



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

SignalR: Real-time Application Development

Utilize real-time functionality in your .NET applications with ease

Einar Ingebrigtsen

www.it-ebooks.info

[PACKT] open source*
PUBLISHING community experience distilled

SignalR: Real-time Application Development

Utilize real-time functionality in your .NET applications with ease

Einar Ingebrigtsen



BIRMINGHAM - MUMBAI

SignalR: Real-time Application Development

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2013

Production Reference: 1310513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-424-1

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author

Einar Ingebrigtsen

Project Coordinator

Joel Goveya

Reviewers

Gustavo Armenta

Roar Flolo

Proofreader

Stephen Silk

Acquisition Editor

Martin Bell

Indexer

Monica Ajmera

Commissioning Editor

Ameya Sawant

Graphics

Ronak Dhruv

Technical Editors

Joyslita Dsouza

Veronica Fernandes

Veena Pagare

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Einar Ingebrigtsen has been working professionally with software since 1994 – ranging from games development on platforms such as Playstation, Xbox, and the PC, to enterprise line of business application development, which he has been working on since 2002. He has always kept the focus on creating great products with great user experiences, putting the user first. Einar is a Microsoft MVP awardee five years in a row, awarded for his work in the Silverlight space with projects such as Balder, a 3D engine for Silverlight. Today Einar runs a company called Dolittle with partners, doing consultancy and building their own products with their own open source projects at the heart of what they do (<http://github.com/dolittle>). Among the clients that Dolittle has been involved with for the last couple of years, include NRK (largest TV broadcaster in Norway), Statoil (Norwegian oil company), Komplett (largest e-commerce in Norway), and Holte (leading Norwegian developer for construction software).

Additionally, Einar loves doing talks at the user groups and conferences, and has been a frequent speaker at Microsoft venues talking about different topics, the last couple of years talking mostly about architecture, code quality, and cloud computing.

It might sound like a cliché, but seriously, without my wife this book could not have happened. Her patience with me and her support is truly what pretty much makes just about anything I turn into a reality. So, Anne Grethe, thank you! I'm also blessed with kids with great patience while I have much going on, but most of all, thanks to them for keeping me playful!

I'd also like to thank my colleagues who have been kind enough to not point to the fact that I've had too much going on in the period of writing this book, with crazy mood-swings. I'll be sure to buy a round the next time we're having a company get together.

About the Reviewers

Gustavo Armenta (Mexico) is an employee at Microsoft Main Campus located in Redmond which is very near to Seattle, WA. His current project is SignalR. He spends a good deal of his time with his team members: Damian Edwards (Australia), David Fowler (Barbados), Stephen Halter (US), Taylor Mullen (US), Abhishek Nanda (India), Xiaohong Tang (China), and Jorge del Conde (Mexico).

Gustavo published a blog entry about how to use SignalR the day after the team released the first version of SignalR. The blog was visited by thousands of developers, and a few of them asked questions about it. He was surprised when his blog was used as reference to explain on Wikipedia what SignalR is. Then, he received an e-mail originating from the blog: Ameya Sawant (India) an editorialist for Packt Publishing who invited him to review a new book on the topic. Gustavo accepted the offer. He started working with Joel Goveya (India), project coordinator for Packt Publishing, and Einar Ingebrigtsen (Norway), the author of this book. To complete this story, we would need to know the location and country of origin of one more person – you, the reader!

As you can realize, I feel very fulfilled with the experiences my professional career is providing me, both in knowledge and lifestyle. Thanks mom and dad, for giving me two amazing siblings.

Roar Flolo started his career as a game developer at Funcom in 1993. Since then he has worked on most game platforms developing real-time 3D graphics, character animation, tools and asset pipelines, AI, networking, physics integration, and vehicle physics.

He worked freelance since 2005 until he co-founded Pixelwerk AS in 2012, where he's currently a CTO working on exciting technologies such as real-time 3D, mobile development, augmented reality, web solutions, WebGL, and graph databases.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Revitalizing the Rich Client	7
The Emperor's new clothes	7
The good old terminal	8
X server	9
Remote desktop	10
Rich clients	11
The Web	12
Full duplex on the Web	12
Events	13
Comet	13
Hand-rolling it all	14
Why?	14
Now what...	14
Think different	14
Summary	15
Chapter 2: Persistent Connections	17
Where are we going?	17
The persistent connection	17
Summary	32
Chapter 3: Hubs	33
Moving up one level	33
Summary	40
Chapter 4: Groups	41
Getting specific with groups	41
Summary	51

Table of Contents

Chapter 5: State	53
Becoming stateful	53
Summary	57
Chapter 6: Security	59
Becoming private	59
Summary	70
Chapter 7: Scaling out	71
Scaling out	71
SQL Server	72
Redis	76
Azure	78
Summary	81
Chapter 8: Monitoring	83
Looking under the covers with monitoring	83
Fiddler	83
Performance counters	85
Summary	89
Chapter 9: Hosting a server using OWIN	91
Self hosting	91
Summary	97
Chapter 10: WinJS and Windows 8	99
WinJS	99
Summary	107
Index	109

Preface

Users are becoming more advanced, technology is moving forward at a rapid pace, and we have more options with cloud computing than ever before. All of this calls for thinking differently about how we make our applications work smarter, deliver faster, and not let the user poll for information but rather have data and changes pushed directly to the user without the user having to interact with anything. With demand on functionality and a world with many platforms, we as developers, are constantly seeking out new ways to make it easier to target all of these; the Web has become the result of this search with HTML5 and JavaScript engines becoming much faster and more widely available. We are relying more and more on the aging HTTP protocol to be our vessel for information to the user. SignalR provides the ability to let you as a developer not have to think about all the details of creating applications that meet all these requirements, and not only for web applications. The purpose of this book is to provide you with a step-by-step guide for starting with SignalR, and also to give an insight into why you would want to SignalR enable your application. Throughout the book we will be building an application with a server component and multiple clients – Web,.NET, and Windows8/WinJS.

