialization in the global.asax:

```
protected void Application_Start(object sender, EventArgs e)
{
    var config = new HubConfiguration()
                 {
                     EnableJavaScriptProxies = false
                 };

    RouteTable.Routes.MapHubs(config);

    // Other initialization code
}
```

3.3 Establishing the connection

Once this script is at the client, we can initiate the connection to the server using the following instruction:

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

The `start()` method implements the Promise pattern, so we can enter logic when the connection has been successfully completed or when an error has occurred:

```
$.connection.hub.start()
    .done(function () {
        alert("Connected");
    })
    .fail(function() {
        alert("Connection failed!");
    });
```

Note that the call to the `start()` method is independent of the hub that we want to use from the client. Opening the connection simply indicates that we are going to use a hub, but we do not necessarily have to specify which one yet.

It is also possible to specifically state the URL to access the hubs before initiating the connection, which opens the possibility of connecting with services published at servers different from the current one:

```
$.connection.hub.url = "http://myserver.com/myurl";
```

The SignalR client will detect *cross-domain* scenarios automatically and use the appropriate technique to succeed in establishing the connection. However, the Hub must have been previously configured at the server to service this type of requests:

```
// File: Global.asax.cs
var config = new HubConfiguration() { EnableCrossDomain=true };
RouteTable.Routes.MapHubs();
```

3.4 Sending messages to the server

As you know, in the dynamically generated script there will be a proxy for every hub in the server. Each hub will have a representative on the client side which we can access via the `$.connection.[hubName]` property. The following code shows how we can reference the proxy of the "AlertService" hub defined at the server:

```
var proxy = $.connection.alertService;
```

Note that to ensure coherence with the syntax and naming standards of JavaScript, **the name of the class has been converted to *camel casing* style** with the first character automatically in lower case. This detail is very important, because if this naming convention is not respected, access to the hub will not be possible:

```
var proxy = $.connection.AlertService; // Error: "AlertService"
                                       // doesn't exist
```

This conversion is performed **whenever we have not used the `[HubName]` attribute** to explicitly specify a name for the hub.

Now we shall pause for a moment to look at what has happened behind the scenes in order for us to be able to access the proxy like this. The code of a simple hub is

shown below, followed by a portion of the code generated dynamically for it by SignalR:

```
// ======================================================
// Server code
public class AlertService : Hub
{
    public Task Alert(string msg)
    {
        // Code
    }
}

// ======================================================
// Portion of client code (generated script)
...

proxies.alertService = this.createHubProxy('alertService');
proxies.alertService.client = { };
proxies.alertService.server = {
    alert: function (msg) {
               return proxies.alertService.invoke.apply(
                   proxies.alertService,
                   $.merge(["Alert"], $.makeArray(arguments))
               );
           }
};
```

In the `proxies` object that appears in the code generated is where the different objects are stored that will act as proxies of the hubs present at the server, and which are available to us through the expression `$.connection`. On this object, we see that in this case an `alertService` property is declared for storing the proxy created. This is the one that we used earlier to obtain a reference to said proxy:

```
var proxy = $.connection.alertService;
```

As you can see, inside this proxy a **`server` property** has been created, which is a faithful reproduction of it counterpart at the server: it includes **a method for each method found in the hub**. Obviously, in their implementation we will only find the code needed to perform the invocation to the server asynchronously.

By default, **the name of the methods is also converted to adapt it to the customary naming conventions in JavaScript**: the `Alert()` method of the server will be mapped to an `alert()` method at the client. However, **the name will not be converted if we use the `[HubMethodName]` attribute** to expressly state the name by which the method will be available.

Therefore, in order to invoke methods from the client which are available at the server, the only thing we have to do is use the `server` property of the proxy towards the hub and call it directly, providing it the required parameters:

```
var proxy = $.connection.alertService;
... // Open connection
proxy.server.alert("Here I am!");
```

The call to these methods implements the Promise pattern again, so we can use the constructions that we already know to take control when the call has been made successfully or when it has failed for any reason. In the first case, it is also possible to retrieve the value returned by the method executed.

```
// ======================================================
// Server code
public Task<int> Divide(int a, int b)
{
    return a / b;
}

// ======================================================
// Client code (Javascript)
var a = prompt("a?");
var b = prompt("b?");
proxy.server.divide(a, b)
    .done(function (result) {
        alert("Result: " + result);
    })
    .fail(function (desc) {
        alert("Error: " + desc);
    });
```

