

SignalR Programming in Microsoft ASP.NET



 Professional

José M. Aguilar

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2014 by Krasis Consulting S.L.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2014930486
ISBN: 978-0-7356-8388-4

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Devon Musgrave

Developmental Editor: Devon Musgrave

Project Editor: Carol Dillingham

Editorial Production: nSight, Inc.

Technical Reviewer: Todd Meister; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Copyeditor: Richard Carey

Indexer: Lucie Haskins

Cover: Twist Creative • Seattle and Joel Panchot

To my parents, for all the love and unconditional support you gave that kid who only liked computers.

And to my three girls, Inma, Inmita, and María, for putting up with me daily and yet being able to give me so much love and happiness.

—JOSÉ M. AGUILAR

Contents at a Glance

	<i>Introduction</i>	<i>xiii</i>
CHAPTER 1	Internet, asynchrony, multiuser...wow!	1
CHAPTER 2	HTTP: You are the client, and you are the boss	5
CHAPTER 3	Introducing SignalR	17
CHAPTER 4	Persistent connections	27
CHAPTER 5	Hubs	57
CHAPTER 6	Persistent connections and hubs from other threads	103
CHAPTER 7	Real-time multiplatform applications	117
CHAPTER 8	Deploying and scaling SignalR	151
CHAPTER 9	Advanced topics	181
	<i>Index</i>	<i>233</i>

Contents

Introduction	xiii
Chapter 1 Internet, asynchrony, multiuser...wow!	1
Chapter 2 HTTP: You are the client, and you are the boss	5
HTTP operations	5
Polling: The answer?	7
Push: The server takes the initiative.	8
WebSockets.	9
Server-Sent Events (API Event Source)	11
Push today.	12
The world needs more than just push	15
Chapter 3 Introducing SignalR	17
What does SignalR offer?	18
Two levels of abstraction	19
Supported platforms	20
OWIN and Katana: The new kids on the block	21
Installing SignalR.	25
Chapter 4 Persistent connections	27
Implementation on the server side	28
Mapping and configuring persistent connections.	28
Events of a persistent connection	30

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Sending messages to clients	32
Asynchronous event processing	34
Connection groups	35
The OWIN startup class	37
Implementation on the client side	38
Initiating the connection by using the JavaScript client	38
Support for older browsers	41
Support for cross-domain connections	41
Sending messages	43
Receiving messages	45
Sending additional information to the server	46
Other events available at the client	47
Transport negotiation	48
Adjusting SignalR configuration parameters	50
Complete example: Tracking visitors	51
Project creation and setup	52
Implementation on the client side	53
Implementation on the server side	54

Chapter 5 Hubs 57

Server implementation	58
Hub registration and configuration	58
Creating hubs	59
Receiving messages	60
Sending messages to clients	64
Sending messages to specific users	68
State maintenance	69
Accessing information about the request context	71
Notification of connections and disconnections	72
Managing groups	72
Maintaining state at the server	73
Client implementation	78

JavaScript clients.	79
Generating the proxy	79
Manual generation of JavaScript proxies.	81
Establishing the connection.	83
Sending messages to the server	86
Sending additional information.	89
Receiving messages sent from the server	90
Logging	91
State maintenance	92
Implementing the client without a proxy	93
Complete example: Shared drawing board	96
Project creation and setup	97
Implementation on the client side	98
Implementation on the server side.	100

Chapter 6 Persistent connections and hubs from other threads 103

Access from other threads.	103
External access using persistent connections	105
Complete example: Monitoring connections at the server	106
Project creation and setup	107
Implementing the website	108
System for tracing requests (server side).	109
System for tracing requests (client side).	111
External access using hubs	111
Complete example: Progress bar.	113
Project creation and setup	113
Implementation on the client side	114
Implementation on the server side.	115

Chapter 7 Real-time multiplatform applications 117

Multiplatform SignalR servers.	117
SignalR hosting in non-web applications.	118
SignalR hosting in platforms other than Windows	126

Multiplatform SignalR clients	129
Accessing services from .NET non-web clients	130
Consumption of services from other platforms	149
Chapter 8 Deploying and scaling SignalR	151
Growing pains	152
Scalability in SignalR.	155
Scaling on backplanes	159
Windows Azure Service Bus	159
SQL Server	165
Redis	167
Custom backplanes.	170
Improving performance in SignalR services	173
Server configuration	174
Monitoring performance	175
Chapter 9 Advanced topics	181
Authorization in SignalR	181
Access control in persistent connections.	181
Access control in hubs.	182
Client authentication	184
An extensible framework.	191
Dependency injection	196
Manual dependency injection	198
Releasing dependencies	200
Inversion of Control containers	200
Unit testing with SignalR	205
Unit testing of hubs	211
Unit testing persistent connections	215
Intercepting messages in hubs.	218
Integration with other frameworks	223
Web API	223

ASP.NET MVC	226
Knockout	227
AngularJS	230
Index	233

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

SignalR, Microsoft's latest addition to the web development technology stack, is a framework that facilitates the creation of amazing real-time applications, such as online collaboration tools, multiuser games, and live information services, whose development has traditionally been quite complex.

This book provides a complete walkthrough of SignalR development from scratch, but it will also deal with more advanced topics. The idea is that after reading it you will be familiar with the possibilities of this framework and able to apply it successfully in practice in the creation of real time systems of any size. It can also be used as a reference manual because, although not exhaustively, it includes most features of practical application in the development of SignalR systems, and it provides the bases for fully mastering them.

Who should read this book

The aim of this book is to help developers understand, know, and program SignalR-based components or applications. It can be of special interest to developers who need to make use of real-time immediacy in existing applications or who want to create new systems based on this paradigm.

Developers specializing in the back end will learn to implement real-time services that can be consumed from any client and to address scenarios such as those requiring scalability or quality improvement via unit tests. Those who are more oriented to the front end will see how they can consume real-time services and add spectacular features to their creations on the client side. Web developers, especially, will find a really simple way to break the limitations characteristic of the HTTP-based world, thanks to the use of push and the asynchrony of these solutions.

Assumptions

In this book, we will assume that the reader has a good knowledge of C# and programming within the .NET environment in general. Also, because SignalR itself and many of the examples and contents are focused on the web world, it is necessary to know the protocols on which it rests, as well as having a certain knowledge of the basic languages of these environments, such as HTML and, in particular, JavaScript.

Although not strictly necessary, the reader might benefit from some prior knowledge about development with jQuery, Windows Phone 8, or WinRT for the chapters that develop examples and contents related to them. Familiarity with techniques such as unit testing, mocking, and dependency injection to get the most out of the final chapters could also prove helpful.

Who should not read this book

Readers who do not know the .NET platform and C# will not be able to benefit from this book. If you do not have prior knowledge of JavaScript, it will be difficult to follow the book's explanations.

Organization of this book

This book is structured into nine chapters, throughout which we will go over different aspects of the development of real-time multiuser systems with SignalR, starting from scratch and all the way up to the implementation of advanced features of this framework.

Chapter 1, "Internet, asynchrony, multiuser...wow!" and Chapter 2, "HTTP: You are the client, and you are the boss," are purely introductory, and they will help you understand the technological context and the foundations on which this new framework rests.

In Chapter 3, "Introducing SignalR," we will present SignalR at a high level, showing its position in the Microsoft web development technology stack and other related concepts such as OWIN and Katana.

From this point, we will begin to look in detail at how to develop applications by using SignalR. We will dedicate Chapter 4, "Persistent connections," and Chapter 5, "Hubs," to study development from different levels of abstraction, using persistent connections and hubs. In Chapter 6, "Persistent connections and hubs from other threads," we will study how to integrate these components with other technologies within the same application, and in Chapter 7, "Real-time multiplatform applications," we will see how to implement multiplatform clients.

Chapter 8, "Deploying and scaling SignalR," will show different deployment scenarios and the scaling solutions offered by SignalR. In Chapter 9, "Advanced topics," we will find miscellanea where we will deal with more advanced aspects, such as security, extensibility, testing, and others.

Finding your best starting point in this book

Although this book is organized in such a way that it can be read from beginning to end following a path of increasing depth in the contents addressed, it can also be used as a reference by directly looking up specific chapters, depending on the level of knowledge the reader starts with and their individual needs.

Thus, for developers who are approaching SignalR for the first time, the recommendation would be to read the book from beginning to end, in the order that the chapters have been written. However, for those who are acquainted with SignalR and have already developed with it in any of its versions, it will suffice to take a quick look at the first three chapters and then to pay closer attention to the ones dedicated to development with persistent connections or hubs to find out aspects they did not know about or changes from previous versions. From there, it would be possible to go directly to resolving doubts in specific areas, such as the scalability features of the framework, implementing authorization mechanisms, or the procedure for performing unit tests on hubs.

In any case, regardless of the chapter or section, it is a good idea to download and install the related example projects, which will allow practicing and consolidating the concepts addressed.

Conventions and features in this book

This book presents information using the following conventions designed to make the information readable and easy to follow:

- Boxed elements with labels such as “Note” provide additional information or alternative methods for successfully completing a task.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you have to hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, “File | Close”) means that you should select the first menu or menu item, then the next one, and so on.

System requirements

To be able to adequately follow the examples shown in this book and practice with them, it is necessary to have, at least, the following hardware and software items:

- A computer equipped with a processor whose speed is at least 1.6 GHz (2 GHz recommended).
- 2 GB RAM (4 GB is advisable).
- A video card compatible with DirectX 9, capable of resolutions above 1024x768.
- The operating systems Windows 7 SP1, Windows 8, Windows 8.1, or Windows Server editions above 2008 R2 SP1.
- Internet Explorer 10.
- Visual Studio 2012 or above, in any of its editions. It is possible to use Express versions in most cases.
- An Internet connection.

Some examples might require that you have a system account with administrator permissions or that you install complements such as the Windows Phone SDK. In some chapters, external resources are also used, such as Windows Azure services.

Code samples

Throughout this book you can find examples, and even complete projects, to illustrate the concepts dealt with. The majority of these, as well as other additional examples, can be downloaded from the following address:

<http://aka.ms/SignalRProg/files>

Follow the instructions to download the SignalRProgramming_codesamples.zip file.



Note In addition to the code samples, your system should have Visual Studio 2012 or 2013 installed.

Notes on the version

This book has been written using version 2.0.0 of SignalR, so throughout it you will find various references to that specific version.

However, the SignalR team at Microsoft is constantly striving to improve its product, so it frequently issues software updates. The numbering of these versions is usually of the 2.0.x or 2.x.0 type. Besides corrections, these updates might include some new or improved features, but not breaking changes or significant modifications of the development APIs.

In any case, the contents of the book will still be valid after updating components to these new versions, although it will obviously be necessary to modify the existing references in the source code of the examples, especially in the case of references to script libraries.

Thus, if we have a code such as the following:

```
<script src="/scripts/jquery.signalR-2.0.0.min.js"></script>
```

after installing version 2.0.1 of SignalR, it should be changed to this:

```
<script src="/scripts/jquery.signalR-2.0.1.min.js"></script>
```

Installing the code samples

To install the code samples, just download the file indicated and decompress it into a folder in your system.

Using the code samples

After decompressing the file, a folder structure will have been created. The folders are organized in the same order as the chapters in the book, starting with Chapter 4, which is where we will begin to look at examples of code:

...

Chapter 04 – Persistent connections

Chapter 05 – Hubs

Chapter 06 – External access

...

Inside each of these folders you can find a subfolder for each sample project included. These subfolders are numbered in the order that the concepts are dealt with in the book:

...

Chapter 08 – Scaling

1-AzureServiceBus

2-SqlServer

...

Inside these folders you can find the specific solution file (*.sln) for each example. The solutions are completely independent of each other and include a fully functional example that is ready to be run (F5 from Visual Studio), although in some cases it will be necessary to make some prior adjustments in configurations. In such cases, detailed instructions are always given for this on the main page of the project or in a readme.txt file.

Acknowledgments

As trite as it might sound, a book such as this would not be possible without the collaboration of many people who have helped with their time and effort for it to become a reality, and it is only fair to dedicate them a special word of thanks.

In particular, I would like to thank my editor at campusMVP.net, Jose M. Alarcón (on Twitter at @jm_alarcon) for his involvement, his ability in the project management, coordination, and revision, as well as for his sound advice, all of which have led us here.

Javier Suárez Ruíz's (@jsuarezruiz) collaboration has also been essential, for his contributions and SignalR client implementation examples in non-web environments such as Windows Phone or WinRT.

I would like to thank Victor Vallejo, of campusMVP.net, for his invaluable help with the text.

On the part of Microsoft, I want to give thanks to the acquisitions editor, Devon Musgrave, for his interest in this project from the start, without which this book would have never been made. I also want to thank project editor Carol Dillingham for her expert work. Thanks go out to technical reviewer Todd Meister, copy editor Richard Carey, project manager Sarah Vostok of nSight, and indexer Lucie Haskins. And thanks to Sarah Hake and Jenna Boyd of O'Reilly Media for their support.

Lastly, I would like to thank Damian Edwards and David Fowler for their invaluable input. It is a privilege to have been able to benefit from the suggestions and contributions of the creators of SignalR to make this book as useful as possible.

Errata & book support

We have made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed at:

<http://aka.ms/SignalRProg/errata>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://aka.ms/tellpress>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

HTTP: You are the client, and you are the boss

HTTTP (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 1996¹, 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.

Although it is still valid in traditional web scenarios, there are others, such as real-time applications or services, for which it is quite limited.

HTTP operations

An HTTP 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 can 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 the connection is closed. This happens every time we access a webpage, images, or other resources that are downloaded by the browser, to name a few examples.

As you can guess by looking at Figure 2-1, 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.

¹ Specification of HTTP 1.0: <http://www.w3.org/Protocols/HTTP/1.0/spec.html>

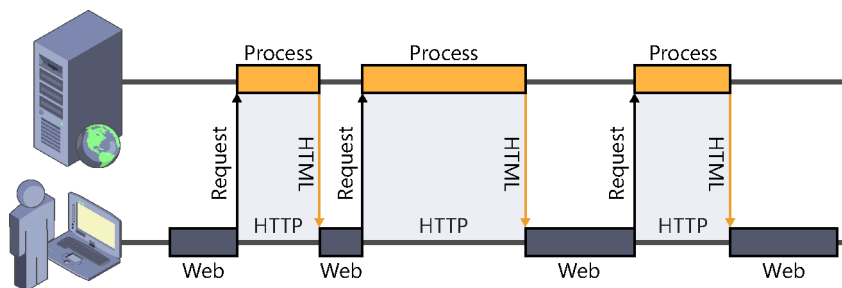


FIGURE 2-1 HTTP communication between a browser and a web server.

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, as shown in Figure 2-2, the client can initiate a connection to the server by 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 can keep using the system while the latter communicates with the server in the background to send or receive information.