What this book covers

Chapter 1, Revitalizing the rich client, will teach you that in order to get started with SignalR and real-time web applications, it is important to understand the motivation behind wanting to have such a technology and approach to application development.

Chapter 2, Persistent Connections, will help you understand that at the core of SignalR sits something called `PersistentConnection`; this is where everything starts. In this chapter you will learn how to get started with it in the backend and consuming it in the frontend.

Chapter 3, Hubs, will move you up one abstraction layer from `PersistentConnection` to something called Hubs. A Hub provides a more natural abstraction for most scenarios. A Hub is easier to write and easier to consume.

Chapter 4, Groups, will explain grouping. Sometimes you want to filter messages so that you can better control which clients get what messages. Grouping is a way in SignalR to accomplish this; you will learn how to deal with this on the server and the client.

Chapter 5, State, will explain you that in addition to sending messages between client and server that are very explicit, you sometimes need to have accompanying metadata or additional data that is cross-cutting. In this chapter, you will learn how state can go back and forth with messages.

Chapter 6, Security, is about how every application needs to take security into consideration. In this chapter, you will learn techniques that you can apply to your SignalR code to secure messages.

Chapter 7, Scaling Out, will show you how to scale out and be able to deal with building applications that scale on multiple servers, not only in an on-premise environment but also in Microsoft's Windows Azure cloud.

Chapter 8, Monitoring, will show you how to monitor messages and look at possible performance bottlenecks. Debugging is part of the everyday life of a developer and, of course, it applies to development with SignalR.

Chapter 9, Hosting a Server Using OWIN, will look at how to self host a simple console application using OWIN. Open Web Interfaces for .NET is an abstraction-enabling web framework to be agnostic about the underlying platform.

Chapter 10, WinJS and Windows8, will port our web client to be a Windows 8 store application through the usage of WinJS.

What you need for this book

In order to get started with SignalR you will need either Visual Studio 2010, any edition, although NuGet, which you also need, is not supported out of the box for the express edition. You can also use Visual Studio 2012; in fact, the book has been written using it. With Visual Studio 2012, the express edition has NuGet out of the box. For NuGet you can go to <http://www.nuget.org> for more information. As a side note, it is also possible to use Mono for development and Xamarin Studio, if you are on a different platform such as Linux or Mac. The code is the same—you just need to manage the references differently. SignalR has Mono packages on NuGet.

Who this book is for

This book is primarily for .NET developers. It also targets those .NET developers who are working on web solutions with HTML and JavaScript.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."


A block of code is set as follows:


```
Var connection = new Connection
    (http://localhost:1599/chat);
```