In this example, the function defined in the `done()` method will receive the result of the operation performed by the server as a parameter. In case of failure, the function specified in `fail()` will receive a text with the description of the error occurred (for example, "divide by zero attempt") if this option has been expressly enabled in application startup.

```
var config = new HubConfiguration() { EnableDetailedErrors = true };
RouteTable.Routes.MapHubs(config);
```

3.5 Receiving messages from the server

We have seen that when we make calls from the server to methods that exist in clients, what actually happens is that the specifications of that call are "packaged" into a data structure and sent to all its recipients using Push:

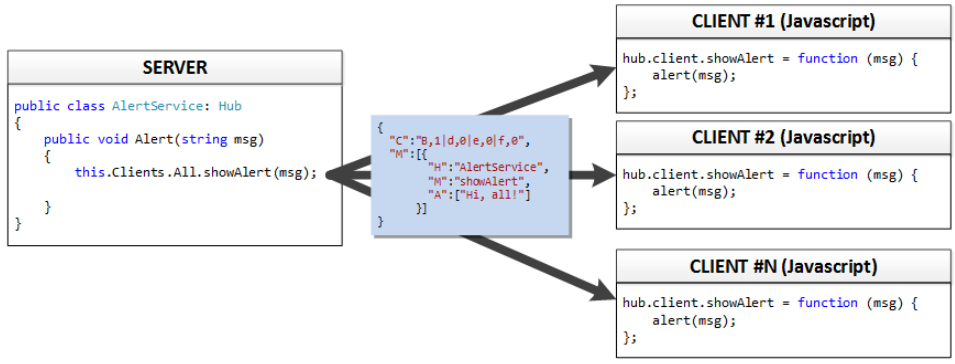


Figure 3. Calls from the server to client-side methods

From the point of view of the client, what SignalR does is interpret the data packet and invoke the relevant methods. In other words, it processes the events received.

In the JavaScript client, methods that can be "executed" from the server must be defined in the `client` property of the proxy object:

```
var alertHub = $.connection.alertService;
alertHub.client.showAlert = function (msg) {
    alert(msg);
};
alertHub.client.newUser = function (userId) {
    alert("New user with id: " + userId);
};
```

It is important to highlight that the name of the method used at the server must match the name at the client exactly, except that this time **the match is case-insensitive**. The following commands at the server will execute the logic expected at the client:

```
public override Task OnConnected()
{
    return Clients.All.NewUser(Context.ConnectionId);
}

// Is equivalent to
public override Task OnConnected()
{
    return Clients.All.newuser(Context.ConnectionId);
}
```

However, if a nonexistent method is invoked from the server, there will not be errors on either end. The server will send the clients the data packet with the command specification, and the clients will not execute any action upon its reception, since the name of the method received will not match any existing one.

3.6 Logging

The client component of SignalR for JavaScript allows registering a trace with the most relevant events that occur during the lifetime of the connection, which can be very descriptive and clarifying in order to debug the applications. To activate this trace, we just have to add the following line to the initialization code:

```
$.connection.hub.logging = true;
```

From that moment on, it will be possible to query the trace in the browser's console:

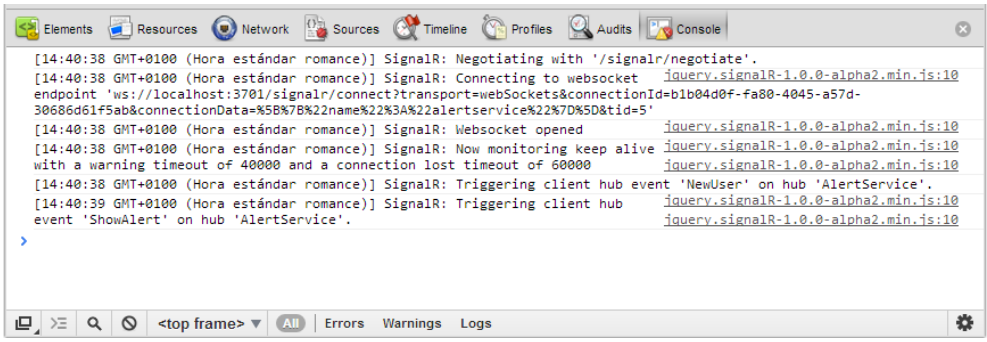


Figure 4. Log of the JavaScript client in Google Chrome

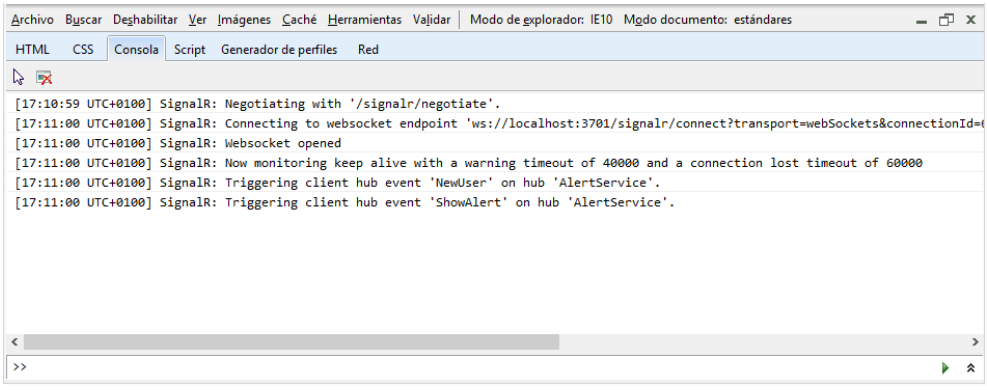


Figure 5. Log of the JavaScript client in Internet Explorer 10 Developer Tools

Actually, if we need to, we can even include custom information in this log easily:

```
$.connection.hub.start()
    .done(function () {
        $.connection.hub.log("My id: " + $.connection.hub.id);
    });
```

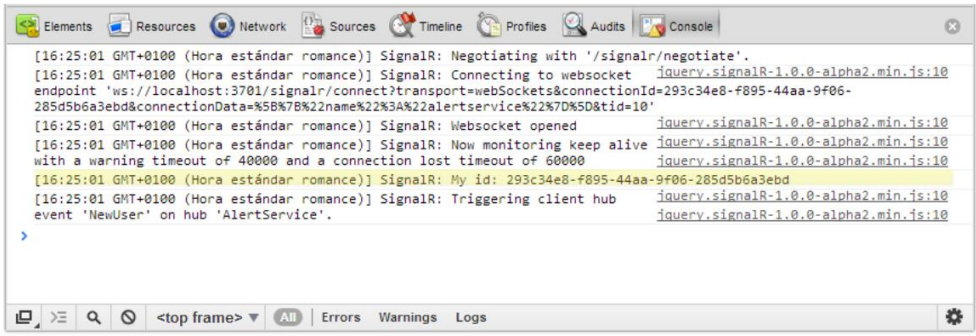


Figure 6. Custom information in the SignalR trace

> **Note**: The identifier assigned to the current client is available in the **`$.connection.hub.id`** property.

3.7 State maintenance

We have previously seen that it is possible to define variables at the client that can be queried or modified directly from the server:

```
// ======================================================
// Client code (Javascript)
hubProxy.state.UserName = "Obi Wan";    // Accessible from server
hubProxy.state.MsgId = 1;               // Accessible from server

hubProxy.server.alert("I felt a great disturbance in the force");


// ======================================================
// Server code (AlertService)
public Task Alert(string msg)
{
    var alert = string.Format("#{0} alert from {1}: {2}",
                              Clients.Caller.MsgId++,
                              Clients.Caller.UserName,
                              msg);
    return Clients.All.ShowAlert(alert);
}
```

As we said earlier and you can infer from the code, these variables must be defined in the hub's proxy, inside the object stored in its `state` property. In this case, **the name of the variable used at the server must match the one defined at the client completely**, including letter case.

The internal operation of this feature is based on the Command pattern and is quite simple. Given the above code, where we saw the setting of two state properties and a

call to a remote method at the client, the information sent in the client-server direction would be approximately as follows:

```
data={
    "H":"alertservice",
    "M":"Alert",
    "A":["I felt a great disturbance in the force"],
    "I":1,
    "S":{
         "UserName":"Obi Wan",
         "MsgId":2
        }
}
```

As you can verify, the message sent encapsulates both the specification of the invocation to be performed on the server side ("H": `hub`, "M": `method`, "A": `args`) and all the state variables ("S": `state`) that have been defined. It is therefore important to use them prudently to avoid excessive bandwidth consumption.