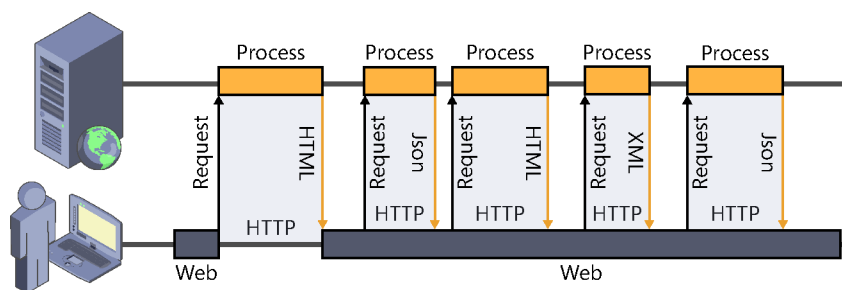


FIGURE 2-2 AJAX in a webpage.

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 in which 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 IRC², which are indeed focused on achieving swifter communication to offer more dynamic and interactive services than the ones we can obtain using pull. In those, the server can take the initiative and send information to the client at any time, without waiting for the client to request it expressly.

Polling: The answer?

As web developers, when we face a scenario in which 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, as shown in Figure 2-3.

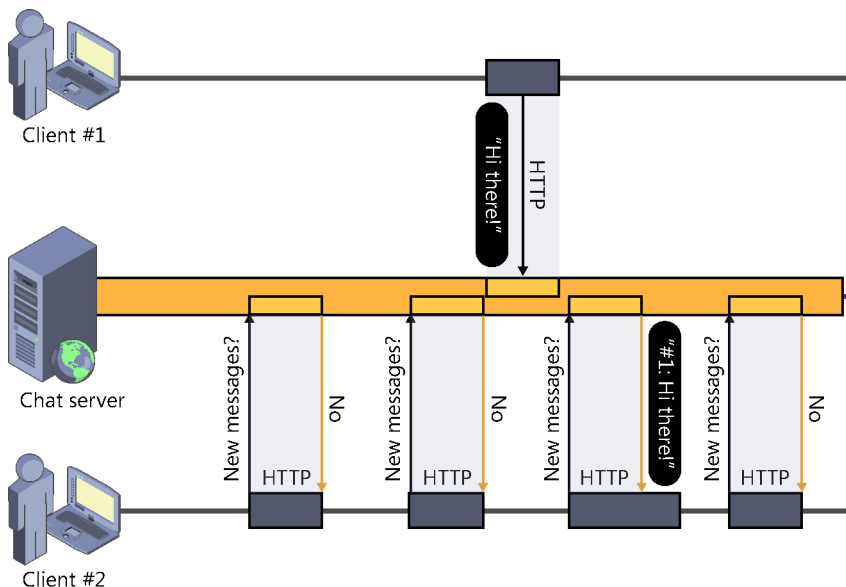


FIGURE 2-3 Polling in a chat room service.

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, because 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. The worst part is that this cost increases proportionally to our need for faster updates and the number of clients making use of the service at a given time. 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.

² Internet Relay Chat (IRC) protocol: <http://www.ietf.org/rfc/rfc1459.txt>

There are techniques to mitigate these problems insofar as possible. One of them is to use adaptive intervals so that the interval between queries regularly adapts to the current system load or to the probability of new updates. This solution 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 can be useful in scenarios that do not require great immediacy and in which the features of the system itself mean that we can be sure that the user will interact with the application frequently.

Of course, these variants can be combined with each other to achieve more efficient usage of resources, offering at the same time a reasonable user experience. For example, to obtain the updates, it would be possible to update the status of a client via piggy backing when the client interacts with the server, using polling with or without adaptive periodicity when there is no such interaction.

In conclusion, polling is a reasonable option despite its disadvantages when we want a solution that is easy to implement and that can be used universally and in scenarios in which a very high update frequency is not required. In fact, it is used a lot in current systems. A real-life example of its application is found in the web version of Twitter, where polling is used to update the timeline every 30 seconds.

Push: The server takes the initiative

We have 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 broadcasting 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.

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 that are push in concept, such as IRC, the protocol that rules the operation of classic chat room services, or SMTP, the protocol 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 have the ability 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.

To circumvent this issue and manage to obtain a similar effect, certain techniques emerged that were based on active elements embedded in webpages (Java applets, Flash, Silverlight apps, and so on). These components normally used sockets to open a persistent connection to the server—that is, a connection that 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 that 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 pure sockets are problematic when there are intermediary elements (firewalls, proxies, and so on) that can block these communications or close the connections after a period of inactivity. They can also pose security risks to servers.

Given the need for reliable solutions to cover these types of scenarios, both W3C and IETF—the main organizations promoting and defining protocols, languages, and standards for the Internet—began to work on two standards that 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.”

WebSockets

The WebSockets standard consists of 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>) has been 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, as shown in Figure 2-4.

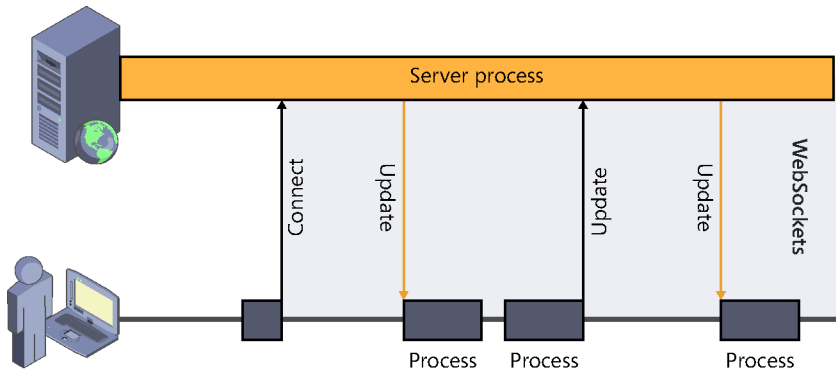


FIGURE 2-4 Operation of the WebSockets standard.

Although at the moment the specifications of both the API and the protocol are quite far advanced, we cannot yet consider this technology to be universally applicable.

We can find implementations of WebSockets in many current browsers, such as Internet Explorer 10, Internet Explorer 11, Chrome, and Firefox. Some feature only partial implementations (Opera mini, Android browser), and in others, WebSockets is simply not available³.

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 WebSockets are accepted.

For example, it has not been possible to use WebSockets easily on Microsoft technologies up until the very latest wave of developments (Internet Explorer 10, ASP.NET 4.5, WCF, IIS 8, and so on), in which it has begun to be supported natively.

From the perspective of a developer, WebSockets offers a JavaScript API that 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...");
};
```

³ Source: <http://caniuse.com/WebSockets>

As you can see, the connection is opened simply by instantiating a WebSockets object pointing to the URL of the service endpoint. The URL uses the `ws://` protocol to indicate that it is a WebSockets 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, WebSockets is the technology of the future for implementing push services in real time.

Server-Sent Events (API Event Source)

Server-Sent Events, also known as *API Event Source*, is the second standard on which the W3 consortium has been working. Currently, this standard is in candidate recommendation state. But this time, because it is a relatively straightforward JavaScript API and no changes are required on underlying protocols, its implementation and adoption are simpler than in the case of the WebSockets standard.

In contrast with the latter, Server-Sent Events 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, as illustrated in Figure 2-5.

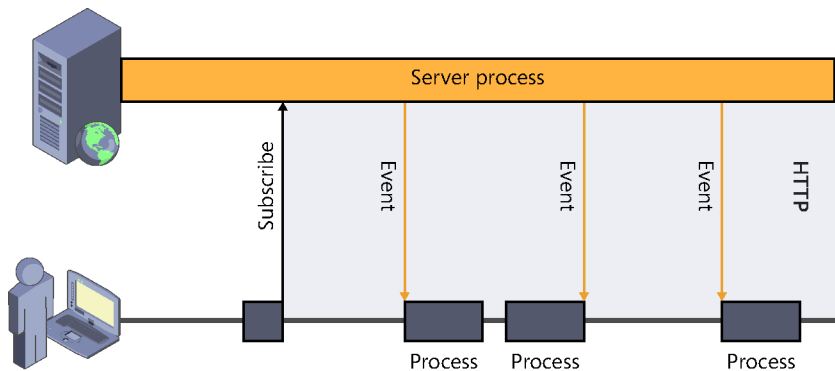


FIGURE 2-5 Operation of the Server-Sent Events standard.

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 earlier 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 call-back function specified to that effect.

Currently, almost all browsers support this standard except for Internet Explorer and some mobile-specific browsers, 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—for example, avoiding buffering responses or disconnections due to time-out.

It is also important to highlight the limitations imposed by the fact that the channel established for this protocol is one-directional from the server to the client: if the client needs to send data to the server, it must do so via a different connection, usually another HTTP request, which involves, for example, having greater resource consumption than if WebSockets were used in this same scenario.

Push today

As we have seen, standards and browsers are both 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. 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. At the moment, we can obtain that only 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, and so on, 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, because they are the most universal ones (they work in all types of client and server systems), and second, because they are used natively by SignalR, as we shall see later on. Thus we will move toward the objectives of this book.

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 is 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, as shown in Figure 2-6.

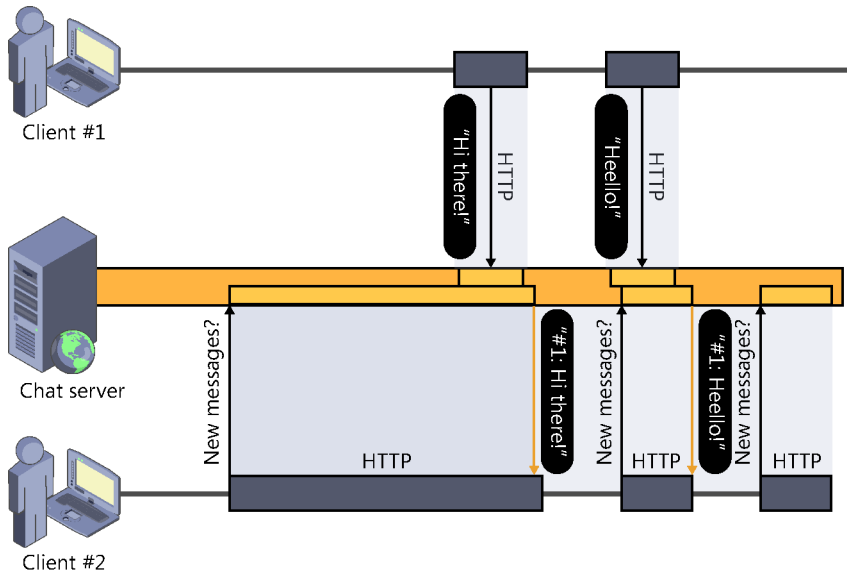


FIGURE 2-6 Long polling.

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

- The server sends data to the client through the connection.
- A time-out 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 upward, 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, because 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, not forgetting the additional connection that has to be used when the client wants to send data to the server. Also, the time it takes to establish connections means that there might be some delay between notifications. These delays could become more evident if the server

had to send a series of successive notifications to the client. Unless we implemented some kind of optimization, such as packaging several messages into one same HTTP response, each message would have to wait to be sent while the client received the previous message in the sequence, processed it, and reopened the channel to request a new update.

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. In a way, this is very similar to Server-Sent Events.

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 that are executed at the client as they are received.

Because 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, as shown in Figure 2-7.

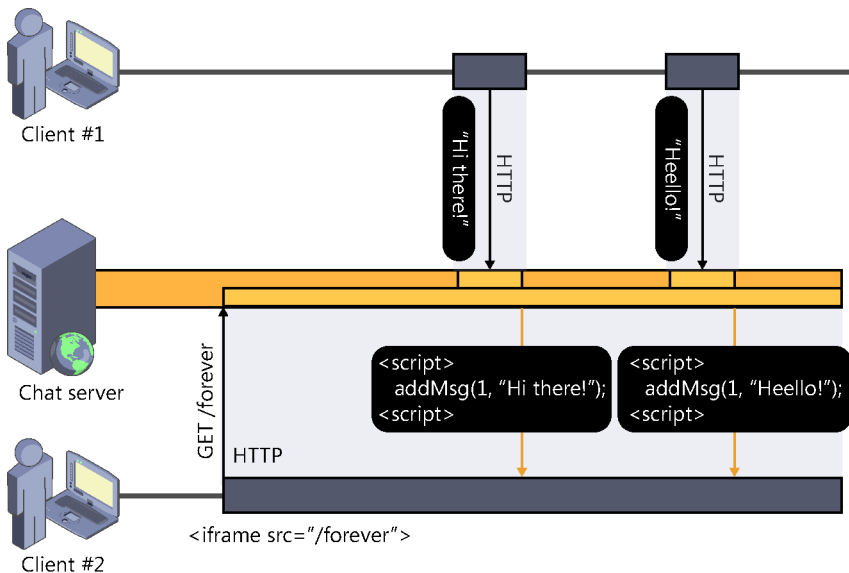


FIGURE 2-7 Forever frame.

This technique is not exempt from disadvantages either. In its implementation, it is necessary to take into account that there might be time-outs 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 might end up taking up too much RAM, so we have to “recycle” or eliminate contents periodically.

Finally, the fact that the connection is used only to send data from the server to the client makes it necessary to use an additional connection when we want to send it in the opposite direction—that is, from the client to the server.

The world needs more than just push

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 that 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. To create these always surprising systems, we need many more capabilities. 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 with 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** The server be capable not only of sending information to clients in real time but also of receiving it and processing it on the fly.
- **Monitoring submissions** Because we cannot guarantee that all clients connect under the same conditions, there might 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 submissions individually to ensure that all clients are updated.
- **Offering a flexible API, capable of being consumed easily by multiple clients** This is even truer 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 in developing these types of applications.

Enter SignalR....

Persistent connections

The lower-level API with which we can approach the persistent connection that SignalR offers is illustrated in Figure 4-1. This API provides us with a layer of abstraction that isolates us from the complexities inherent to keeping a permanent connection open between the client and the server and from the transports used to send information between both ends.

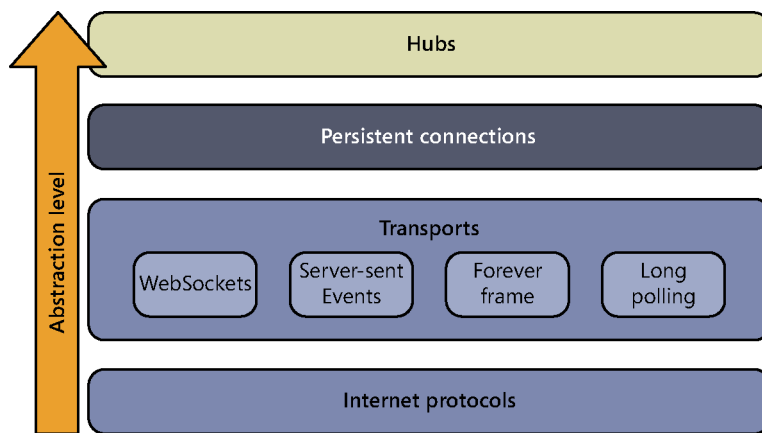


FIGURE 4-1 Abstraction levels used by SignalR.

In practice, this API gives us access to the communication channel that 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 that is invoked by the framework after its reception.

The server side is not very complex either. Take a look at Figure 4-2. 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.

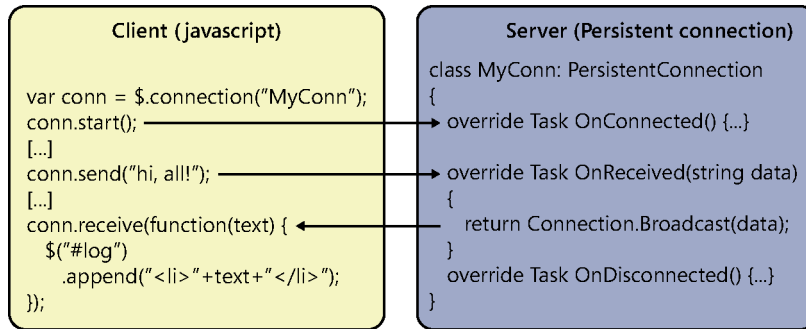


FIGURE 4-2 Conceptual implementation of persistent connections.

Now we will delve into all these aspects.

Implementation on the server side

For the clients to be able to connect to a SignalR service, first it is necessary to include this framework in our web project. For example, we can quickly do it by entering the following command in the NuGet package manager console:

```
PM> Install-package Microsoft.AspNet.SignalR
```

Each persistent connection is externally reachable via a URL, so, in a similar way to using other frameworks such as MVC or Web API, the next step could be to configure SignalR and associate each persistent connection to the path through which it will be available.

Mapping and configuring persistent connections

As usual, this is a process that must take place during application startup. In previous versions of SignalR, this registration was carried out in the `global.asax`, using extensions directly on the route collection of the application. However, since version 2.0, there is greater integration with OWIN¹, and this has changed the way it is implemented. In fact, from said version onwards, we need to register and configure SignalR in the middleware collection of the system in the startup process of the application.

We will come back to this, but for now it will suffice to know that the host process on which our application runs, based on OWIN, will search for a class called `Startup` in the root namespace of the application, and when it finds it, it will execute its `Configuration()` method. We will see that this convention can also be modified.

When the `Configuration()` method is executed, the execution environment will provide it with an argument in the form of an object implementing the `IApplicationBuilder` interface, which basically represents the application being initialized and contains a dictionary with configuration parameters

¹ OWIN (Open Web Interface for .NET): <http://owin.org/>

and methods that allow us to configure the different OWIN middleware that will process the requests, such as SignalR, Web API, authentication, tracing, and so on.

Configuration is normally performed by using extension methods on `IApplicationBuilder`. These methods are provided by the frameworks themselves or by middleware to facilitate their implementation. In our case, we will use the `MapSignalR()` method, defined by SignalR as an extension method for `IApplicationBuilder` in the Owin namespace, to link the used persistent connections to the paths through which we access them, as shown in the following OWIN startup class:

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        app.MapSignalR<EchoConnection>("/echo");

        // Configuration of other OWIN modules
    }
}
```

In any case, what we will achieve by calling `MapSignalR()` is “mapping” the paths of the type “/echo/something” to the class where we will implement the persistent connection, which in this case we have called `EchoConnection`. This path identifies the endpoint where the clients must be connected to consume the services. Obviously, at this point, there must be as many calls to the `MapSignalR<TConnection>()` method as the number of connections that are offered to the clients.

When this first step is completed, we are in position to implement the SignalR service, which will consist only 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 to process the request, which might depend on the transport selected each time. For example, if *WebSockets* is used as a transport, after the connection is established, the instance of `PersistentConnection` will remain active until the client disconnects, because it will be used both to send and receive data from the server. Contrariwise, if we use forever frame, an object will also be instantiated each time the client sends data, because 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 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, because it will not be possible to distribute the load among several servers.

Events of a persistent connection

The `PersistentConnection` class offers virtual methods that are invoked when certain events occur that are related to the service and the connections associated with the class, such as the arrival of a new connection, the disconnection of a client, or the reception of data. 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, 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 that can be quickly and synchronously executed, or to return a background task represented by a `Task` object that performs it asynchronously.

We can also observe that there are always at least two parameters sent to the methods: an `IRequest` object and a text string called `connectionId`.

The first one, `IRequest`, allows accessing specific information on the request received, such as cookies, information about the authenticated user, parameters, server variables, and so on, as shown in Figure 4-3. The `IRequest` interface is a specific SignalR abstraction that allows separating the implementation of services from ASP.NET. As we pointed out when we mentioned OWIN, this will allow them to be hosted in any .NET application. We will look at this in more depth later on.

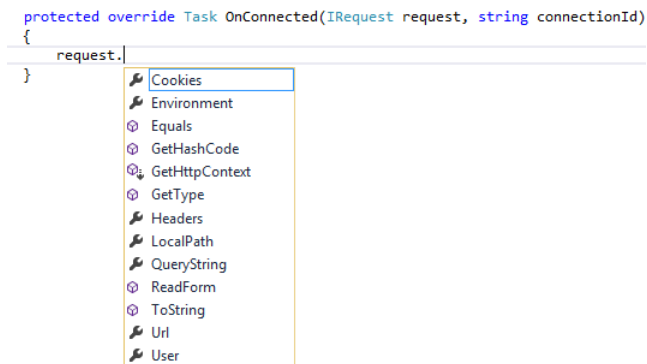


FIGURE 4-3 IntelliSense showing the members of `IRequest`.

The second parameter, `connectionId`, is a unique identifier associated with the connection that is generated by SignalR automatically during the initial negotiation process. The framework will use a GUID² by default, as shown in Figure 4-4.

² Globally Unique Identifier: http://en.wikipedia.org/wiki/Globally_unique_identifier

```
protected override Task OnConnected(IRequest request, string connectionId)
{
    return base.OnConnected(request, connectionId);
}
```

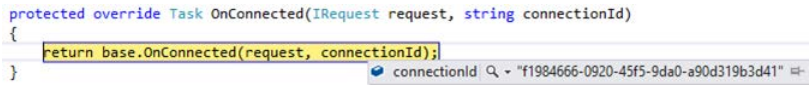


FIGURE 4-4 Value of a connectionID.

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 service:

```
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);
    }
}
```

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 time-out, a communication problem between the two ends, an application crash, a server reboot, or any other such incident. From the point of view of the SignalR 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 time-out and determining to which groups the connection should be reassigned.

The `OnReceived()` method of the persistent connection allows processing the data sent by the clients. In this method, the information submitted will be received as a text string:

```
protected override Task OnReceived(IRequest request,
                                   string connectionId,
                                   string data)
{
}
```

```

    // Do something interesting here
}

```

Of course, if an object serialized in any format came in this string, we should deserialize it manually before processing it. If it was JSON serialized, which will likely be the usual case, we could use 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;
        // ...
    }
    // ...
}

```

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 service, the notification text is sent to all connected clients (including the newcomer) through the SignalR 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 pass to `Broadcast()` is an object type (as shown in Figure 4-5), which means that we can send any type of object, which SignalR will serialize to JSON automatically.

```
protected override Task OnReceived(IRequest request, string connectionId, string data)
{
    var message = connectionId + ">> " + data;
    return Connection.Broadcast(
        (awaitable, extension) Task IConnection.Broadcast(object value, params string[] excludeConnectionIds)
        Broadcasts a value to all connections, excluding the connection ids specified.

        Usage:
        await Broadcast(...);
        value: The value to broadcast.
    );
}
```

FIGURE 4-5 `Broadcast()` receives an object type parameter.

The `Broadcast()` method also accepts an optional parameter where we can specify a collection of `connectionIds` to which the message will not be sent. The following example shows how to use this feature to send a notification to all users of the service except the one who has just connected:

```
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);
}
```



Note If the object supplied to `Send()` or `Broadcast()` is of the `ArraySegment<byte>` type, it will not be serialized. This could be useful if we already have the JSON representation of the object to be sent, because it would prevent double serialization of the object.

Although the most frequent procedure for making deliveries will be using the methods just described, the `Send()` method has some additional overloads. One of them receives as arguments a list of strings representing the connection identifiers to which we want to send the message. Another, actually used internally by `Broadcast()` to make the delivery, simply receives a `ConnectionMessage` type object that defines the recipients, the message, and, optionally, the connection identifiers that we want to exclude:

```
var connMessage = new ConnectionMessage(
    Connection.DefaultSignal, textMessage);
return Connection.Send(connMessage);
```

The preceding code would be equivalent to making a broadcast with the value of `textMessage`. In fact, the `Broadcast()` method uses `Send()` internally in a way that is quite similar to the one shown. `Connection.DefaultSignal` is a unique code made up by the full name of the persistent connection class preceded by a constant prefix, which in this case indicates that the message must be sent to all the users “subscribed” to this signal, which are all those connected to the persistent connection.

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 very large. If they were executed synchronously, the threads in charge of performing these tasks would be blocked until said tasks ended. In high load environments, this is truly a waste of resources; we want these threads to be released as soon as possible so that they can keep managing requests and providing their services.

For this reason, those commands are executed asynchronously, returning a `Task` object representing the task that will take care of them 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);
```

Following this same example, we must use asynchrony 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, or heavy queries to databases.

The following example shows how to use the `async/await` construct of C# 5 to invoke asynchronous methods in a very clean way:

```
protected override async Task OnConnected(IRequest request,
                                         string connectionId)
{
    // Store the new connection in the database
    await _services.SaveNewConnectionAsync(connectionId);

    // And then, send the notifications
    await this.Connection.Broadcast("A new user is online!");
    await this.Connection.Send(connectionId,
                              "Welcome, " + connectionId);
}
```

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 that 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, implementing this very simple 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 args = data.Split(new[] { " " },
                          StringSplitOptions.RemoveEmptyEntries);

    if (args.Length == 2 && args[0].ToLower() == "join")
    {
        return this.Groups.Add(connectionId, args[1]);
    }
    if (args.Length == 2 && args[0].ToLower() == "leave")
    {
        return this.Groups.Remove(connectionId, args[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 preceding code would interpret a message such as `"signalr:hello!"` by sending the text `"hello!"` to all clients connected to the `"signalr"` group.

Also, there is an overload of the `Send()` method that allows specifying a list of group names as the recipient:

```
var groupNames = new[] { "firstgroup", "secondgroup" };
this.Groups.Send(groupNames, message);
```

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.

The OWIN startup class

We have previously seen where to enter the mapping and configuration code of our persistent connections and, in general, of any middleware based on OWIN. We will now go back to that briefly, to go over some details that were left pending.

When an OWIN-based system starts up, the host process on which it is executed will try to execute configuration code that must be implemented in a member predefined by convention. By default, it will try to execute the `Configuration()` method of the `Startup` class, which must be located in the root namespace of the application. However, to adapt it to our preferences, it is possible to modify this convention in one of the following ways:

- Specifying the class and the method that we want to employ by using the assembly attribute `OwinStartup`:

```
[assembly:OwinStartup(typeof(MyApp.MyStartupClass),
                      methodName: "MyConfigMethod")]
```

- Including the entry `"owin:AppStartup"` in the `<AppSettings>` section of the `.config` file of the application and setting as a value the fully qualified name of the class and the method to be used:

```
<configuration>
  <appSettings>
    ...
    <add key="owin:AppStartup"
        value="MyApp.MyStartupClass.MyConfigMethod"/>
  </appSettings>
  ...
</configuration>
```

In either case, the configuration method can be static or an instance method (although in the latter case the class must have a public constructor without parameters), and it must necessarily be defined with an `IApplicationBuilder` parameter:

```
namespace MyApp
{
    public class MyStartupClass
    {
        public void MyConfigMethod(IApplicationBuilder app)
        {
            // Configure OWIN app here
        }
    }
}
```

Because all the OWIN middleware of the application will be initialized in this method, it might be advisable to take the specific configuration of SignalR to an independent class. Obviously, if we are using the conventions of ASP.NET MVC or Web API on file location, it will probably be a much better idea to enter it into the /App_Start directory and, if the configuration code is too long, we should even separate it into an independent class and file. The following example shows a possible way to organize these files:

```
//  
// File: /App_Start/Startup.cs  
//  
public class Startup  
{  
    public void Configuration(IAppBuilder app)  
    {  
        SignalRConfig.Setup(app);  
    }  
}  
  
//  
// File: /App_Start/SignalRConfig.cs  
//  
public class SignalRConfig  
{  
    public static void Setup(IAppBuilder app)  
    {  
        app.MapSignalR<EchoConnection>("/echo");  
    }  
}
```

In any case, this does not constitute a norm to be used mandatorily. The point is to stick to the conventions defined and well-known by the development team so that things are located where it is expected they should be.

Implementation on the client side

SignalR offers many client libraries with the purpose that practically any kind of application can use the 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.

Initiating the connection by 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: Web Forms, ASP.NET MVC, Web Pages, PHP, or even from pure HTML pages.

In fact, to access services in real time from an HTML page, it is enough to reference jQuery (version 1.6.4 or above)—because the client is implemented as a plug-in of this renowned framework—and then the `jquery.signalR` library (version 2.0.0 or above). 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-1.6.4.min.js"></script>
<script src="Scripts/jquery.signalR-2.0.0.min.js"></script>
```

After they are referenced, we can begin to work with SignalR from the client side. The first thing we have to do is 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 assigned during application startup. Here you can see an example:

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        app.MapSignalR<EchoConnection>("/realtime/echo");
    }
}
```

Given the previous configuration, which maps the path `/realtime/echo` to the persistent connection implemented in the `EchoConnection` class, the following code shows how to create and open a connection to it using JavaScript:

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

"SignalR: Connection must be started before data can be sent. Call .start() before .send()"

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

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

It is important to know that the specified callback function will be executed whenever the connection is initiated. That is, if the `start()` method is invoked from another point in the client code, the previously registered callback function will also be executed when the connection process completes successfully.

It is also possible to use the well-known promise³ pattern and the implementation of jQuery based on Deferred⁴ objects to take control when the connection has been successful, as well as in the event that an error has occurred. This is the recommended option:

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

After the connection is established, we can begin to send and receive information by using the mechanisms that are described later in this chapter.

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

³ The promise pattern: <http://wiki.commonjs.org/wiki/Promises>

⁴ The Deferred object in jQuery: <http://api.jquery.com/category/deferred-object/>

Support for older browsers

A problem that might arise when we open the connection using the `start()` method is that the user's browser might be too old and not have the JSON parser built in—for example, as it happens with Internet Explorer 7. In this case, execution will stop, indicating the problem and even its solution:

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

"SignalR: Connection must be started before data can be sent. Call .start() before .send()"

The `json2.js` file that we are told must be referenced can be easily obtained from NuGet with the following command in the console:

```
PM> Install-package json2
```

When this is done, we would have to include only the reference to the script before doing so with SignalR:

```
<script src="Scripts/jquery-1.6.4.min.js"></script>
<script src="Scripts/json2.min.js"></script>
<script src="Scripts/jquery.signalR-2.0.0.min.js"></script>
```

Support for cross-domain connections

SignalR includes "out-of-the-box" support for connections with a different server from the one that has served the script currently being executed, something that is normally not allowed for security reasons. This type of request, called *cross-domain*, requires the use of some kind of special technique to avoid this restriction that is usual in browsers. Examples of those techniques are JSONP⁵ or, only in some browsers, the use of the CORS⁶ specification.