Any command-line input or output is written as follows:

```
$("#sendButton").click(function() {
    chat.server.send($("#messageTextBox").val());
    $("#messageTextBox").val("");
});
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Revitalizing the Rich Client

This chapter walks you through the history of application development, especially web applications.

The topics covered are:

- Why do we need to think differently about our applications and how they relate to a server
- The different techniques that can be used without something like SignalR
- The protocols and techniques SignalR uses to do what it is doing
- Why do we need something like SignalR
- What UX improvements one could make in an application when using SignalR

At this stage the developer should have the basic knowledge of how SignalR works, and what he/she needs to rethink when designing applications that have a persistent connection to the server.

The Emperor's new clothes

As with fashion, it sometimes feels a lot like history repeats itself in our industry as well. It seems that we are going full circle with how software should be architected. I guess this comes from having pretty much the same goal; software solutions that scale for lots of users, and keep data as updated as possible for all users. What this means is that we probably want to have a shared data source that all clients can get their data from. It also means that we need some kind of network connection for the clients to connect to the centralized data source. The clients are typically distributed across multiple offices, maybe even different geolocations. With different geolocations often comes the challenge of different network types and bandwidth.

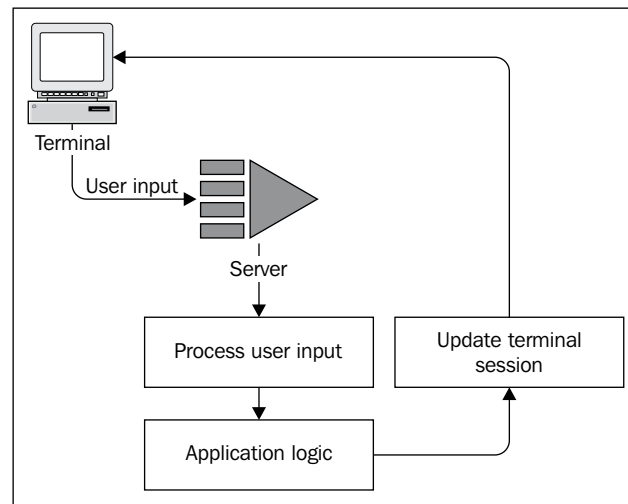
The good old terminal

Before we get to the cool things, it's important to gain some perspective on the problem we're trying to solve. It is, in fact, an old problem dating back to the early days of computers.

Back in the 1970s, in the early computer days, it was quite common to see terminals much like the following one in offices:



The nature of these terminals was to be as primitive as possible. They didn't do any computation, nor did they hold any state. The terminal only reflected what the server wanted the terminal to show on screen, so in many ways they were just really fancy television sets. Any input from the users' keyboard was sent to the server and the server would interpret the user input, update the user's terminal session, and send the screen update back to the terminal as shown in the following figure:

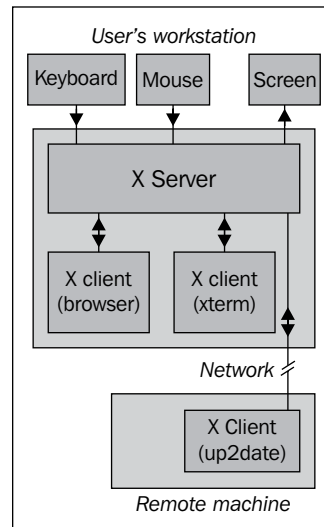


This model proved very helpful, technically, as we developers had everything on our server and didn't have to think about any rich clients holding state making it all the more complex. We only needed to scale the server and deal with potentially multiple servers and keep them in sync or work against a centralized data source. But it didn't prove useful for good user experience. The terminals were limited to text only, and the types of user interfaces one could create were limited, often ending up being very data-centric and keyboard-friendly.

X server

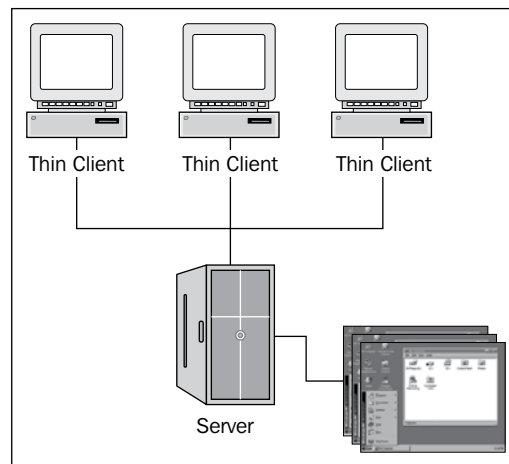
In 1984, the X Window System (commonly known as X11) was introduced, originating from the **Massachusetts Institute of Technology (MIT)**; a graphical user interface system. With it came a network protocol for enabling networked computers to connect to servers in a similar fashion to the terminals of the 70s, and a big step up from the text-based terminal with its graphical capabilities.

As with the terminal solution, the users' input would be sent to a server and the software the user was using would in fact run on that very server, and the result in the graphical user interface would then be sent back to the client machine. Again, leaving the client to be rather dumb and passive.



Remote desktop

Meanwhile in 1998, in the world of Windows, Windows NT 4 got a post-fixed edition of the operating system "Terminal Server Edition". With this edition, Microsoft introduced a new protocol called **Remote Desktop Protocol (RDP)**. It allowed the ability for a client to view another computer's desktop and with NT 4 Terminal Server, the operating system would be able to host multiple desktops for multiple users at the same time. From this remote desktop, clients could then launch any applications they wanted that existed on the server they had connected to. As with the good old terminals, the client computer did not need to be very capable. In fact, this gave birth to an industry, thin-client computers, capable of connecting to an RDP-enabled server.