After processing the method at the server, the information returned to the client is more or less as follows:

```
{
   "S":{ "MsgId":3 },
   "I":"1"
}
```

In this case we see that the client is returned a structure that contains information about several aspects, including the new value of the state variables modified during server processing. Thus when the data packet reaches the client, the latter can extract the property values and modify them on its local copy, thus maintaining a consistent application state between calls.

3.8 Implementing the client without a proxy

There may be scenarios where using a proxy is inconvenient, such as applications with a large number of hubs or applications that expose a large number of operations to their clients. In such—perhaps somewhat extreme—cases, using a proxy may be too costly in terms of the bandwidth and processing capacity needed on both ends.

For these scenarios, SignalR's JavaScript client offers an additional alternative to directly communicate with hubs. In this case, the API is in fact quite similar to the one we find in other clients such as the generic client of .NET. Since its sphere of application is smaller, we will just quickly look at its main features.

To initiate a connection to a hub without using proxies, it will not be necessary to reference the /SignalR/Hubs script or whichever one we have set in the routing table as we were doing before. Here it will suffice to include jQuery and SignalR in the page:

```
<script src="Scripts/jquery-1.9.1.min.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
```

Next we must obtain a reference to the connection to the hub server and open it, in a very similar way to what we have done in previous examples:

```
var connection = $.hubConnection();
connection.start()
    .done(function () {
        // Code
    });
```

The URL assumed by default will be /SignalR/Hubs, but we can modify it by supplying it to the `$.hubConnection()` method as a parameter.

On this connection we can explicitly create a proxy, which will give us access to the server methods and will allow taking control upon reception of events sent from it.

Thus to execute a server operation we will use the `invoke()` method on the proxy:

```
var connection = $.hubConnection("/realtime");
var proxy = connection.createHubProxy("AlertService");
connection.start().done(function () {
    proxy.invoke("Alert", "I felt a great disturbance in the force");
});
```

The `invoke()` method implements, as usual, the Promise pattern, so we can take control when execution has ended, either in success or in an error.

This programming model also offers the ability to maintain state in client variables which are accessible from the server. In the same way as we saw when using dynamic proxies, we can use the `state` property for this:

```
... // Code
var proxy = connection.createHubProxy("AlertService");
proxy.state.MsgId = 1;              // Property accessible from server
proxy.state.UserName = "Obi Wan";  // Property accessible from server
```

Finally, when client methods are invoked from the server, we can capture these calls using the `On()` method as follows:

```
proxy.on("ShowAlert", function(msg) {
    alert(msg);
});
```

The first parameter supplied to this method is the name of the method "invoked" from the server. The second one is the code to be executed, in the form of an anonymous function, with the parameters that have been provided to it from the remote end.

4.- COMPLETE EXAMPLE: SHARED WHITEBOARD

In this example we will use hubs to implement a simplified version of a shared whiteboard system, with the purpose of showing how some of the concepts presented in this chapter work in practice.

Each user will be able to draw freely on the canvas, and their actions will be visible to the other connected users in real time. Furthermore, we will allow selecting the stroke color and erasing the entire whiteboard at any moment.

The result we want to obtain is shown in the following screenshot:

Figure 7. Shared whiteboard operating

4.1 Implementation on the client side

4.1.1 HTML markup

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/jquery-1.9.1.min.js"></script>
    <script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
    <script src="/signalr/hubs"></script>
    <script src="Scripts/DrawingBoard.js"></script>
    <style>
```

```
            div { margin: 3px; }
            canvas { border: 2px solid #808080; }
        </style>
    </head>
    <body>
        <div>
            <div>
                <label for="color">Color: </label>
                <select id="color">
                    <option>black</option><option>red</option>
                    <option>green</option><option>blue</option>
                    <option>yellow</option><option>pink</option>
                </select>
            </div>
            <canvas id="canvas" width="300" height="300"></canvas>
            <div>
                <button id="clear">Clear canvas</button>
            </div>
        </div>
    </body>
    </html>
```

4.1.2 Scripts (Scripts/DrawingBoard.js)

```
    $(function() {

        ////////////////////////////////////////////////////////////////
        // Standard drawing board functionalities
        ////////////////////////////////////////////////////////////////
        var canvas = $("#canvas");
        var buttonPressed = false;
        canvas
            .mousedown(function() {
                buttonPressed = true;
            })
            .mouseup(function() {
                buttonPressed = false;
            })
            .mousemove(function (e) {
                if (buttonPressed) {
                    setPoint(e.offsetX, e.offsetY, $("#color").val());
                }
            });

        var ctx = canvas[0].getContext("2d");
        function setPoint(x, y, color) {
            ctx.fillStyle = color;
            ctx.beginPath();
            ctx.arc(x, y, 2, 0, Math.PI * 2);
            ctx.fill();
        }
        function clearPoints() {
            ctx.clearRect(0, 0, canvas.width(), canvas.height());
        }