JSONP is not particularly recommended for security reasons, but if our service must provide support to clients using older browsers such as Internet Explorer 7, it might be the only option available to obtain connections from external domains. This feature is disabled by default at server level, but it can be activated by supplying a `ConnectionConfiguration` object during initial mapping, setting its `EnableJSONP` property to `true`:

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        var config = new ConnectionConfiguration()
        {
```

⁵ JSONP (JSON with Padding): <http://en.wikipedia.org/wiki/JSONP>

⁶ CORS (Cross-Origin Resource Sharing): http://en.wikipedia.org/wiki/Cross-origin_resource_sharing

```

        EnableJSONP = true
    };
    app.MapSignalR<EchoConnection>("/realtime/echo", config);
}
}

```

This way, the server will be ready to accept connections using long polling transport with JSONP.

Because CORS is quite a cross-cutting technique and independent of frameworks and applications, it is implemented as OWIN middleware; therefore, to allow this type of connection, it will be necessary to first download the module using NuGet:

```
PM> Install-Package microsoft.owin.cors
```

After this, we can specify that we want to use CORS in our initialization method, entering the module into the OWIN pipeline with the extension method `UseCors()` so that it is executed before the SignalR middleware:

```

public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        app.Map("/realtime/echo",
            map => {
                map.UseCors(CorsOptions.AllowAll);
                map.RunSignalR<EchoConnection>();
            }
        );
    }
}

```

Note that in this case we have used a different construct to map and configure the service available in the path `/realtime/echo`. First, we have used the `Map()` extension to specify the path just once, followed by a lambda that receives as a parameter the mapping that we are defining (in turn, an `IApplicationBuilder` object). On this parameter, we have used the extensions provided by the different modules to enter them into the pipeline associated to the URL provided:

- `UseCors()` enables the use of CORS at the server, according to the options sent as an argument in the form of a `CorsOptions` object. `CorsOptions.AllowAll` is an object preconfigured in a very permissive mode; it allows all origins, verbs, and headers. However, it is possible to supply it a customized `CorsOptions` object to fine-tune its configuration and usage policies.
- `RunSignalR()` is the equivalent of the `MapSignalR()` method that we had been using up until now, but as opposed to it, we do not need to supply it the path because it has already been defined.

At the client side, SignalR will automatically detect that we are making a cross-domain connection if when the connection is made it notices that an endpoint has been specified that is hosted in

a domain different than the one of the current page. In this case, it will try to use CORS to make the connection. Only if it is not possible will it automatically fall back to JSONP, using long polling as the transport.

In scenarios where we need to force this last option, we can specifically indicate that we want to use JSONP when initiating the connection from the client. At this moment, it is possible to send an object with settings that allow fine-tuning said process:

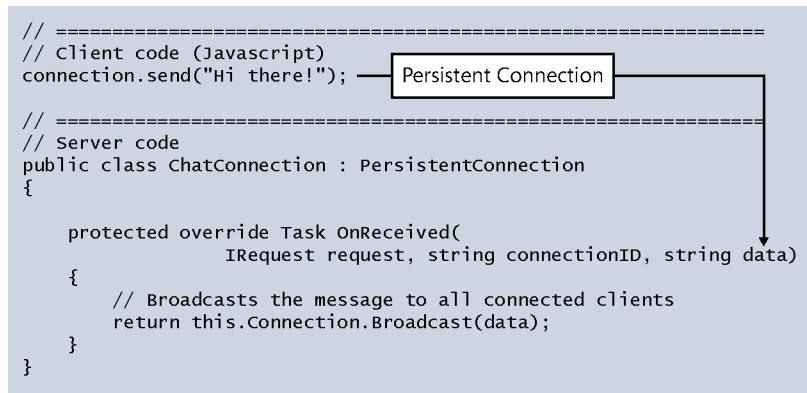
```
var connection = $.connection("http://localhost:3701/realtime/echo");
connection.start({ jsonp: true })
    .done(function() {
        alert(connection.transport.name);
    });
```

The `connection.transport` property contains the transport used by the current connection.

Sending messages

As you can probably guess, 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 object to be sent as a parameter. It will be received in the `data` parameter of the `OnReceived()` method of the server in the form of a text string, as we saw previously:



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

// =====
// Server code
public class ChatConnection : PersistentConnection
{
    protected override Task OnReceived(
        IRequest request, string connectionID, string data)
    {
        // Broadcasts the message to all connected clients
        return this.Connection.Broadcast(data);
    }
}
```

Of course, we can send any object type. SignalR will serialize it automatically before sending it:

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

Because what would arrive at the server would be a text string with the JSON representation of the object, 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);
    }
    // ...
}
```

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 whether there has been an error in its execution.

Nevertheless, it is possible to detect errors on the connection by using the `error()` method of the connection to set the callback function to be executed when there is any problem on it:

```
var connection = $.connection("/chat");
connection.error(function (err) {
    alert("Oops! It seems there is a problem. \n" +
        "Error: " + err.message);
});
connection.start();
```

The callback function receives a JavaScript object from which we can obtain information describing the problem that has occurred, as shown in Figure 4-6.

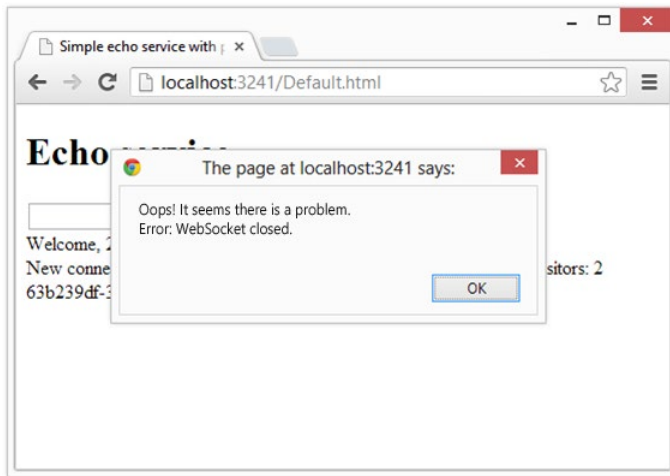


FIGURE 4-6 Connection interrupted captured by the callback.

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. SignalR will be responsible for serializing and deserializing the data on both ends.

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 preceding 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);
    });

```

Sending additional information to the server

We have seen that the events available at the server receive an `IRequest` type parameter through which it is possible to access the environment of the request that is behind the persistent connection. Thus, using this object, it would be possible to retrieve, among other things, the identity of the user authenticated into the system, information sent in cookies, or even the parameters of the query string or server headers.

For example, if you are using cookie-based authentication, when a user is authenticated in a web-site, the browser will automatically include the authentication cookie in all requests to SignalR, which offers the possibility of implementing code such as the following at the server:

```

protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    string message = request.User.Identity.IsAuthenticated
        ? "Welcome, " + request.User.Identity.Name
        : "You must be logged in!";

    return Connection.Send(connectionId, message);
}

```

In the same vein, when it is necessary to send additional information from the client to the server, we can make use of cookies. They are easy to manage, and they allow entering arbitrary information into requests that are going to be made *a posteriori*, as illustrated in the following example:

```

// Client side:
var username = prompt("Your username");
document.cookie = "username=" + username;
var connection = $.connection("/realtime/chat");
...
connection.start();

// Server side:
protected override Task OnConnected(IRequest request,
                                    string connectionId)
{
    Cookie cookie;
    var username =
        request.Cookies.TryGetValue("Username", out cookie)
        ? cookie.Value
        : connectionId;
}

```

```

    var message = "Welcome, " + username + "!";
    return Connection.Send(connectionId, message);
}

```

Another possibility would be to use the query string to send information. For this, the SignalR client allows us to specify an additional parameter at the point at which the connection is defined. In this parameter, we can add key-value mappings, either in the form of a string or an object, which will be annexed to all the requests made to the SignalR server:

```

// Client side:
var name = prompt("Your username");
var conn = $.connection(
    "/realtime/chat",
    "username="+name // or { username: name }
);
...
conn.start();

// Server side:
protected override Task OnConnected(IRequest request,
                                     string connectionId)
{
    var userName = request.QueryString["username"] ?? connectionId;
    var message = "Welcome, " + userName + "!";
    return Connection.Send(connectionId, message);
}

```

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 the following:

- `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 time-out or any other cause.

In the SignalR repository in GitHub⁷, you can find the documentation about methods, events, and properties offered by the JavaScript client connections.

⁷ Javascript client documentation: <https://github.com/SignalR/SignalR/wiki/SignalR-JS-Client>

Transport negotiation

We have seen that, after 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 WebSockets, 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, to determine which is the most efficient solution available at both ends, a fallback procedure will begin:

- In the browsers that support it (basically all with the exception of Internet Explorer), contact will be attempted using Server-Sent Events, which at least provides a standard mechanism to obtain *push*.
- In Internet Explorer, the possibility of using forever frame will be examined.

If neither of the previous steps has succeeded, to maintain the persistent connection, long polling will be tried.

It is easy to trace this process, because the SignalR web client allows us to activate the tracing of events with the JavaScript console available in major browsers. (See Figure 4-7.) This option is enabled at the moment the connection is created, setting the third parameter of the `$.connection()` method to `true`:

```
var connection = $.connection("/realtime/chat", null, true);
```

Or, in an equivalent and much more legible way, directly on the `logging` property of the connection:

```
var connection = $.connection("/myconn");  
connection.logging = true;
```

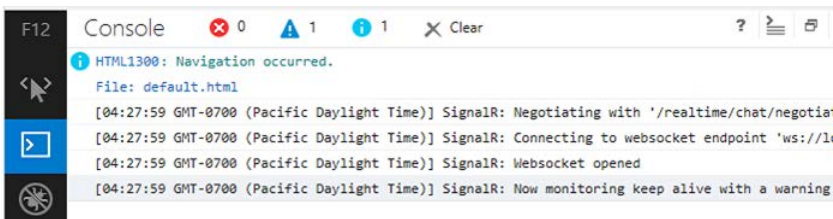


FIGURE 4-7 Trace of the SignalR client in Internet Explorer 11.

We can also trace requests using Fiddler⁸ or the development tools available in our browsers to trace requests, and thus we can view the main terms of the “agreement” reached by the client and the server.

Figure 4-8 shows the tracing of connections using the development tools of Chrome, where you can see the process of content negotiation between this browser and IIS 8. Both support WebSockets.

Elements Resources Network Sources Timeline Profiles Audits Console				
Name	Method	Status	Type	Initiator
Path		Text		
Default.html	GET	304 Not Modified	text/html	Other
jQuery-1.6.4.min.js /Scripts	GET	304 Not Modified	application/javascr...	Default.html:5 Parser
json2.min.js /Scripts	GET	304 Not Modified	application/javascr...	Default.html:5 Parser
jQuery.signalR-2.0.0.min.js /Scripts	GET	304 Not Modified	application/javascr...	Default.html:5 Parser
negotiate?clientProtocol=1.3&_=1374665787052 /realtime/chat	GET	200 OK	application/json	jQuery-1.6.4.min.js:4 Script
connect?transport=webSockets&connectionToken=IOy5niOdIR... /realtime/chat	GET	101 Switching Protocols	Pending	Other

FIGURE 4-8 Process of transport negotiation with Chrome.

However, Fiddler⁹ is used in Figure 4-9 to show the negotiation procedure of Internet Explorer 7 and how the fallback mechanism works to select the best transport supported by it—forever frame:

Fiddler Web Debugger				
File Edit Rules Tools View Help GET /book				
Win8 Config Win8 Replay X Resume Stream Decode Keep: All sessions Any Process Fi				
#	Result	Protocol	Host	URL
4	200	HTTP	localhost:2501	/default.html
5	200	HTTP	localhost:2501	/Scripts/jquery-1.6.4.min.js
6	200	HTTP	localhost:2501	/Scripts/jquery.signalR-2.0.0.min.js
7	200	HTTP	localhost:2501	/Scripts/json2.min.js
8	200	HTTP	localhost:2501	/realtime/chat/negotiate?clientProtocol=1.3&_=137466729...
9	-	HTTP	localhost:2501	/realtime/chat/connect?transport=foreverFrame&connectio...

FIGURE 4-9 Transport negotiation procedure with Internet Explorer 7.

Finally, Figure 4-10 shows the trace by the Firefox console (Firebug) of the negotiation of a cross-domain connection, which is resolved employing the WebSocket transport using CORS.

```
[14:32:22 GMT+0200] SignalR: Auto detected cross domain url.
[14:32:22 GMT+0200] SignalR: Negotiating with 'http://localhost:2980/tracker/negotiate?clientProtocol=1.
[14:32:22 GMT+0200] SignalR: Connecting to websocket endpoint 'ws://localhost:2980/tracker/connect?trans
connectionToken=E6xAUVq1Mxg4sppMOjAokJAg420h84tIMlohH*2FtahWEehmnExjDcDaEo8NjUbHKquvvvJPTavAMU4h8f4ImSAf
tid=2'
[14:32:22 GMT+0200] SignalR: Websocket opened
[14:32:22 GMT+0200] SignalR: Now monitoring keep alive with a warning timeout of 13333.33333333332 and
```

FIGURE 4-10 Cross-domain connection negotiation with Firefox.

⁸ Fiddler: <http://www.fiddler2.com>

⁹ Notes on the use of Fiddler with SignalR: <https://github.com/SignalR/SignalR/wiki/Using-fiddler-with-signalr>

Adjusting SignalR configuration parameters

SignalR allows us to adjust certain parameters that affect the way in which connections are made, as well as other aspects related to their management. This is done through a configuration object available in the global object `GlobalHost`. As we shall see throughout this book, this object allows static access to some interesting functions of SignalR, but for now, we will focus on its `Configuration` property, which is where we will be able to adjust the value of the following parameters:

- `TransportConnectTimeout`, which is a `TimeSpan` that specifies the length of time that a client is to allow for the connection to be made using a transport, before falling back to another with inferior features or before the connection fails. The default value is five seconds.
- `ConnectionTimeout` is a `TimeSpan` specifying the length of time for which a connection must remain open and inactive before a time-out occurs. The default value is 110 seconds. It is effective only for transports that don't support keep alive, or if keep alive is disabled.
- `DisconnectTimeout` specifies the length of time from when a connection is closed until the disconnect event is fired. The default value is 30 seconds.
- `KeepAlive` is a nullable `TimeSpan` that allows us to specify the length of time between messages sent to the server indicating that the client remains active. The default value is 10 seconds, but we can disable it by setting a null value; see Figure 4-11. In any case, we cannot enter a value smaller than two seconds nor greater than one third of the value of `DisconnectTimeout`.

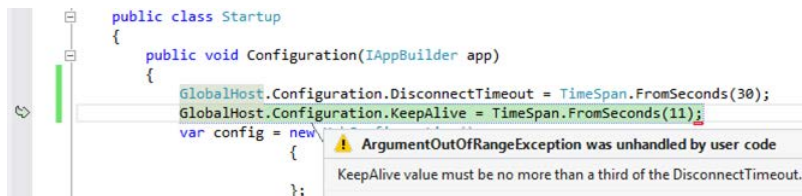


FIGURE 4-11 Exception during application startup.