Rich clients

By having the software running on the server, one puts tremendous pressure on the server(s) and its capability. It must be capable of handling all of the users and their input, which leads to the need for a certain computational power, of course, depending on the application itself.

Sometimes it does not make sense to have it all running on a server. It might not be worth it for your particular application, it might be too costly to try to scale for what you need. It could also be a question of responsiveness; your app might need more responsiveness to make sense to the user. But stepping into the world of a rich, stateful client normally increases the complexity of our solutions. Of course, depending on what we're making.

If we do not have concurrency issues, or data not becoming stale, we don't necessarily have anything that is needing to be solved. Unfortunately, for most line of business software out there, this is not the case. We need to take into consideration that there might be multiple users out there and decide on how to deal with it. We can go down the optimistic path and pretend users seldom run into each other's data and we just overwrite any data that might have been modified while we were making a change to the same piece of data. Or we could go pessimistic and not allow that at all, giving us an exceptional state in the application that we often let our users deal with. One way we can let rich clients deal with this and pretty much leave the problem behind is to use something like TCP sockets and communicate between the clients as they are changing any state. The other respective clients can then pick up the change and alter their own state before the user gets to the point of saving, maybe even notify the user that someone else has modified it.

The Web

And here we are again, back to the dumb client. Our web browsers have served as the passive client for years. The model is frighteningly similar to the terminal solution of the 70s; a dumb client capable of taking input and rendering whatever comes across the network and a rich server doing all of the work.

Hypertext Transfer Protocol (HTTP) is what makes up the Web. It surfaced for the first time in 1991 and basically describes a protocol for sending a request to a server and the server sending a response. The protocol is stateless and you will need to keep state either on the server or on the client. Within the protocol today, there are well-defined methods that can be used, such as POST, GET, PUT, DELETE, and more. These methods let us describe what we are doing. Although a well-defined and rich protocol, it has nothing defined in it to let clients be persistently connected. You can read more about HTTP at <http://en.wikipedia.org/wiki/Http>.

With web browser capabilities increasing over time, we've watched them go from very passive to rich clients. The mid 2000s gave us the buzz often referred to as Web 2.0 and AJAX; Asynchronous JavaScript and XML. At the core of this JavaScript is something called **XHR (XMLHttpRequest)**, making it programmatically accessible to call the server from the client without user interaction. The technique leverages HTTP and you find that instead of getting whole web pages, you get parts, or just the data and put the data into the already rendered web page. You can find more details about AJAX at [http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)).

Modern web applications are turning into a hybrid of rich client and thin client; very capable but they shouldn't do it all—we do need server-side logic. A true step in the right direction would be letting the client be good at what it is good at and likewise, with the server, the correct step would be separating the concerns of the two tiers.

Now that we have all of this power in the browser, we quickly run into similar problems we have with regular rich clients; state on the client.

Full duplex on the Web

With the evolution going backwards into the future, meaning that we are now at the point where we need the same kind of connectivity we needed for rich desktop applications, the demand is that applications live on the Web. With the users, demands come technical challenges. The Web was not built for this; the Web is based on a request/response pattern. The browser goes to a specific URL and a server spits out a response.