        $("#clear").click(function() {
            clearPoints();
        });
```

```
        ////////////////////////////////////////////////////////////////
        // SignalR specific code
        ////////////////////////////////////////////////////////////////

        var hub = $.connection.drawingBoard;
        hub.state.color = $("#color").val(); // Accessible from server
        var connected = false;

        // UI events
        $("#color").change(function() {
            hub.state.color = $(this).val();
        });
        canvas.mousemove(function(e) {
            if (buttonPressed && connected) {
                hub.server.broadcastPoint(e.offsetX, e.offsetY);
            }
        });
        $("#clear").click(function () {
            if (connected) {
                hub.server.broadcastClear();
            }
        });

        // Event handlers
        hub.client.clear = function() {
            clearPoints();
        };
        hub.client.drawPoint = function(x, y, color) {
            setPoint(x, y, color);
        };

        // Voila!
        $.connection.hub.start()
            .done(function () {
                connected = true;
            });

    });
```

4.2 Implementation on the server side

4.2.1 Hub (DrawingBoard.cs)

```
    public class DrawingBoard: Hub
    {
        public Task BroadcastPoint(float x, float y)
        {
            return Clients.Others.drawPoint(x, y, Clients.Caller.color);
        }
        public Task BroadcastClear()
        {
            return Clients.Others.clear();
        }
    }
```

4.2.2 Startup code (Global.asax.cs)

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapHubs();

        // … Other initialization code
    }
}
```

CHAPTER 6

Persistent connections and hubs from other processes

1.- ACCESS FROM OTHER PROCESSES

All server implementations seen up until this point had something in common: they always responded to messages from a client. Even though they are Push systems, there has always been a client initiating the procedure:

- In a chat, when a user sends a message to the server is when this message is sent to the rest of connected users.
- In the tracking system that we showed when studying persistent connections, a user's mouse movement was the action that initiated the notification of the other users.
- In the shared whiteboard, when a user drew a point or pressed the button to erase the canvas, the action was sent to the rest so that they could update their whiteboards.

Although this is what we will need in most cases, SignalR takes a step further and **allows us to send messages to clients connected to a hub or persistent connection from another process**, that is, without the need for a client to initiate the procedures. Obviously, this makes sense within the application domain, and not from external systems, even if they are in the same physical computer.

This approach can be very interesting in scenarios where there are unattended or automatically executed processes which need to send information to clients interested in receiving it in real time.

For example, there might be a background process obtaining information from a data source (such as stock quotes, meter readings, etc.) and sending it to clients connected to a hub or a persistent connection. It would also be quite easy to create a

system which registered the exceptions thrown in an ASP.NET application and sent an alert at exactly that point to a set of users previously connected to a service in real time. Or we could enter logic in the method treating a WebForms event or an ASP.NET MVC action in order to notify clients connected to a hub or persistent connection that something important for them has occurred.

Certainly, the range of possibilities that unfolds thanks to this capability of SignalR is unimaginable.

2.- USING PERSISTENT CONNECTIONS

To submit information to the clients connected to a persistent connection, we simply have to obtain a reference to said connection and use the usual methods for sending data.

In the following example we see how, from a WebForms application, we could notify connected clients that an event which they must know of has taken place, such as the elimination of an invoice in a business management system:

```
protected void BtnDeleteInvoice_Click(object sender, EventArgs e)
{
    var id = invoiceId.Text;
    _invoiceServices.DeleteInvoice(id);
    var context = GlobalHost.ConnectionManager
                            .GetConnectionContext<ErpNotifications>();

    context.Connection.Broadcast(
        "Invoice #" + id + " deleted by " + User.Identity.Name);
}
```

The `GetConnectionContext()` method used above returns an object of the `IPersistentConnectionContext` type which gives access to all the functionalities normally available from inside `PersistentConnection`: sending broadcasts directly, sending data to groups or even the actual management of groups of clients.