- `DefaultMessageBufferSize` is an integer indicating the size of the message buffer for a specific signal (connection, group, users, and so on). The default value is 1,000.
- `LongPollDelay` is a `TimeSpan` that allows us to specify the length of time that a client must wait before opening a new long polling connection after having sent data to the server. The default value is 0.

Some of these parameters, such as `TransportConnectTimeout` or `KeepAliveTimeout`, are sent to the client in the negotiation phase of the connection so that it can apply them during said connection.

Naturally, for them to take effect, the changes on the configuration properties should be made during application startup—for example, like this:

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        GlobalHost.Configuration
            .DisconnectTimeout = TimeSpan.FromSeconds(30);

        app.MapSignalR<EchoConnection>("/echo");
    }
}
```



Note These settings are also valid when we use hubs, because they are global configurations of the server.

Complete example: Tracking visitors

We will now look at 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.

Figure 4-12 shows the system being executed on a busy page.

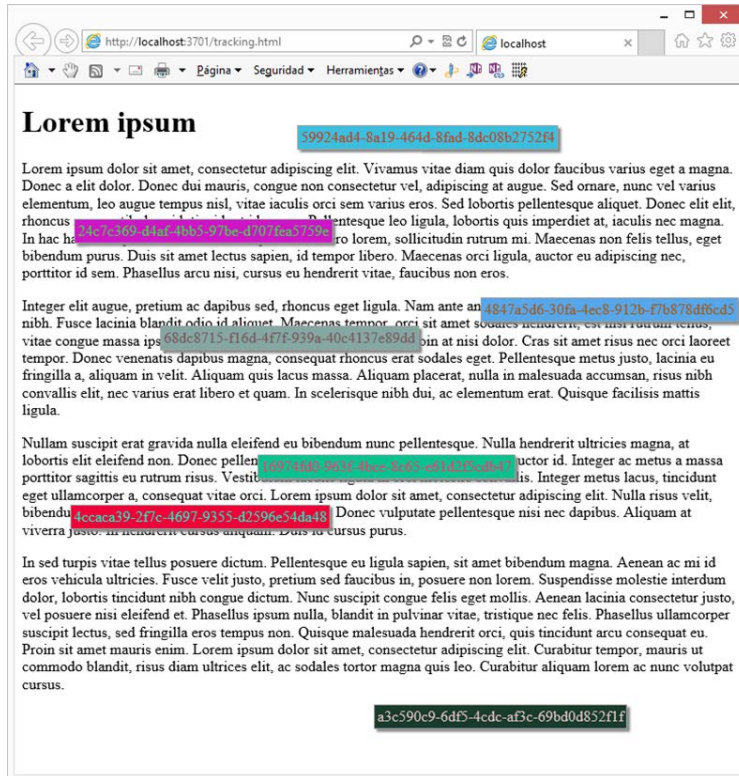


FIGURE 4-12 System for tracking users in real time in operation.

Project creation and setup

For the purpose of creating the application that we will develop over the following pages, it is necessary to first create a project of the “ASP.NET Web Application” type from Visual Studio 2013 and then select the “Empty” template to create a completely empty project¹⁰. The version of the .NET framework used must be at least 4.5.

After we have created it, we must install the following package using NuGet:

```
PM> install-package Microsoft.AspNet.SignalR
```

¹⁰ In Visual Studio 2012, we can achieve the same goal by creating a project from the template “ASP.NET Empty Web Application.”

Implementation on the client side

HTML markup (tracking.html)

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <script src="Scripts/jquery-1.6.4.min.js"></script>
  <script src="Scripts/jquery.signalR-2.0.0.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>
```

Scripts (Scripts/Tracking.js)

```
$(function() {

  /* SignalR client */
  var connection = $.connection("/tracker");
  connection.start(function () {
    startTracking();
  });

  connection.receive(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(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),
    };
}
});

```

Implementation on the server side

Persistent connection (TrackerConnection.cs)

```

using System.Threading.Tasks;
using Microsoft.AspNet.SignalR;

public class TrackerConnection : PersistentConnection
{
    protected override Task OnReceived(IRequest request,
        string connectionId,
        string data)
    {
        return Connection.Broadcast(data);
    }
}

```

Startup code (Startup.cs)

```
using Owin;

public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.MapSignalR<TrackerConnection>("/tracker");
    }
}
```


Index

A

- AAA acronym, 207
- access control
 - in hubs, 182–184
 - in persistent connections, 181–182
- actions, receiving and processing, 15
- Active Directory Domain Services (AD DS), 181, 185
- AD DS (Active Directory Domain Services), 181, 185
- adaptive intervals, 8
- Add Counters dialog box, 176–177
- AJAX (Asynchronous JavaScript And XML), 6, 190
- AJAX push, 12
- AjaxMin package, 196
- Android platform, 150
- AngularJS framework, 230–232
- API Event Source (Server-Sent Events)
 - additional techniques supporting, 15
 - described, 11–12
 - forever frame and, 14
 - push and, 12–15
 - transport negotiation, 48
- ApiController class, 224
- appcmd.exe tool, 174
- AppFunc (application delegate), 23
- application delegate (AppFunc), 23
- ArraySegment type, 34
- ASP.NET. *See also* MVC framework
 - Authorize attribute, 182–184
 - performance counters, 180
 - server configuration, 174–175
- ASP.NET stack, 17
- async/await construct (C#)
 - asynchronous event processing, 35
 - communicating with server using hubs, 136
 - creating and opening persistent connections, 133

- receiving messages, 63
 - sending messages to clients, 67
- asynchronous communication
 - AJAX operations, 6
 - event processing, 34–35
 - hub methods, 63
 - recommendations, 173
- Asynchronous JavaScript And XML (AJAX), 6, 190
- authentication
 - client, 184–190
 - cookie-based, 46, 185
 - OAuth 2.0, 185
 - in SignalR, 181–190
- authentication tokens, 186
- authorization in SignalR, 181–190
- Authorize attribute, 182–184
- Autofac IoC container, 201
- automatic proxies, 79, 83
- Available KBytes performance counter, 180
- Available MBytes performance counter, 180
- Azure Management Tool, 163

B

- backplanes
 - custom, 170–172
 - described, 156–157
 - operation with, 156
 - Redis storage system, 157, 167–170
 - scaling on, 159–170
 - SQL Server, 157, 165–167
 - Windows Azure, 157, 159–164
- Browser Link feature, 18
- browsers
 - cookie-based authentication, 46
 - JSON parser and, 41

buffering

- multiplatform SignalR servers, 120
- transport negotiation, 48–49
- WebSockets support, 10
- buffering, 173

C

- C++ client, 149
- C# language
 - application example, 142–147
 - async/await construct, 35, 63, 67, 133, 136
- Calculator class, 208–209
- camel casing style, 86
- CDN (Content Delivery Networks), 154
- client authentication, 184–190
- client/server architecture
 - HTTP operations, 5–7
 - messaging bus, 19
 - negotiation, 18
 - polling technique, 7–8
 - push concept, 8–15
- client-side processing
 - console application example, 140–141
 - hubs
 - communication with server, 135–138
 - creating and opening connections to, 134–135
 - described, 78
 - establishing the connection, 83–85
 - generating the proxy, 79–81
 - implementing without a proxy, 93–96
 - JavaScript clients, 79
 - logging, 91–92
 - manual generation of JavaScript proxies, 81–83
 - progress bar example, 114–115
 - receiving messages sent from server, 90–91
 - security and, 9
 - sending additional information, 89–90
 - sending message to server, 86–89
 - shared drawing board example, 98
 - state maintenance, 92–93
 - multiplatform applications
 - accessing services from non-web clients, 130–149
 - consumption of services from other platforms, 149–150
 - described, 129–130
 - persistent connections
 - creating and opening, 130–133
 - cross-domain connection support, 41–43
 - described, 27, 38
 - events available at, 47
 - initiating using JavaScript client, 38–40
 - older browser support, 41
 - receiving messages, 45–46
 - sending additional information to server, 46–47
 - sending and receiving data, 133–134
 - sending messages, 43–45
 - tracing requests, 111
 - tracking visitors example, 53
 - recommendations, 174
 - Windows Phone problems, 147–149
- Clients property (Hub class)
 - All value, 64–65, 112, 212
 - AllExcept value, 65, 112
 - Caller value, 66, 70, 138
 - Client value, 66, 68, 112, 214
 - Clients value, 66
 - Group value, 66, 72, 112
 - Groups value, 67, 73, 112
 - Others value, 66
 - OthersInGroups value, 67, 72
 - User value, 67
- cloud services, 154
- clustering servers, 173
- Comet, 12
- compression, OWIN middleware modules, 24
- ConcurrentDictionary class, 74
- .config file, 37, 175
- Connection class
 - AddClientCertificate() method, 131
 - Closed event, 134
 - ConnectionId property, 40, 131
 - ConnectionSlow event, 47, 134
 - CookieContainer property, 131–132
 - Credentials property, 132
 - Error event, 44, 47, 134
 - Headers property, 132
 - Logging property, 48
 - Proxy property, 132
 - QueryString property, 89
 - Received event, 45, 47, 134
 - Reconnected event, 47
 - Reconnecting event, 134

- Send() method, 43–44, 133, 142–149
- Start() method, 39–41, 48, 83–84, 132–133
- State property, 132
- StateChanged event, 47, 134
- Stop() method, 40
- Trace() method, 138
- TraceLevel property, 138–139
- TraceWriter property, 139
- connection groups, 35–37
- Connection Messages Received/Sec performance counter, 178
- Connection Messages Received Total performance counter, 178
- Connection Messages Sent/Sec performance counter, 178
- Connection Messages Sent Total performance counter, 178
- ConnectionConfiguration class, 41
- connectionID identifier, 30–33, 35
- ConnectionMessage structure, 34
- Connections Connected performance counter, 178
- Connections Current performance counter, 178
- Connections Disconnected performance counter, 178
- Connections Reconnected performance counter, 178
- console applications
 - application example, 140–141
 - multiplatform SignalR clients, 130
 - multiplatform SignalR servers, 119–122
- Content Delivery Networks (CDN), 154
- Context property (Hub class)
 - ConnectionId value, 71, 212
 - described, 72, 89
 - Headers value, 71
 - QueryString value, 71, 89
 - RequestCookies value, 71
 - unit testing example, 212
 - User value, 71
- Controller class, 226
- cookie-based authentication, 46, 185
- CORS (Cross Origin Resource Sharing)
 - multiplatform SignalR servers, 120
 - OWIN middleware modules, 24
 - persistent connections and, 41–42
 - responding to requests, 85
- Creative Commons license, 21
- cross-domain connections, 41–43
- Cross Origin Resource Sharing (CORS)
 - multiplatform SignalR servers, 120

- OWIN middleware modules, 24
- persistent connections and, 41–42
- responding to requests, 85

D

- death by success, 153
- DefaultDependencyResolver class, 192, 202
- Deferred object (jQuery), 40
- Dependency Injection (DI)
 - described, 191, 196–198
 - Inversion of Control containers, 200–205
 - manual, 198–200
 - releasing dependencies, 200
- Dependency Resolver, 69, 191–196
- deployment scenarios
 - death by success and, 153
 - described, 151–153
 - improving performance in SignalR services, 173–180
 - OWIN standard example, 23
- DI (Dependency Injection)
 - described, 191, 196–198
 - Inversion of Control containers, 200–205
 - manual, 198–200
 - releasing dependencies, 200
- Dictionary class, 74
- dynamic proxies, 79–81

E

- EchoConnection class, 29, 39, 217
- Edwards, Damian, 17
- Errors: All/Sec performance counter, 178
- Errors: All Total performance counter, 178
- Errors: Hub Invocation/Sec performance counter, 178
- Errors: Hub Invocation Total performance counter, 178
- Errors: Hub Resolution/Sec performance counter, 178
- Errors: Hub Resolution Total performance counter, 178
- Errors: Transport/Sec performance counter, 178
- Errors: Transport Total performance counter, 178
- events
 - asynchronous processing, 34–35
 - described, 30–32
 - logging, 91–92, 138–139

EventSource object

EventSource object, 12
extensible framework, 191–196

F

Facebook OAuth authentication, 185
FakeltEasy framework, 209
Fakes framework, 209
Fiddler tool, 49, 84
FileSystemWatcher class, 172
forever frame
 described, 12, 14–15
 persistent connections and, 29
 transport negotiation, 48–49
Fowler, David, 17

G

garbage collection, 180
Get-Package command, 25
GlobalHost class
 Configuration property, 50–51
 ConnectionManager property, 111, 224–225
 DependencyResolver property, 192–193
 HubPipeline property, 219
globally unique identifier (GUID), 30
GUID (globally unique identifier), 30

H

horizontal scalability, 154–155
hosting
 multiplatform SignalR servers
 console applications, 119–122
 described, 118
 platforms other than Windows, 126–129
 Windows service, 122–126
 OWIN standard, 22–24, 118–119
HTML <IFRAME> tag, 14
HTML5, 9
HTTP (HyperText Transfer Protocol)
 503 errors, 175
 described, 5
 HTTP operations, 5–7
 inefficiencies of, 6–7
 polling, 7–8, 12–14
 Server-Side Events, 11–12

 synchronous communication, 5–6
HTTP push, 12
HttpContext class, 184
HttpListener class, 118–119, 123
Hub class
 Clients property, 64–68, 70, 72–73, 112, 138, 212, 214
 Context property, 71–72, 89, 212
 OnConnected() method, 211, 213
HubConnection class
 Closed event, 135
 CreateHubProxy() method, 135
 described, 134–135
 Start() method, 135
 StateChanged event, 135
 Stop() method, 135
HubException exception, 88
HubPipeline class
 BuildOutgoing() method, 220
 described, 219
 OnAfterConnect() method, 221
 OnAfterDisconnect() method, 221
 OnAfterIncoming() method, 221
 OnAfterOutgoing() method, 221
 OnAfterReconnect() method, 221
 OnBeforeAuthorizeConnect() method, 221
 OnBeforeConnect() method, 221
 OnBeforeDisconnect() method, 221
 OnBeforeIncoming() method, 221
 OnBeforeOutgoing() method, 221
 OnBeforeReconnect() method, 221
 OnIncomingError() method, 221
HubProxy class, 135–136, 138
hubs
 access control in, 182–184
 access from other threads, 103–105, 111–116
 client-side implementation
 communication with server, 135–138
 creating and opening connections to, 134–135
 described, 78
 establishing the connection, 83–85
 generating the proxy, 79–81
 implementing the client without a proxy, 93–96
 JavaScript clients, 79
 logging, 91–92
 manual generation of JavaScript proxies, 81–83