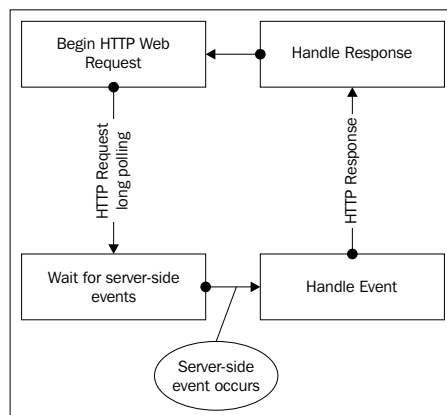
One of the things that the W3C organization has done to accommodate this need is the standardization of something called WebSocket; full-duplex communication channels over a single TCP connection. A very good initiative is something that will be supported by all browser vendors as well as web servers. The challenge with it getting a broad adoption is on the infrastructure that makes up the Web. The entire infrastructure has been optimized for the request/response pattern, and steady connections making a point-to-point connection between two computers, and all of a sudden scalability becomes an issue. So, in many cases, it might not be the best solution.

Events

Another initiative was by Opera, the Norwegian browser vendor called server-sent events, now being standardized by W3C. This gives us the opportunity to push events from the server to any clients that are connected. Combining it with the regular HTTP request/response, we are able to meet the requirement of rich applications. You can read more about server-sent events at http://en.wikipedia.org/wiki/Server-sent_events.

Comet

Not changing the subject just yet, but a technique called Comet has also been applied with great success. The basis of this is to utilize something called long polling HTTP requests. One opens an HTTP request to the server from the client; the server does not return anything until it has something to return, like an event that happens on the server. When the response has been received, the client starts a new long polling connection and keeps on doing so forever. This simulates a full-duplex connection, and scales very well with the existing infrastructure of the Web. You can read more about Comet [http://en.wikipedia.org/wiki/Comet_\(programming\)](http://en.wikipedia.org/wiki/Comet_(programming)).



Hand-rolling it all

You're probably getting where I am going with this. The techniques described previously, are some of the techniques SignalR utilizes. Surely, the techniques and standards are well known and nothing holds you back from working with them directly. But this is where SignalR comes in and saves the day.

Why?

The most important thing to ask in software development is *why*? (http://en.wikipedia.org/wiki/5_whys). Why do we want all this? What is it that we're really trying to solve? We're trying to make software more collaborative, and make users work together without having artificial technical limitations to this collaboration. In fact, why not have any change done occur in real time with all users collaborating?

Now what...

SignalR represents an abstraction for all the techniques that it supports today, and with it also the extensibility of supporting future techniques that might come along. Built in sits a fallback mechanism which enables it to pick the best solution for your app, based on the server and connecting client capabilities. In addition SignalR provides great mechanisms for scaling out in a multi-server environment, enabling applications to be blissfully unaware of which server they are running on and just work with the same abstraction as if it was only one server.

Think different

Apple coined the phrase "Think Different" back in 1997. The phrase in itself forces you to think differently, since it is grammatically incorrect. With all the asynchronous operations and events going into a technology like SignalR, one really has to think in different ways, but different in a good way. Good for the users, as we are now forced to create user experiences that are non-blocking. Of course, you as a developer can force locks onto the users, but I would argue it would be easier not to and rather approach building the user interface in different ways.

For instance, one of the things we tend to build into our apps is the notion of concurrency and stale data. We don't want to run the risk of two users updating exactly the same data and one client not having the updated data from the other user. Often, we leave this so our users get bizarre error messages that they usually won't understand. A better solution would be to have all of the data on all user screens be updated as they are looking at it, maybe even make them aware in a subtle way of any changes that happened from the other user(s).

Summary