3.- USING HUBS

If we use hubs, the procedure is very similar to what we previously saw. The main difference is that we will obtain the reference to the `Hub` using the `GetHubContext()` o `GetHubContext<T>()` methods, as follows:

```
protected void BtnShutdown(object sender, EventArgs e)
{
   var hubcontext = GlobalHost.ConnectionManager
                              .GetHubContext<Chat>();
   hubcontext.Clients.All
                     .SendMessage("The system is shutting down!");
   ... // Code
}
```

In this case, the methods return an instance of `IHubContext`, through which we can access the functionalities for sending information and managing groups available in hubs:

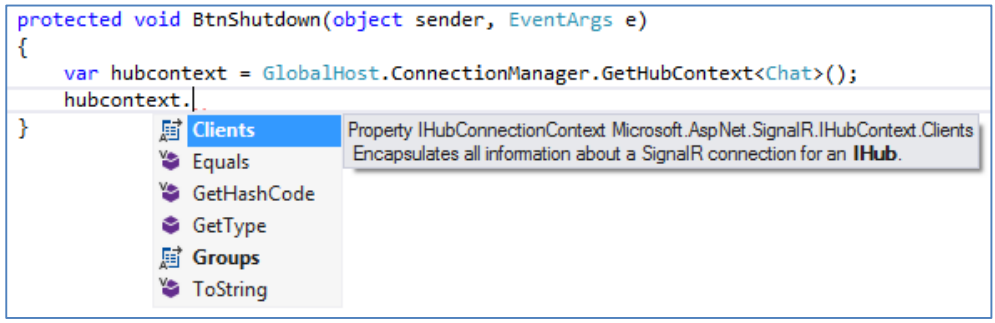


Figure 1. Members of the IHubContext interface

4.- COMPLETE EXAMPLE: PROGRESS BAR

In order to better illustrate the operation and possibilities of external access to hubs or persistent connections, we will present the complete development of a system that notifies the progress of expensive processes in real time.

In this case, we will have a client page from which we will launch an Ajax request using jQuery to a costly process written inside an ASPX page. There will be notifications of the progress in real time from inside this process, and this will be displayed as a progress bar over the page.

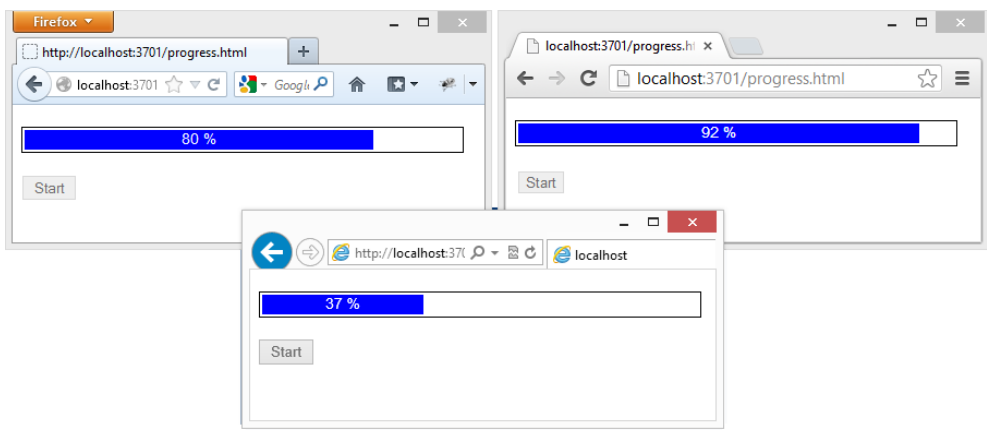


Figure 2. Progress bars operating

4.1.- Implementation on the client side

4.1.1.- HTML markup

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Progress bar</title>
    <script src="Scripts/jquery-1.9.1.min.js"></script>
    <script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
    <script src="/signalr/hubs"></script>
    <script src="Scripts/progressbar.js"></script>
    <link rel="stylesheet" href="styles/progressbar.css"/>
</head>
<body>
    <div id="progressBarContainer">
        <div id="progressBar"></div>
    </div>
    <input type="button"
           id="start" value="Start" disabled="disabled" />
    <div id="result" style="display: none;"></div>
</body>
</html>
```

4.1.2.- Styles (Styles/ProgressBar.css)

```
#progressBarContainer {
    width: 400px;
    height: 18px;
    border: 1px solid black;
    padding: 2px;
    margin: 20px 0 20px 0;
}