- receiving messages sent from server, 90–91
 - sending additional information, 89–90
 - sending message to server, 86–89
 - state maintenance, 92–93
 - console applications and, 119
 - Dependency Injection, 196–205
 - described, 20, 57–58
 - global configurations, 51
 - intercepting messages in, 218–222
 - progress bar example, 113
 - security and, 50
 - server-side implementation
 - accessing request context information, 71
 - creating hubs, 59–60
 - hub registration and configuration, 58–59
 - managing groups, 72–73
 - notifications of connections and disconnections, 72
 - receiving messages, 60–64
 - sending messages to clients, 64–67
 - sending messages to specific users, 68–69
 - state maintenance, 69–70, 73–78
 - shared drawing board example, 96–101
 - unit testing, 211–215
 - HyperText Transfer Protocol (HTTP)
 - 503 errors, 175
 - described, 5
 - HTTP operations, 5–7
 - inefficiencies of, 6–7
 - synchronous communication, 5–6
- ## I
- IApplicationBuilder interface
 - described, 28–29, 37
 - MapSignalR() method, 29, 59
 - multiplatform SignalR servers, 120
 - ICalculator interface, 209–210
 - IClock interface, 201, 203–204
 - IConnection interface
 - Broadcast() method, 32–34, 106
 - DefaultSignal property, 34
 - Send() method, 34, 36, 106
 - Task object and, 35
 - IConnectionGroupManager interface
 - Add() method, 106
 - described, 35
 - Remove() method, 106
 - Send() method, 106
 - IDataStore interface, 200
 - IDependencyResolver interface, 192
 - IDisposable interface, 200
 - IETF (Internet Engineering Task Force), 9
 - <IFRAME> tag, 14
 - IGroupManager interface, 225
 - IHubConnectionContext interface, 112, 225
 - IHubContext interface, 112
 - IHubPipelineModule interface, 219
 - IHubProxy interface, 137
 - IIS Express, 148
 - IIS (Internet Information Services)
 - recommendations, 174
 - server configuration, 174
 - SignalR support, 20
 - transport negotiation, 49
 - IJavaScriptMinifier interface, 196
 - IJavaScriptProxyGenerator interface, 195
 - IMessageFormatter interface, 201, 205
 - Install-Package command, 25, 28
 - installing SignalR, 25–26
 - integration with other frameworks
 - AngularJS, 230–232
 - described, 223
 - Knockout, 227–230
 - MVC, 226
 - Web API, 226
 - intercepting messages in hubs, 218–222
 - Interlocked class, 74
 - Internet Engineering Task Force (IETF), 9
 - Internet Information Services (IIS)
 - recommendations, 174
 - server configuration, 174
 - SignalR support, 20
 - transport negotiation, 49
 - Internet Relay Chat (IRC), 7
 - Inversion of Control (IoC)
 - described, 191, 200–202
 - Ninject IoC container, 204–205
 - Unity IoC container, 202–204
 - IoC (Inversion of Control)
 - described, 191, 200–202
 - Ninject IoC container, 204–205
 - Unity IoC container, 202–204
 - IoCConfig class, 203
 - iOS platform, 150
 - IPersistentConnectionContext interface, 105

IRC (Internet Relay Chat)

- IRC (Internet Relay Chat), 7–8
- IRepository interface, 200–201
- IRequest interface
 - access control in persistent connections, 181–182
 - accessing information about request context, 71
 - described, 30
 - sending additional information to servers, 46
 - unit testing example, 212
- IService interface, 201
- IUserIdProvider interface, 68–69

J

- JabbR service, 18
- jQuery, 39–40
- JSON format, 34, 41, 185
- JSON with Padding (JSONP), 41–43, 85
- json2.js file, 41
- Json.NET library, 32
- JSONP (JSON with Padding), 41–43, 85
- JustMock framework, 209

K

- Katana open source project, 24, 185–187
- Knockout framework, 227–230
- ko object, 229

L

- Lazy class, 195
- Linux platform, 167–168
- load balancers, 154–155
- logging events, 91–92
- long polling, 12–14

M

- manual dependency injection, 198–200
- mapping
 - hubs, 58
 - persistent connections, 28–29, 59
- MbUnit testing framework, 206
- memory
 - performance counters, 180
 - recommendations, 174
 - server state and, 76

- Memory#bytes performance counter, 180
- Message Bus Allocated Workers performance counter, 178
- Message Bus Busy Workers performance counter, 179
- Message Bus Messages Published/Sec performance counter, 179
- Message Bus Messages Published Total performance counter, 179
- Message Bus Messages Received/Sec performance counter, 179
- Message Bus Messages Received Total performance counter, 179
- Message Bus Subscribers Current performance counter, 179
- Message Bus Subscribers/Sec performance counter, 179
- Message Bus Subscribers Total performance counter, 179
- Message Bus Topics Current performance counter, 179
- MessageFormatter class, 201
- messages
 - intercepting in hubs, 218–222
 - receiving from clients, 59
 - receiving from servers, 45–46, 90–91
 - recommendations, 173
 - sending to clients, 32–34, 64–67
 - sending to servers, 43–45, 86–88
 - sending to specific users, 68–69
 - tracing requests, 109, 111
- messaging bus, 19, 178–179
- Microsoft Patterns & Practices, 202
- Microsoft.AspNet.SignalR namespace/package, 29, 39, 59
- Microsoft.AspNet.SignalR.Client namespace/package, 25, 130, 140
- Microsoft.AspNet.SignalR.Hubs namespace, 219
- Microsoft.AspNet.SignalR.Infrastructure namespace, 68
- Microsoft.AspNet.SignalR.JS package, 39
- Microsoft.AspNet.SignalR.Redis namespace/package, 169
- Microsoft.AspNet.SignalR.SelfHost package, 119, 122
- Microsoft.AspNet.SignalR.ServiceBus namespace/package, 163
- Microsoft.AspNet.SignalR.SqlServer namespace/package, 166

Microsoft.AspNet.SignalR.Utils package, 81, 175
 Microsoft.Owin.Compression package, 24
 Microsoft.Owin.Cors package, 24, 85, 120
 Microsoft.Owin.Host.HttpListener namespace, 24, 118
 Microsoft.Owin.Host.SystemWeb package, 118
 Microsoft.Owin.Security namespace, 24
 Microsoft.Owin.Security.ActiveDirectory module, 185
 Microsoft.Owin.Security.Cookies module, 185–187
 Microsoft.Owin.Security.Facebook module, 185
 Microsoft.Owin.Security.Google module, 185
 Microsoft.Owin.Security.Jwt module, 185
 Microsoft.Owin.Security.MicrosoftAccount module, 185
 Microsoft.Owin.Security.OAuth module, 185
 Microsoft.Owin.Security.Twitter module, 185
 Microsoft.Owin.StaticFiles package, 24
 middleware, OWIN standard, 22–24, 185–186
 Model-View-View-Model (MVVM), 227
 monitoring

- connections at server example, 106–111
- performance in SignalR services, 175–180

 Mono project, 126–129, 149
 MonoDevelop platform, 128, 149
 Moq framework, 209–211
 multiplatform applications

- described, 117
- multiplatform SignalR clients, 129–150
- multiplatform SignalR servers, 117–128

 MVC framework

- Authorize attribute, 183
- deployment of web applications, 152
- OWIN middleware and, 37
- positioning in ASP.NET stack, 17
- SignalR integration with, 226

 MVVM (Model-View-View-Model), 227

N

negotiation, transport, 18, 48–49
 Ninject IoC container, 201, 204–205
 NServiceBus, 170
 NuGet package manager

- Get-Package command, 25
- Install-package command, 25, 28
- packages supported, 25, 39

 # of current logical Threads performance counter, 180

of current physical Threads performance counter, 180
 NUnit testing framework, 206

O

OAuth 2.0 authentication, 185
 Office Web Apps, 17
 Office365, 17
 Open Web Interface for .NET (OWIN)

- bifurcation in pipeline, 85
- client authentication, 184–190
- described, 21–24
- Startup class, 28–29, 37–38, 122
- system architecture, 185

 OpenID, 185
 overloaded methods, 61
 OWIN (Open Web Interface for .NET)

- bifurcation in pipeline, 85
- client authentication, 184–190
- key features, 21–24
- mapping and configuring persistent connections, 28–29
- multiplatform SignalR servers, 118–119
- Startup class, 28–29, 37–38, 58, 122, 189
- system architecture, 185

P

performance counters, 176–180
 performance improvement

- described, 173–174
- monitoring performance, 175–180
- server configuration, 174–175

 Performance Monitor, 175–177
 persistent connections

- access control in, 181–182
- access from other threads, 103–111
- adjusting configuration parameters, 50–51
- client-side implementation
 - creating and opening, 130–133
 - cross-domain connection support, 41–43
 - described, 38
 - events available at, 47
 - initiating using JavaScript client, 38–40
 - older browser support, 41
 - receiving messages, 45–46

PersistentConnection class

- sending additional information to server, 46–47
- sending and receiving data, 133–134
- sending messages, 43–45
- console applications and, 119
- described, 19–20, 27
- forever frame and, 14
- impression of, 18
- monitoring at server example, 106–111
- server-side implementation
 - asynchronous event processing, 34–35
 - configuring, 28–29
 - connection groups, 35–37
 - events of, 30–32
 - mapping, 28–29, 59
 - OWIN startup class, 37–38
 - sending messages to clients, 32–34
- tracking visitors example, 51–55
- transport negotiation, 48–49
- unit testing, 215–218
- WebSockets support, 9

PersistentConnection class

- access control and, 181–182
- AuthorizeRequest() method, 182
- Connection property, 32, 216
- described, 27–29
- external access using, 105–106
- Groups property, 35
- Initialize() method, 192
- manual dependency injection and, 199–200
- OnConnected() method, 30, 216, 218
- OnDisconnected() method, 30, 32
- OnReceived() method, 30–31, 43, 60, 216
- OnReconnected() method, 31
- OnRejoiningGroups() method, 31
- unit testing example, 215–218

piggy backing technique, 8

polling technique

- adaptive intervals, 8
- advantages/disadvantages, 7–8
- defined, 7
- long polling, 12–14

PrincipalUserIdProvider class, 68–69

Processor information / % Processor time

- performance counter, 180

progress bar example, 113

proxies

- automatic, 79, 83
- dynamic, 79–81

- generating, 79–81
- generating manually, 81–83
- receiving messages sent from servers, 90–91
- sending messages to servers, 86–88

pull model, 5, 7

push concept

- described, 8–9
- forever frame and, 14–15
- long polling and, 12–14
- Server-Sent Events, 11–15
- transport negotiation, 48
- WebSockets standard, 9–11
- XHR streaming and, 12

R

RabbitMQ, 170

real-time multiplatform applications

- described, 117
- multiplatform SignalR clients, 129–150
- multiplatform SignalR servers, 117–128

Redis storage system

- activating the backplane, 169–170
- installing, 167–169
- SignalR backplane support, 157, 167

Remote Procedure Call (RPC), 20, 38

request-response schemas, 5

Requests Current performance counter, 180

Requests Queued performance counter, 180

Requests Rejected performance counter, 180

resource consumption, long polling and, 13–14

reverse AJAX, 12

Rhino Mocks framework, 209

RPC (Remote Procedure Call), 20, 38

S

SaaS (Software as a Service), 158

scalability

- custom backplanes, 170–172
- death by success and, 153
- horizontal, 154–155
- improving performance in SignalR services, 173–180
- scaling on backplanes, 159–170
- session affinity and, 155
- in SignalR, 155–159
- state storage and, 76, 78

- vertical, 153–154
- scale-out approach
 - described, 154–155
 - performance counters, 179
- scale-up approach, 153, 155
- Scaleout Errors/Sec performance counter, 179
- Scaleout Errors Total performance counter, 179
- Scaleout Message Bus Messages Received/Sec performance counter, 179
- Scaleout Send Queue Length performance counter, 179
- Scaleout Streams Buffering performance counter, 179
- Scaleout Streams Open performance counter, 179
- Scaleout Streams Total performance counter, 179
- ScaleoutMessageBus class, 170–172
- Secure Sockets Layer (SSL), 184
- security
 - authorization in SignalR, 181–190
 - client communications and, 9
 - JSONP and, 41
 - OWIN middleware modules, 24
 - public methods within hubs, 50
 - push concept and, 12
 - state storage and, 78
- server push concept
 - described, 8–9
 - forever frame and, 14–15
 - long polling and, 12–14
 - Server-Sent Events, 11–15
 - transport negotiation, 48
 - WebSockets standard, 9–11
 - XHR streaming and, 12
- Server-Sent Events (API Event Source)
 - additional techniques supporting, 15
 - described, 11–12
 - forever frame and, 14
 - push and, 12–15
 - transport negotiation, 48
- server-side processing
 - hubs
 - accessing request context information, 71
 - creating, 59–60
 - managing groups, 72–73
 - notifications of connections and disconnections, 72
 - progress bar example, 115–116
 - receiving messages from clients, 60–64
 - registration and configuration, 58–59
 - sending messages to clients, 64–67
 - sending messages to specific users, 68–69
 - shared drawing board example, 100
 - state maintenance, 69–70, 73–78
 - multiplatform applications
 - described, 117
 - non-web applications, 118–126
 - platforms other than Windows, 126–129
 - OWIN standard, 22–23
 - persistent connections
 - asynchronous event processing, 34–35
 - configuring, 28–29
 - connection groups, 35–37
 - events of, 30–32
 - mapping, 28–29
 - OWIN startup class, 37–38
 - sending messages to clients, 32–34
 - tracing requests, 109
 - tracking visitors example, 54
 - recommendations, 174
- Service Locator pattern, 191, 196, 200
- ServiceBase class, 123
- ServiceBusScaleoutConfiguration class, 164
- session affinity, 155
- session variables, 174
- shared drawing board example, 96–101
- SharePoint, SignalR support, 17
- SignalR
 - abstraction levels, 19–20, 27, 57
 - adjusting configuration parameters, 50–51
 - advantages, 19
 - authorization in, 181–190
 - Authorize attribute, 183
 - background, 17–18
 - Dependency Injection, 196–205
 - extensible framework, 191–196
 - improving performance in services, 173–180
 - installing, 25–26
 - integration with other frameworks, 223–232
 - intercepting messages in hubs, 218–222
 - Katana open source project, 24
 - key features, 18–19
 - monitoring performance in services, 175–180
 - multiplatform clients, 129–150
 - multiplatform servers, 117–129
 - OWIN standard, 21–24
 - positioning in ASP.NET stack, 17

SignalR.exe tool

- scalability in, 155–159
- supported platforms, 20–21
- unified programming model, 19
- unit testing, 205–218
- SignalR.exe tool, 81–82, 175–176
- Single Page Applications (SPA) framework, 230
- SkyDrive, SignalR support, 17
- SMTP, 8
- Software as a Service (SaaS), 158
- SPA (Single Page Applications) framework, 230
- SQL Server
 - activating the backplane, 166–167
 - configuring the database, 165–166
 - SignalR backplane support, 157, 165
- SQL Server Management Studio, 165
- SSL (Secure Sockets Layer), 184
- Startup class (OWIN)
 - Configuration() method, 28–29, 37, 58, 189
 - described, 28–29, 37–38, 58
 - multiplatform SignalR servers, 122
- state maintenance (hubs)
 - client-side, 92–93
 - recommendations, 174
 - server-side, 69–70, 73–78
- static files, OWIN middleware modules, 24
- sticky sessions, 155
- StructureMap IoC container, 201
- submissions
 - monitoring, 15
 - recommendations, 173
- subscriptions, managing, 15
- synchronous communication, HTTP operations, 5–6
- System.Collections.Concurrent namespace, 74
- System.Web.Http namespace, 182, 225
- System.Web.Mvc namespace, 226