The world of software development is in my opinion a very fun place to be, with a lot of things happening all the time. Looking back at the last 30-40 years, we've come a long way; although, we do tend to go around in loops, and that comes, in my opinion, from us trying to solve the same problem in the same manner every time with the modern techniques available at that time.

With today's technologies and users demands, there are great opportunities to change the pattern a bit and do things differently – think differently about the problems and make things even more user friendly. SignalR represents one of these opportunities. It holds an abstraction that makes it a lot easier to embrace the different approaches that we can have when creating great user experience. It does, at the same time, represent robustness and with Microsoft committing resources to the project, it now also represents a commitment from one of the largest software vendors in the world. Another aspect SignalR represents is Microsoft's growing commitment to open source. All of the source code for SignalR is freely available, and they are also accepting outside commits.

In the next chapter we'll get started with SignalR and the entry level for getting started, that is `PersistentConnection`. With this, you will be able to keep a connection open persistently between the client and the server.

2

Persistent Connections

This chapter will cover the basics of getting a client connected to the server and how messaging works.

The topics covered include:

- Setting up a web application with SignalR
- Exposing a persistent connection to the server
- Consuming the connection in a .NET client
- Consuming the connection in a JavaScript client
- Enabling WebSockets on Windows Server 2012

At this stage the developer should be able to have two client types and a server connected together.

Where are we going?

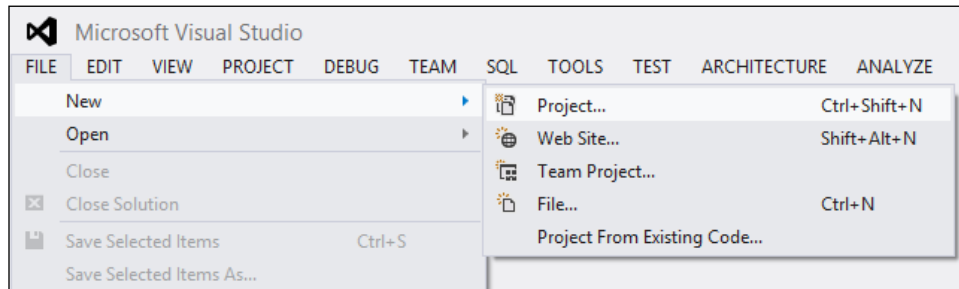
The remainder of the book will try to keep to the narrow path of developing iteratively on the same application; a Chat application with a web client, a .NET console client, and a Windows 8 WinRT client all working with the same server. We will also go into how to self-host SignalR in your own app for any clients to connect to, without having to have a web server installed and configured.

The persistent connection

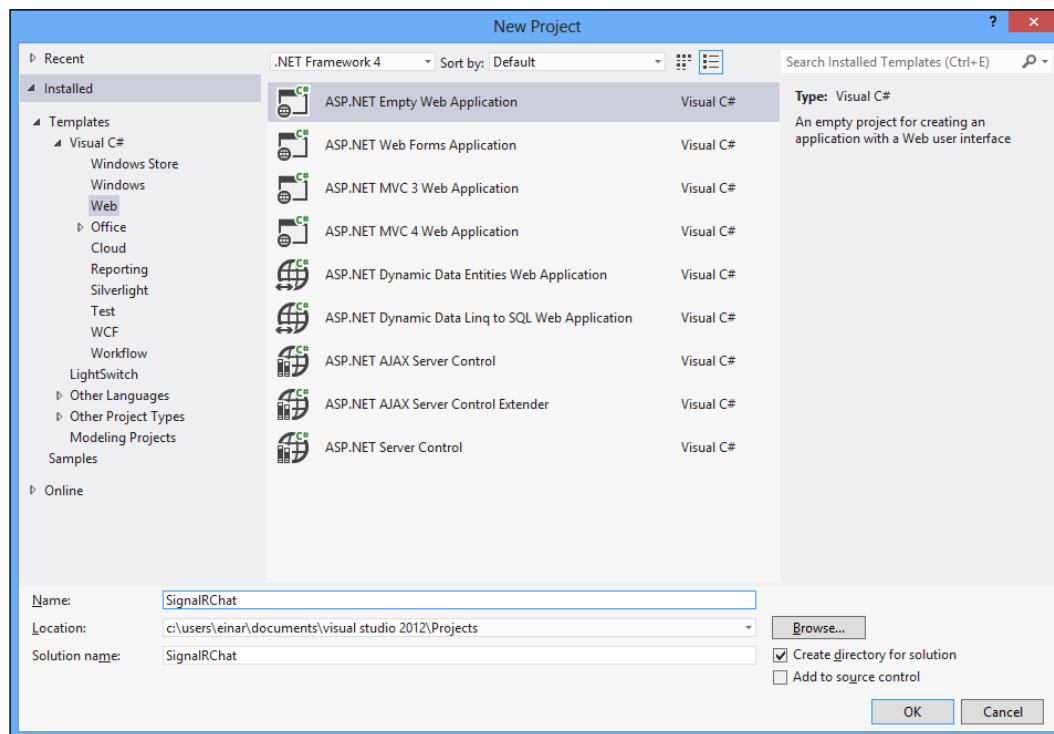
This chapter will start with showing the simplest construct that SignalR provides; `PersistentConnection`. A `PersistentConnection` provides a simple abstraction for sending data back and forth between a client and a server.

The process is the same for Visual Studio 2010 and 2012, but the samples in this book use Visual Studio 2012. You will however need to have NuGet installed. If you haven't, go to <http://www.nuget.org> and install it by clicking on the large **Install NuGet** button.

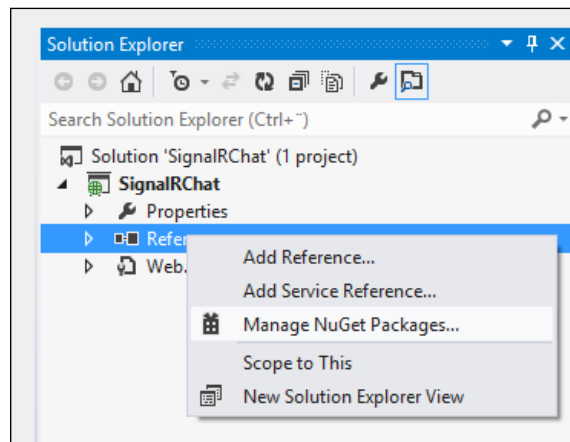
1. Let's start by creating a new project in Visual Studio.



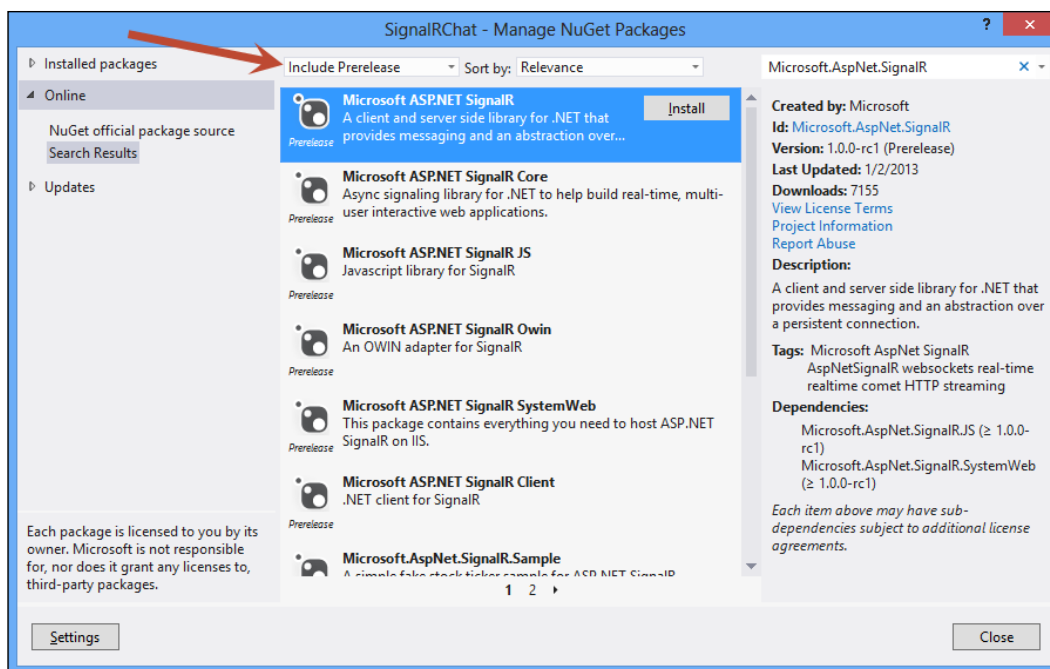
2. Select the regular **ASP.NET Empty Web Application** project template situated under **Visual C# | Web**. Give it a name, SignalRChat.



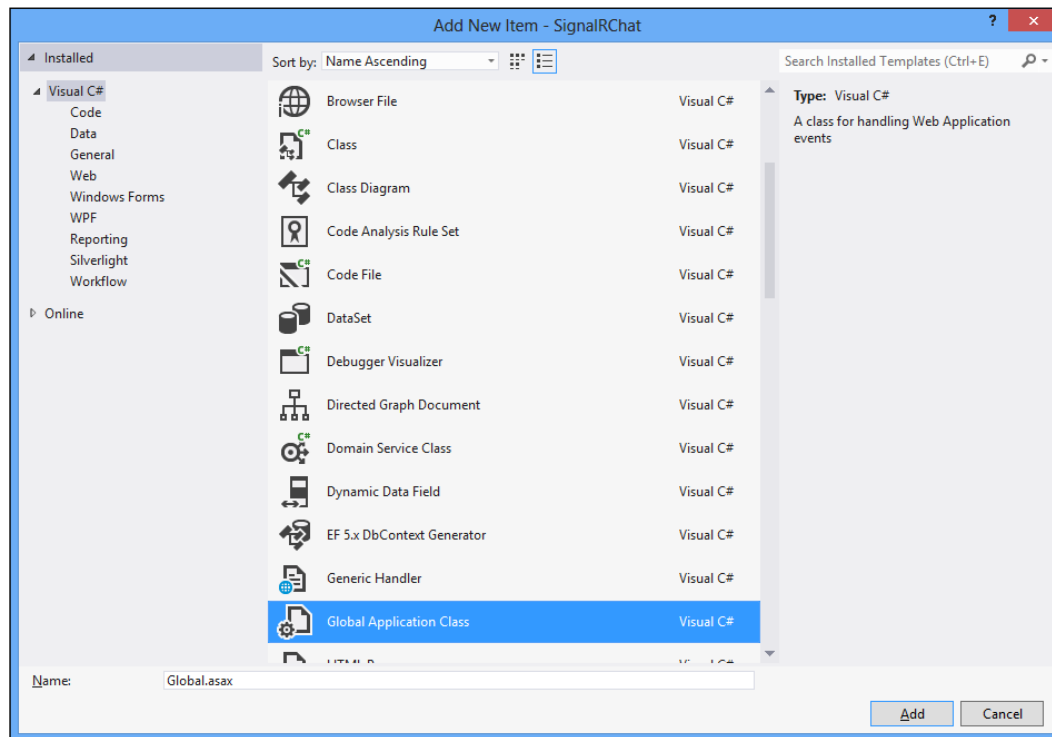
3. We're going to pull SignalR in from NuGet. Right-click on the **References** node in your **SignalRChat** project.



4. From the **Manage NuGet Packages** window, select **Online** in the tree to the left and type `Microsoft.AspNet.SignalR` in the search box in the upper-right corner of the window.

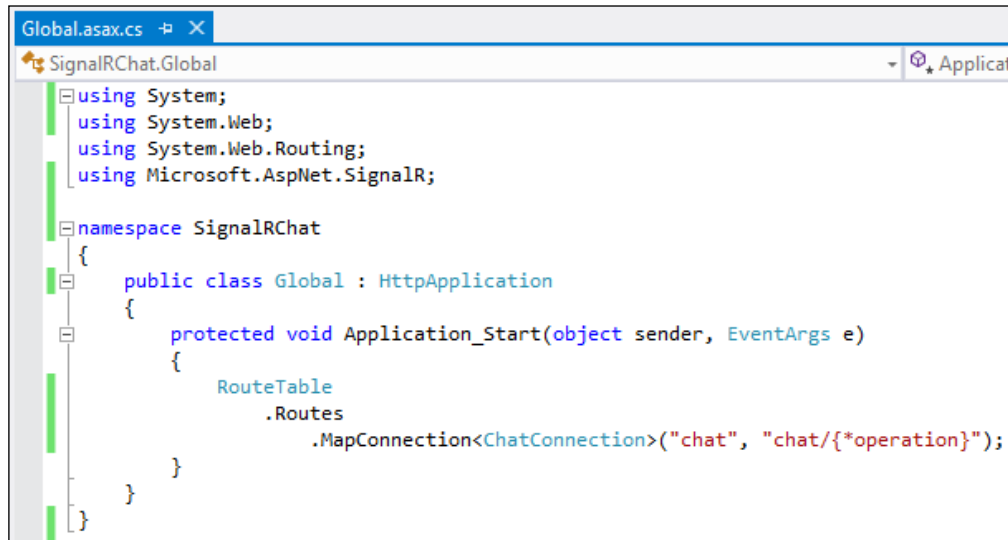


- Now that we have a basic web application, we will need to configure SignalR by registering a route in ASP.NET. In order to do that, we will need a Global Application Class (`Global.asax`) in the Web project. We add it by right-clicking on the project again and selecting **Add | New Item**. In the dialog, select **Global Application Class** found under the **Web** category.



- We won't be needing most of what comes out of the template, so we can strip it down to only have the `Application_Start` method.
- Also, we won't be needing all of the namespaces imported at the top, so strip it down to `System`, `System.Web`, `System.Web.Routing`, and `Microsoft.AspNet.SignalR`.

8. The only configuration we need at this stage is to add a route to the ASP.NET routing mechanism that associates `http://<your-site>:port/chat` with your `ChatConnection` class. Your file should look as follows:



```
Global.asax.cs
SignalRChat.Global
using System;
using System.Web;
using System.Web.Routing;
using Microsoft.AspNet.SignalR;

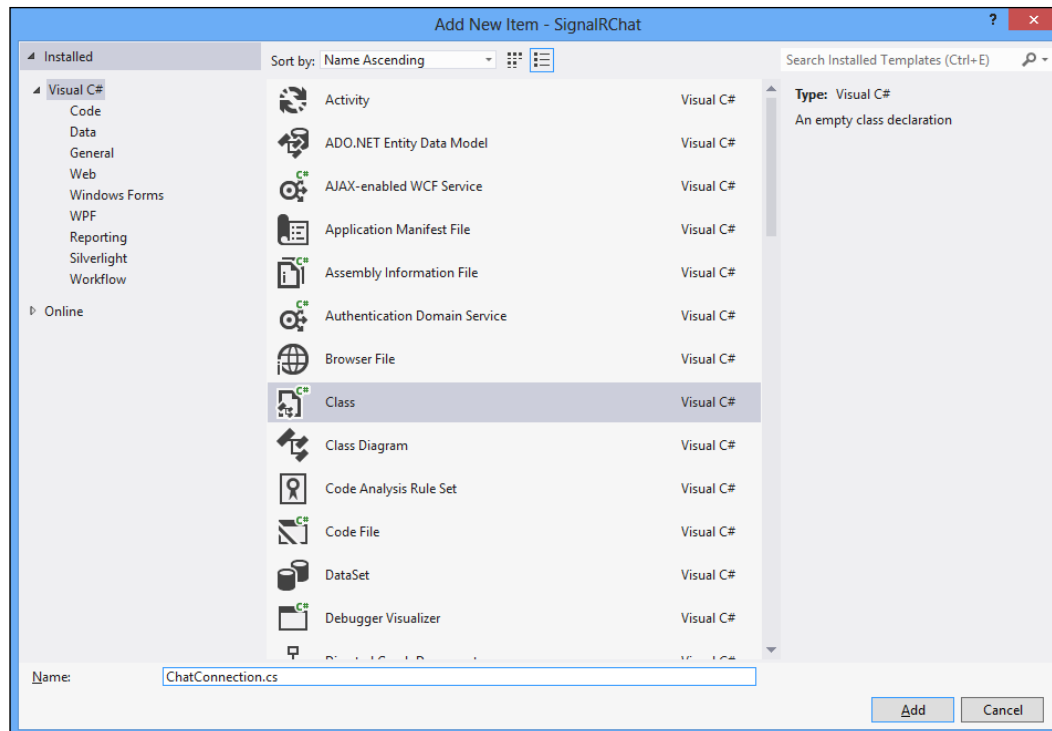
namespace SignalRChat
{
    public class Global : HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable
                .Routes
                .MapConnection<ChatConnection>("chat", "chat/{*operation}");
        }
    }
}
```

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

- Now we will create the `ChatConnection` class we mapped previously by adding a simple class, which will be our `PersistentConnection` for the chat. Right-click on the **SignalRChat** project and select **Add | New Item**. In the dialog, chose **Class** and give it the name `ChatConnection.cs`.

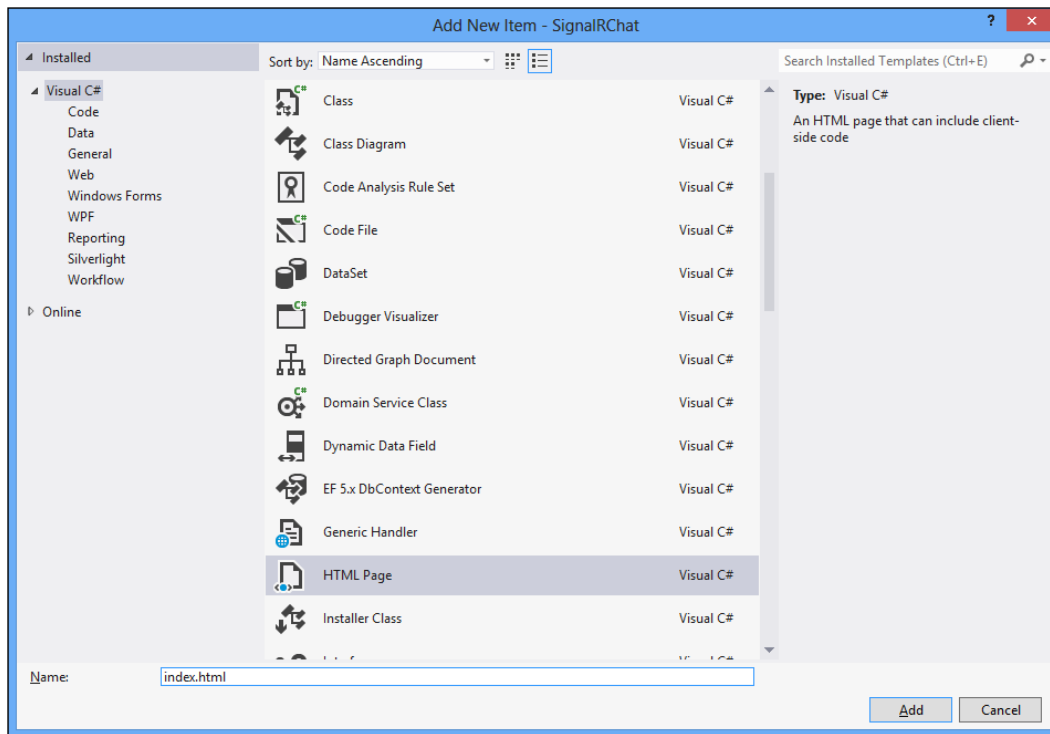


- The class represents the connection, but we don't need anything in it at this point, so we'll just leave it empty for now.