#progressBar {
    width: 0px;
    height: 18px;
    background-color: blue;
    margin: 0;
    overflow: hidden;
    text-align: center;
    color: white;
    font-family: arial;
    vertical-align: middle;
    font-size: 14px;
}

#result {
    border: 1px solid black;
    background-color: yellow;
    padding: 3px;
    margin-top: 10px;
}
```

4.1.3.- Scripts (Scripts/ProgressBar.js)

```
$(function () {
    var hub = $.connection.progressBar;
    hub.client.update = function (value) {
        console.log(value);
        $("#progressBar").css("width", value + "%")
                         .text(value + " %");
    };

    $("#start").click(function () {
        $(this).attr("disabled", true);
        $("#result")
            .hide("slow")
            .load("hardprocess.aspx?connId=" + $.connection.hub.id,
                  function () {
                      $(this).slideDown("slow");
                      $("#start").attr("disabled", false);
                  });
    });

    $.connection.hub.start(function () {
        $("#start").attr("disabled", false);
    });
});
```

4.2.- Implementation on the server side

4.2.1.- Hub

```
namespace ProgressBarDemo
{
    public class ProgressBar : Hub { }
}
```

Remember that we do not need any method in the hub, since the information is sent to clients through the external process, in the code shown below.

4.2.2.- Expensive process (HardProcess.Aspx)

```
<%@ Page Language="C#"
         Inherits="System.Web.UI.Page" EnableSessionState="false" %>
<%@ Import Namespace="System.Threading" %>
<%@ Import Namespace="Microsoft.AspNet.SignalR" %>
<%@ Import Namespace="ProgressBarDemo" %>
<%
    Response.Expires = -1;
    var connectionId = Request["connId"];
    var hub = GlobalHost.ConnectionManager
                        .GetHubContext<ProgressBar>();

    // Simulate a very hard process...
```

```
            for (int i = 1; i <= 100; i++)
            {
                hub.Clients.Client(connectionId).update(i);
                Thread.Sleep(150);
            }
        %>
        <p><%: DateTime.Now.ToShortTimeString() %></p>
        <p>The answer to life, the universe and everything is: 42.</p>
```

This page receives the connection identifier as a parameter. This allows it to send the progress data only to the specific client that initiated the process.

4.2.3.- Startup code (Global.asax.cs)

```
        public class Global : System.Web.HttpApplication
        {
            protected void Application_Start(object sender, EventArgs e)
            {
                RouteTable.Routes.MapHubs();
                // Other initialization code
            }
        }
```

CHAPTER 7

SignalR outside the Web

1.- SIGNALR CLIENTS

Up until now we have focused on studying the specific client component for JavaScript. However, this library (whose sphere of application is limited to web environments) is only one among those available.

It is possible to find client components for Windows Phone, Silverlight, WinRT or generic .NET clients, which broadens the range of SignalR's possibilities as a framework for the development of real-time applications in any kind of scenario.

The packages of client components currently available are the ones shown in the following table:

Table 1. SignalR clients

Client Technology	Package to install
Javascript	Microsoft.AspNet.SignalR.Client.JS
.NET 4.0 **.NET 4.5** **Silverlight 5** **WinRT** **Windows Phone 8**	Microsoft.AspNet.SignalR.Client

2.- ACCESSING SERVICES FROM NON-WEB CLIENTS

In Chapter 5 we saw the implementation of a shared whiteboard using the JavaScript client of SignalR. Here we will use the generic .NET client to implement a simplified client of the whiteboard from a console application.

Our goal is illustrated by the screenshot below. The console application is connected to the hub and represents the points drawn by the client on screen:

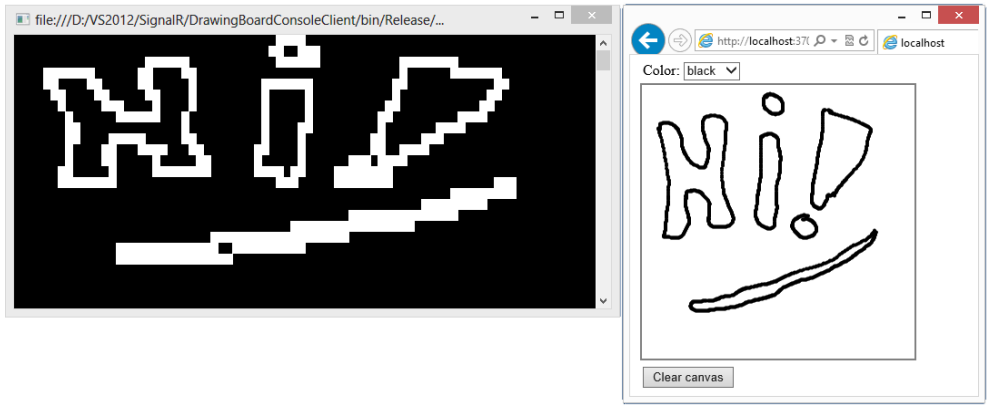


Figure 1. Console application as the client of the shared whiteboard

Once the console-type project has been created, we have to download the `Microsoft.AspNet.SignalR.Client` package using NuGet and, next, implement the client as shown below. You can see that the code is quite similar to the one used when several chapters ago we developed the JavaScript client without using the dynamic proxy, although it is obviously adapted to the characteristics and syntax of C#. We define a connection to the endpoint, we create the proxy to the Hub on it, and we implement the logic for processing the events sent from the server side:

```
static void Main(string[] args)
{
    var connection = new HubConnection("http://localhost:3701");
    var hubProxy = connection.CreateHubProxy("DrawingBoard");

    hubProxy.On("clear", () =>
           {
               Console.BackgroundColor = ConsoleColor.Black;
               Console.Clear();
           });

    hubProxy.On("drawPoint", (int x, int y) =>
           {
              int translatedx = Console.WindowWidth * x / 300;
              int translatedy = Console.WindowHeight * y / 300;

              Console.SetCursorPosition(translatedx, translatedy);
              Console.BackgroundColor = ConsoleColor.White;
              Console.Write(" ");
           });

    connection.Start();
    Console.WriteLine("Ready");
    Console.ReadKey();
}
```

3.- HOSTING SERVICES OUTSIDE ASP.NET

An interesting feature of SignalR, like other recent technologies oriented to providing services to the outside, is that it is in no way tied to ASP.NET, so virtually any .NET application (console, forms, Windows services…) can act as the host of these services.

This is possible because SignalR is aligned with the OWIN[13] standard, an open specification which describes a homogeneous interface for communication between applications and web servers which manages to eliminate dependencies between them and hence the coupling that normally exists between these two elements. Thus inside SignalR we will not find references to usual ASP.NET classes or objects because it uses the abstractions proposed by OWIN for the exchange of information with the host.

In fact, SignalR running on ASP.NET applications is simply due to the existence of a hosting component specific for the ASP.NET platform (`Microsoft.AspNet.SignalR.SystemWeb`), which is automatically installed when the main SignalR package is downloaded through NuGet, and which includes a specific OWIN adapter for the infrastructure available in `System.Web`.

To provide services from other types of applications (console, desktop, Windows services and others) we need to download a set of additional packages that contain the components in charge of listening and attending to entering HTTP requests, as well as the OWIN adapters that communicate them with SignalR.

The packages to install are the following:

- Microsoft.Owin.Hosting
- Microsoft.Owin.Host.HttpListener
- Microsoft.AspNet.SignalR.Owin

Once this is done, we could initiate a server and host persistent connections or hubs in it in an easy way, for example in a console application:

```
class Program
{
    static void Main(string[] args)
    {
        string url = "http://localhost:9876";
        using (WebApplication.Start<Startup>(url))
        {
            Console.WriteLine("Server running on {0}", url);
            Console.ReadLine();
        }
    }
}

class Startup
{
    public void Configuration(IAppBuilder app)
```

[13] Open Web Interface for .NET, **http://owin.org/**

```
        {
            app.MapHubs();
        }
    }
```

In this code you can intuitively make out the creation of a Web application in the specified URL, whose initialization, in the `Configuration()` method of the Startup class, we can use to map the hubs using the `MapHubs()` extensor and the persistent connections using `MapConnection().`

This console project would also host the hubs or persistent connections exposed to the outside through the URLs previously specified. Since SignalR is independent of the hoster, there is nothing in their code that should be especially taken into account: thus the following hub, which receives numbered Ping notifications and returns a response one second later, would be completely valid:

```
    public class PingEcho: Hub
    {
        public Task Ping(int number)
        {
            Return Task.Run(() => Thread.Sleep(1000))
                      .ContinueWith(_ =>
                           Clients.Caller.Pong("... Pong " + number)
                      );
        }
    }
```

The way these clients connect to this hub is identical to the one that we have used up to now, when the SignalR service was hosted on IIS or using ASP.NET.