T

- Task object
 - asynchronous event processing, 35
 - communicating with server using hubs, 136
 - creating and opening persistent connections, 132
 - receiving messages, 63
 - sending and receiving data using persistent connections, 133
 - sending messages to clients, 67
- testing
 - of hubs, 211–215

- of persistent connections, 215–218
- unit testing, 205–218
- Windows Phone solutions, 148
- TextWriter class, 139
- TraceLevel property (Connection class)
 - All value, 139
 - Events value, 138
 - Messages value, 138
 - None value, 139
 - StateChanges value, 138
- tracing requests, 109, 111
- tracking visitors example, 51–55
- transport negotiation, 18, 48–49
- Twitter OAuth 2.0 authentication, 185
- TypeMock framework, 209

U

- Ubuntu platform, 168
- unified programming model, 19
- unit testing
 - described, 205–210
 - of hubs, 211–215
 - of persistent connections, 215–218
- Unity IoC container, 201–204

V

- vertical scalability, 153–154
- View-Model class, 227–229
- Visual Studio
 - Authorize attribute and, 182–183
 - Browser Link feature, 18
 - implementing native clients, 149
 - testing framework, 206–210

W

- W3C (World Wide Web Consortium), 9
- Web API
 - Authorize attribute, 183
 - deployment of web applications, 152
 - OWIN middleware and, 37
 - positioning in ASP.NET stack, 17
 - SignalR integration with, 226
- web applications
 - deployment scenarios, 152

- MVC framework and, 226
- OWIN standard, 22–23
- recommendations, 174
- Web Forms
 - positioning in ASP.NET stack, 17
 - update dependencies, 21
- web framework (OWIN), 22–23
- Web Pages, positioning in ASP.NET stack, 17
- Web Tokens (JSON), 185
- WebApp class, 121
- WebSockets standard
 - described, 9–11
 - persistent connections and, 29
 - transport negotiation, 48–49
- Windows Azure
 - activating the backplane, 163–164
 - Azure Management Tool, 163
 - cache system, 76
 - cloud services, 154
 - configuring the service, 159–163
 - Katana project and, 185
 - SignalR support, 20, 157, 159
- Windows Phone clients, 147–149
- Windows platforms
 - application example, 142–149
 - recommendations, 174
 - Redis and, 168–169
 - SignalR support, 20
- Windows Server platform, 20
- Windows Services, 122–126
- Windsor Castle IoC container, 201
- World Wide Web Consortium (W3C), 9
- ws://protocol, 11

X

- Xamarin Studio, 149–150
- XAML application example, 142–147
- XHR streaming, 12
- XUnit testing framework, 206

buffering

- multiplatform SignalR servers, 120
- transport negotiation, 48–49
- WebSockets support, 10
- buffering, 173

C

- C++ client, 149
- C# language
 - application example, 142–147
 - async/await construct, 35, 63, 67, 133, 136
- Calculator class, 208–209
- camel casing style, 86
- CDN (Content Delivery Networks), 154
- client authentication, 184–190
- client/server architecture
 - HTTP operations, 5–7
 - messaging bus, 19
 - negotiation, 18
 - polling technique, 7–8
 - push concept, 8–15
- client-side processing
 - console application example, 140–141
- hubs
 - communication with server, 135–138
 - creating and opening connections to, 134–135
 - described, 78
 - establishing the connection, 83–85
 - generating the proxy, 79–81
 - implementing without a proxy, 93–96
 - JavaScript clients, 79
 - logging, 91–92
 - manual generation of JavaScript proxies, 81–83
 - progress bar example, 114–115
 - receiving messages sent from server, 90–91
 - security and, 9
 - sending additional information, 89–90
 - sending message to server, 86–89
 - shared drawing board example, 98
 - state maintenance, 92–93
- multiplatform applications
 - accessing services from non-web clients, 130–149
 - consumption of services from other platforms, 149–150
 - described, 129–130
 - persistent connections
 - creating and opening, 130–133
 - cross-domain connection support, 41–43
 - described, 27, 38
 - events available at, 47
 - initiating using JavaScript client, 38–40
 - older browser support, 41
 - receiving messages, 45–46
 - sending additional information to server, 46–47
 - sending and receiving data, 133–134
 - sending messages, 43–45
 - tracing requests, 111
 - tracking visitors example, 53
 - recommendations, 174
 - Windows Phone problems, 147–149
- Clients property (Hub class)
 - All value, 64–65, 112, 212
 - AllExcept value, 65, 112
 - Caller value, 66, 70, 138
 - Client value, 66, 68, 112, 214
 - Clients value, 66
 - Group value, 66, 72, 112
 - Groups value, 67, 73, 112
 - Others value, 66
 - OthersInGroups value, 67, 72
 - User value, 67
- cloud services, 154
- clustering servers, 173
- Comet, 12
- compression, OWIN middleware modules, 24
- ConcurrentDictionary class, 74
- .config file, 37, 175
- Connection class
 - AddClientCertificate() method, 131
 - Closed event, 134
 - ConnectionId property, 40, 131
 - ConnectionSlow event, 47, 134
 - CookieContainer property, 131–132
 - Credentials property, 132
 - Error event, 44, 47, 134
 - Headers property, 132
 - Logging property, 48
 - Proxy property, 132
 - QueryString property, 89
 - Received event, 45, 47, 134
 - Reconnected event, 47
 - Reconnecting event, 134

- Send() method, 43–44, 133, 142–149
- Start() method, 39–41, 48, 83–84, 132–133
- State property, 132
- StateChanged event, 47, 134
- Stop() method, 40
- Trace() method, 138
- TraceLevel property, 138–139
- TraceWriter property, 139
- connection groups, 35–37
- Connection Messages Received/Sec performance counter, 178
- Connection Messages Received Total performance counter, 178
- Connection Messages Sent/Sec performance counter, 178
- Connection Messages Sent Total performance counter, 178
- ConnectionConfiguration class, 41
- connectionID identifier, 30–33, 35
- ConnectionMessage structure, 34
- Connections Connected performance counter, 178
- Connections Current performance counter, 178
- Connections Disconnected performance counter, 178
- Connections Reconnected performance counter, 178
- console applications
 - application example, 140–141
 - multiplatform SignalR clients, 130
 - multiplatform SignalR servers, 119–122
- Content Delivery Networks (CDN), 154
- Context property (Hub class)
 - ConnectionId value, 71, 212
 - described, 72, 89
 - Headers value, 71
 - QueryString value, 71, 89
 - RequestCookies value, 71
 - unit testing example, 212
 - User value, 71
- Controller class, 226
- cookie-based authentication, 46, 185
- CORS (Cross Origin Resource Sharing)
 - multiplatform SignalR servers, 120
 - OWIN middleware modules, 24
 - persistent connections and, 41–42
 - responding to requests, 85
- Creative Commons license, 21
- cross-domain connections, 41–43
- Cross Origin Resource Sharing (CORS)
 - multiplatform SignalR servers, 120

- OWIN middleware modules, 24
- persistent connections and, 41–42
- responding to requests, 85

D

- death by success, 153
- DefaultDependencyResolver class, 192, 202
- Deferred object (jQuery), 40
- Dependency Injection (DI)
 - described, 191, 196–198
 - Inversion of Control containers, 200–205
 - manual, 198–200
 - releasing dependencies, 200
- Dependency Resolver, 69, 191–196
- deployment scenarios
 - death by success and, 153
 - described, 151–153
 - improving performance in SignalR services, 173–180
 - OWIN standard example, 23
- DI (Dependency Injection)
 - described, 191, 196–198
 - Inversion of Control containers, 200–205
 - manual, 198–200
 - releasing dependencies, 200
- Dictionary class, 74
- dynamic proxies, 79–81

E

- EchoConnection class, 29, 39, 217
- Edwards, Damian, 17
- Errors: All/Sec performance counter, 178
- Errors: All Total performance counter, 178
- Errors: Hub Invocation/Sec performance counter, 178
- Errors: Hub Invocation Total performance counter, 178
- Errors: Hub Resolution/Sec performance counter, 178
- Errors: Hub Resolution Total performance counter, 178
- Errors: Transport/Sec performance counter, 178
- Errors: Transport Total performance counter, 178
- events
 - asynchronous processing, 34–35
 - described, 30–32
 - logging, 91–92, 138–139

EventSource object

EventSource object, 12
extensible framework, 191–196

F

Facebook OAuth authentication, 185
FakeltEasy framework, 209
Fakes framework, 209
Fiddler tool, 49, 84
FileSystemWatcher class, 172
forever frame
 described, 12, 14–15
 persistent connections and, 29
 transport negotiation, 48–49
Fowler, David, 17

G

garbage collection, 180
Get-Package command, 25
GlobalHost class
 Configuration property, 50–51
 ConnectionManager property, 111, 224–225
 DependencyResolver property, 192–193
 HubPipeline property, 219
globally unique identifier (GUID), 30
GUID (globally unique identifier), 30

H

horizontal scalability, 154–155
hosting
 multiplatform SignalR servers
 console applications, 119–122
 described, 118
 platforms other than Windows, 126–129
 Windows service, 122–126
 OWIN standard, 22–24, 118–119
HTML <IFRAME> tag, 14
HTML5, 9
HTTP (HyperText Transfer Protocol)
 503 errors, 175
 described, 5
 HTTP operations, 5–7
 inefficiencies of, 6–7
 polling, 7–8, 12–14
 Server-Side Events, 11–12

 synchronous communication, 5–6
HTTP push, 12
HttpContext class, 184
HttpListener class, 118–119, 123
Hub class
 Clients property, 64–68, 70, 72–73, 112, 138, 212, 214
 Context property, 71–72, 89, 212
 OnConnected() method, 211, 213
HubConnection class
 Closed event, 135
 CreateHubProxy() method, 135
 described, 134–135
 Start() method, 135
 StateChanged event, 135
 Stop() method, 135
HubException exception, 88
HubPipeline class
 BuildOutgoing() method, 220
 described, 219
 OnAfterConnect() method, 221
 OnAfterDisconnect() method, 221
 OnAfterIncoming() method, 221
 OnAfterOutgoing() method, 221
 OnAfterReconnect() method, 221
 OnBeforeAuthorizeConnect() method, 221
 OnBeforeConnect() method, 221
 OnBeforeDisconnect() method, 221
 OnBeforeIncoming() method, 221
 OnBeforeOutgoing() method, 221
 OnBeforeReconnect() method, 221
 OnIncomingError() method, 221
HubProxy class, 135–136, 138
hubs
 access control in, 182–184
 access from other threads, 103–105, 111–116
 client-side implementation
 communication with server, 135–138
 creating and opening connections to, 134–135
 described, 78
 establishing the connection, 83–85
 generating the proxy, 79–81
 implementing the client without a proxy, 93–96
 JavaScript clients, 79
 logging, 91–92
 manual generation of JavaScript proxies, 81–83

- receiving messages sent from server, 90–91
 - sending additional information, 89–90
 - sending message to server, 86–89
 - state maintenance, 92–93
 - console applications and, 119
 - Dependency Injection, 196–205
 - described, 20, 57–58
 - global configurations, 51
 - intercepting messages in, 218–222
 - progress bar example, 113
 - security and, 50
 - server-side implementation
 - accessing request context information, 71
 - creating hubs, 59–60
 - hub registration and configuration, 58–59
 - managing groups, 72–73
 - notifications of connections and disconnections, 72
 - receiving messages, 60–64
 - sending messages to clients, 64–67
 - sending messages to specific users, 68–69
 - state maintenance, 69–70, 73–78
 - shared drawing board example, 96–101
 - unit testing, 211–215
 - HyperText Transfer Protocol (HTTP)
 - 503 errors, 175
 - described, 5
 - HTTP operations, 5–7
 - inefficiencies of, 6–7
 - synchronous communication, 5–6
- ## I
- IApplicationBuilder interface
 - described, 28–29, 37
 - MapSignalR() method, 29, 59
 - multiplatform SignalR servers, 120
 - ICalculator interface, 209–210
 - IClock interface, 201, 203–204
 - IConnection interface
 - Broadcast() method, 32–34, 106
 - DefaultSignal property, 34
 - Send() method, 34, 36, 106
 - Task object and, 35
 - IConnectionGroupManager interface
 - Add() method, 106
 - described, 35
 - Remove() method, 106
 - Send() method, 106
 - IDataStore interface, 200
 - IDependencyResolver interface, 192
 - IDisposable interface, 200
 - IETF (Internet Engineering Task Force), 9
 - <IFRAME> tag, 14
 - IGroupManager interface, 225
 - IHubConnectionContext interface, 112, 225
 - IHubContext interface, 112
 - IHubPipelineModule interface, 219
 - IHubProxy interface, 137
 - IIS Express, 148
 - IIS (Internet Information Services)
 - recommendations, 174
 - server configuration, 174
 - SignalR support, 20
 - transport negotiation, 49
 - JavaScriptMinifier interface, 196
 - JavaScriptProxyGenerator interface, 195
 - IMessageFormatter interface, 201, 205
 - Install-Package command, 25, 28
 - installing SignalR, 25–26
 - integration with other frameworks
 - AngularJS, 230–232
 - described, 223
 - Knockout, 227–230
 - MVC, 226
 - Web API, 226
 - intercepting messages in hubs, 218–222
 - Interlocked class, 74
 - Internet Engineering Task Force (IETF), 9
 - Internet Information Services (IIS)
 - recommendations, 174
 - server configuration, 174
 - SignalR support, 20
 - transport negotiation, 49
 - Internet Relay Chat (IRC), 7
 - Inversion of Control (IoC)
 - described, 191, 200–202
 - Ninject IoC container, 204–205
 - Unity IoC container, 202–204
 - IoC (Inversion of Control)
 - described, 191, 200–202
 - Ninject IoC container, 204–205
 - Unity IoC container, 202–204
 - IoCConfig class, 203
 - iOS platform, 150
 - IPersistentConnectionContext interface, 105

IRC (Internet Relay Chat)

- IRC (Internet Relay Chat), 7–8
- IRepository interface, 200–201
- IRequest interface
 - access control in persistent connections, 181–182
 - accessing information about request context, 71
 - described, 30
 - sending additional information to servers, 46
 - unit testing example, 212
- IService interface, 201
- IUserIdProvider interface, 68–69

J

- JabbR service, 18
- jQuery, 39–40
- JSON format, 34, 41, 185
- JSON with Padding (JSONP), 41–43, 85
- json2.js file, 41
- Json.NET library, 32
- JSONP (JSON with Padding), 41–43, 85
- JustMock framework, 209

K

- Katana open source project, 24, 185–187
- Knockout framework, 227–230
- ko object, 229

L

- Lazy class, 195
- Linux platform, 167–168
- load balancers, 154–155
- logging events, 91–92
- long polling, 12–14

M

- manual dependency injection, 198–200
- mapping
 - hubs, 58
 - persistent connections, 28–29, 59
- MbUnit testing framework, 206
- memory
 - performance counters, 180
 - recommendations, 174
 - server state and, 76

- Memory#bytes performance counter, 180
- Message Bus Allocated Workers performance counter, 178
- Message Bus Busy Workers performance counter, 179
- Message Bus Messages Published/Sec performance counter, 179
- Message Bus Messages Published Total performance counter, 179
- Message Bus Messages Received/Sec performance counter, 179
- Message Bus Messages Received Total performance counter, 179
- Message Bus Subscribers Current performance counter, 179
- Message Bus Subscribers/Sec performance counter, 179
- Message Bus Subscribers Total performance counter, 179
- Message Bus Topics Current performance counter, 179
- MessageFormatter class, 201
- messages
 - intercepting in hubs, 218–222
 - receiving from clients, 59
 - receiving from servers, 45–46, 90–91
 - recommendations, 173
 - sending to clients, 32–34, 64–67
 - sending to servers, 43–45, 86–88
 - sending to specific users, 68–69
 - tracing requests, 109, 111
- messaging bus, 19, 178–179
- Microsoft Patterns & Practices, 202
- Microsoft.AspNet.SignalR namespace/package, 29, 39, 59
- Microsoft.AspNet.SignalR.Client namespace/package, 25, 130, 140
- Microsoft.AspNet.SignalR.Hubs namespace, 219
- Microsoft.AspNet.SignalR.Infrastructure namespace, 68
- Microsoft.AspNet.SignalR.JS package, 39
- Microsoft.AspNet.SignalR.Redis namespace/package, 169
- Microsoft.AspNet.SignalR.SelfHost package, 119, 122
- Microsoft.AspNet.SignalR.ServiceBus namespace/package, 163
- Microsoft.AspNet.SignalR.SqlServer namespace/package, 166

Microsoft.AspNet.SignalR.Utils package, 81, 175
 Microsoft.Owin.Compression package, 24
 Microsoft.Owin.Cors package, 24, 85, 120
 Microsoft.Owin.Host.HttpListener namespace, 24, 118
 Microsoft.Owin.Host.SystemWeb package, 118
 Microsoft.Owin.Security namespace, 24
 Microsoft.Owin.Security.ActiveDirectory module, 185
 Microsoft.Owin.Security.Cookies module, 185–187
 Microsoft.Owin.Security.Facebook module, 185
 Microsoft.Owin.Security.Google module, 185
 Microsoft.Owin.Security.Jwt module, 185
 Microsoft.Owin.Security.MicrosoftAccount module, 185
 Microsoft.Owin.Security.OAuth module, 185
 Microsoft.Owin.Security.Twitter module, 185
 Microsoft.Owin.StaticFiles package, 24
 middleware, OWIN standard, 22–24, 185–186
 Model-View-View-Model (MVVM), 227
 monitoring

- connections at server example, 106–111
- performance in SignalR services, 175–180

 Mono project, 126–129, 149
 MonoDevelop platform, 128, 149
 Moq framework, 209–211
 multiplatform applications

- described, 117
- multiplatform SignalR clients, 129–150
- multiplatform SignalR servers, 117–128

 MVC framework

- Authorize attribute, 183
- deployment of web applications, 152
- OWIN middleware and, 37
- positioning in ASP.NET stack, 17
- SignalR integration with, 226

 MVVM (Model-View-View-Model), 227

N

negotiation, transport, 18, 48–49
 Ninject IoC container, 201, 204–205
 NServiceBus, 170
 NuGet package manager

- Get-Package command, 25
- Install-package command, 25, 28
- packages supported, 25, 39

 # of current logical Threads performance counter, 180

of current physical Threads performance counter, 180
 NUnit testing framework, 206

O

OAuth 2.0 authentication, 185
 Office Web Apps, 17
 Office365, 17
 Open Web Interface for .NET (OWIN)

- bifurcation in pipeline, 85
- client authentication, 184–190
- described, 21–24
- Startup class, 28–29, 37–38, 122
- system architecture, 185

 OpenID, 185
 overloaded methods, 61
 OWIN (Open Web Interface for .NET)

- bifurcation in pipeline, 85
- client authentication, 184–190
- key features, 21–24
- mapping and configuring persistent connections, 28–29
- multiplatform SignalR servers, 118–119
- Startup class, 28–29, 37–38, 58, 122, 189
- system architecture, 185

P

performance counters, 176–180
 performance improvement

- described, 173–174
- monitoring performance, 175–180
- server configuration, 174–175

 Performance Monitor, 175–177
 persistent connections

- access control in, 181–182
- access from other threads, 103–111
- adjusting configuration parameters, 50–51
- client-side implementation
 - creating and opening, 130–133
 - cross-domain connection support, 41–43
 - described, 38
 - events available at, 47
 - initiating using JavaScript client, 38–40
 - older browser support, 41
 - receiving messages, 45–46

PersistentConnection class

- sending additional information to server, 46–47
- sending and receiving data, 133–134
- sending messages, 43–45
- console applications and, 119
- described, 19–20, 27
- forever frame and, 14
- impression of, 18
- monitoring at server example, 106–111
- server-side implementation
 - asynchronous event processing, 34–35
 - configuring, 28–29
 - connection groups, 35–37
 - events of, 30–32
 - mapping, 28–29, 59
 - OWIN startup class, 37–38
 - sending messages to clients, 32–34
- tracking visitors example, 51–55
- transport negotiation, 48–49
- unit testing, 215–218
- WebSockets support, 9

PersistentConnection class

- access control and, 181–182
- AuthorizeRequest() method, 182
- Connection property, 32, 216
- described, 27–29
- external access using, 105–106
- Groups property, 35
- Initialize() method, 192
- manual dependency injection and, 199–200
- OnConnected() method, 30, 216, 218
- OnDisconnected() method, 30, 32
- OnReceived() method, 30–31, 43, 60, 216
- OnReconnected() method, 31
- OnRejoiningGroups() method, 31
- unit testing example, 215–218

piggy backing technique, 8

polling technique

- adaptive intervals, 8
- advantages/disadvantages, 7–8
- defined, 7
- long polling, 12–14

PrincipalUserIdProvider class, 68–69

Processor information / % Processor time

- performance counter, 180

progress bar example, 113

proxies

- automatic, 79, 83
- dynamic, 79–81

- generating, 79–81
- generating manually, 81–83
- receiving messages sent from servers, 90–91
- sending messages to servers, 86–88

pull model, 5, 7

push concept

- described, 8–9
- forever frame and, 14–15
- long polling and, 12–14
- Server-Sent Events, 11–15
- transport negotiation, 48
- WebSockets standard, 9–11
- XHR streaming and, 12

R

RabbitMQ, 170

real-time multiplatform applications

- described, 117
- multiplatform SignalR clients, 129–150
- multiplatform SignalR servers, 117–128

Redis storage system

- activating the backplane, 169–170
- installing, 167–169
- SignalR backplane support, 157, 167

Remote Procedure Call (RPC), 20, 38

request-response schemas, 5

Requests Current performance counter, 180

Requests Queued performance counter, 180

Requests Rejected performance counter, 180

resource consumption, long polling and, 13–14

reverse AJAX, 12

Rhino Mocks framework, 209

RPC (Remote Procedure Call), 20, 38

S

SaaS (Software as a Service), 158

scalability

- custom backplanes, 170–172
- death by success and, 153
- horizontal, 154–155
- improving performance in SignalR services, 173–180
- scaling on backplanes, 159–170
- session affinity and, 155
- in SignalR, 155–159
- state storage and, 76, 78

- vertical, 153–154
- scale-out approach
 - described, 154–155
 - performance counters, 179
- scale-up approach, 153, 155
- Scaleout Errors/Sec performance counter, 179
- Scaleout Errors Total performance counter, 179
- Scaleout Message Bus Messages Received/Sec performance counter, 179
- Scaleout Send Queue Length performance counter, 179
- Scaleout Streams Buffering performance counter, 179
- Scaleout Streams Open performance counter, 179
- Scaleout Streams Total performance counter, 179
- ScaleoutMessageBus class, 170–172
- Secure Sockets Layer (SSL), 184
- security
 - authorization in SignalR, 181–190
 - client communications and, 9
 - JSONP and, 41
 - OWIN middleware modules, 24
 - public methods within hubs, 50
 - push concept and, 12
 - state storage and, 78
- server push concept
 - described, 8–9
 - forever frame and, 14–15
 - long polling and, 12–14
 - Server-Sent Events, 11–15
 - transport negotiation, 48
 - WebSockets standard, 9–11
 - XHR streaming and, 12
- Server-Sent Events (API Event Source)
 - additional techniques supporting, 15
 - described, 11–12
 - forever frame and, 14
 - push and, 12–15
 - transport negotiation, 48
- server-side processing
 - hubs
 - accessing request context information, 71
 - creating, 59–60
 - managing groups, 72–73
 - notifications of connections and disconnections, 72
 - progress bar example, 115–116
 - receiving messages from clients, 60–64
 - registration and configuration, 58–59
 - sending messages to clients, 64–67
 - sending messages to specific users, 68–69
 - shared drawing board example, 100
 - state maintenance, 69–70, 73–78
 - multiplatform applications
 - described, 117
 - non-web applications, 118–126
 - platforms other than Windows, 126–129
 - OWIN standard, 22–23
 - persistent connections
 - asynchronous event processing, 34–35
 - configuring, 28–29
 - connection groups, 35–37
 - events of, 30–32
 - mapping, 28–29
 - OWIN startup class, 37–38
 - sending messages to clients, 32–34
 - tracing requests, 109
 - tracking visitors example, 54
 - recommendations, 174
- Service Locator pattern, 191, 196, 200
- ServiceBase class, 123
- ServiceBusScaleoutConfiguration class, 164
- session affinity, 155
- session variables, 174
- shared drawing board example, 96–101
- SharePoint, SignalR support, 17
- SignalR
 - abstraction levels, 19–20, 27, 57
 - adjusting configuration parameters, 50–51
 - advantages, 19
 - authorization in, 181–190
 - Authorize attribute, 183
 - background, 17–18
 - Dependency Injection, 196–205
 - extensible framework, 191–196
 - improving performance in services, 173–180
 - installing, 25–26
 - integration with other frameworks, 223–232
 - intercepting messages in hubs, 218–222
 - Katana open source project, 24
 - key features, 18–19
 - monitoring performance in services, 175–180
 - multiplatform clients, 129–150
 - multiplatform servers, 117–129
 - OWIN standard, 21–24
 - positioning in ASP.NET stack, 17

SignalR.exe tool

- scalability in, 155–159
- supported platforms, 20–21
- unified programming model, 19
- unit testing, 205–218
- SignalR.exe tool, 81–82, 175–176
- Single Page Applications (SPA) framework, 230
- SkyDrive, SignalR support, 17
- SMTP, 8
- Software as a Service (SaaS), 158
- SPA (Single Page Applications) framework, 230
- SQL Server
 - activating the backplane, 166–167
 - configuring the database, 165–166
 - SignalR backplane support, 157, 165
- SQL Server Management Studio, 165
- SSL (Secure Sockets Layer), 184
- Startup class (OWIN)
 - Configuration() method, 28–29, 37, 58, 189
 - described, 28–29, 37–38, 58
 - multiplatform SignalR servers, 122
- state maintenance (hubs)
 - client-side, 92–93
 - recommendations, 174
 - server-side, 69–70, 73–78
- static files, OWIN middleware modules, 24
- sticky sessions, 155
- StructureMap IoC container, 201
- submissions
 - monitoring, 15
 - recommendations, 173
- subscriptions, managing, 15
- synchronous communication, HTTP operations, 5–6
- System.Collections.Concurrent namespace, 74
- System.Web.Http namespace, 182, 225
- System.Web.Mvc namespace, 226

T

- Task object
 - asynchronous event processing, 35
 - communicating with server using hubs, 136
 - creating and opening persistent connections, 132
 - receiving messages, 63
 - sending and receiving data using persistent connections, 133
 - sending messages to clients, 67
- testing
 - of hubs, 211–215

- of persistent connections, 215–218
- unit testing, 205–218
- Windows Phone solutions, 148
- TextWriter class, 139
- TraceLevel property (Connection class)
 - All value, 139
 - Events value, 138
 - Messages value, 138
 - None value, 139
 - StateChanges value, 138
- tracing requests, 109, 111
- tracking visitors example, 51–55
- transport negotiation, 18, 48–49
- Twitter OAuth 2.0 authentication, 185
- TypeMock framework, 209

U

- Ubuntu platform, 168
- unified programming model, 19
- unit testing
 - described, 205–210
 - of hubs, 211–215
 - of persistent connections, 215–218
- Unity IoC container, 201–204

V

- vertical scalability, 153–154
- View-Model class, 227–229
- Visual Studio
 - Authorize attribute and, 182–183
 - Browser Link feature, 18
 - implementing native clients, 149
 - testing framework, 206–210

W

- W3C (World Wide Web Consortium), 9
- Web API
 - Authorize attribute, 183
 - deployment of web applications, 152
 - OWIN middleware and, 37
 - positioning in ASP.NET stack, 17
 - SignalR integration with, 226
- web applications
 - deployment scenarios, 152

- MVC framework and, 226
- OWIN standard, 22–23
- recommendations, 174
- Web Forms
 - positioning in ASP.NET stack, 17
 - update dependencies, 21
- web framework (OWIN), 22–23
- Web Pages, positioning in ASP.NET stack, 17
- Web Tokens (JSON), 185
- WebApp class, 121
- WebSockets standard
 - described, 9–11
 - persistent connections and, 29
 - transport negotiation, 48–49
- Windows Azure
 - activating the backplane, 163–164
 - Azure Management Tool, 163
 - cache system, 76
 - cloud services, 154
 - configuring the service, 159–163
 - Katana project and, 185
 - SignalR support, 20, 157, 159
- Windows Phone clients, 147–149
- Windows platforms
 - application example, 142–149
 - recommendations, 174
 - Redis and, 168–169
 - SignalR support, 20
- Windows Server platform, 20
- Windows Services, 122–126
- Windsor Castle IoC container, 201
- World Wide Web Consortium (W3C), 9
- ws://protocol, 11

X

- Xamarin Studio, 149–150
- XAML application example, 142–147
- XHR streaming, 12
- XUnit testing framework, 206

About the author



JOSÉ M. AGUILAR is a technical computer systems engineer. For more than 20 years he has been working in the world of software development, mainly with Microsoft technologies. He has worked as a programmer, analyst, head of computer systems in the area of strategic consulting, and technical director of a development company. He is currently a freelancer, providing technical consulting, training, and development services.

He is a recognized expert and periodically writes about subjects related to software development in his blog in English (<http://www.campusmvp.net/blog/author/jose-m-aguilar>) and on Twitter (@jmaguilar). He has been recognized as a Microsoft MVP in ASP.NET/IIS every year since 2011.