```
ChatConnection.cs
SignalRChat.ChatConnection
using Microsoft.AspNet.SignalR;

namespace SignalRChat
{
    public class ChatConnection : PersistentConnection
    {
    }
}
```

11. That's it for now for the server-side part of the solution. We'll need an HTML page and some JavaScript to connect to the server and get chatting. Add an HTML page to the Web project and name it `index.html`.




12. In the HTML file, we will need to add the necessary script imports. You can drag the file you want from the `scripts` folder onto the code editor at the right place within the `<head/>` element, or just type it out.
13. First we need to add jQuery, so any version of jQuery higher than 1.6 will do:

```
<script src="Scripts/jquery-1.8.3.js"></script>
```
14. Then we need to add the SignalR script that got added through NuGet:

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


15. Now we have enough to get started and we can add a new `<script/>` tag and connect to the server. The construct that follows `$(function() {})` is basically jQuery's way of hooking up the document-ready event, inside the braces we can then put the code that will run when the HTML document is ready. The following code connects to the server and when the connection is made, it writes to the JavaScript console that it is connected to:



```
index.html  X
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<script src="Scripts/jquery-1.8.3.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.js"></script>

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

        connection.start().done(function () {
            console.log("Connected");
        });
    });
</script>
</head>
<body>
</body>
</html>
```

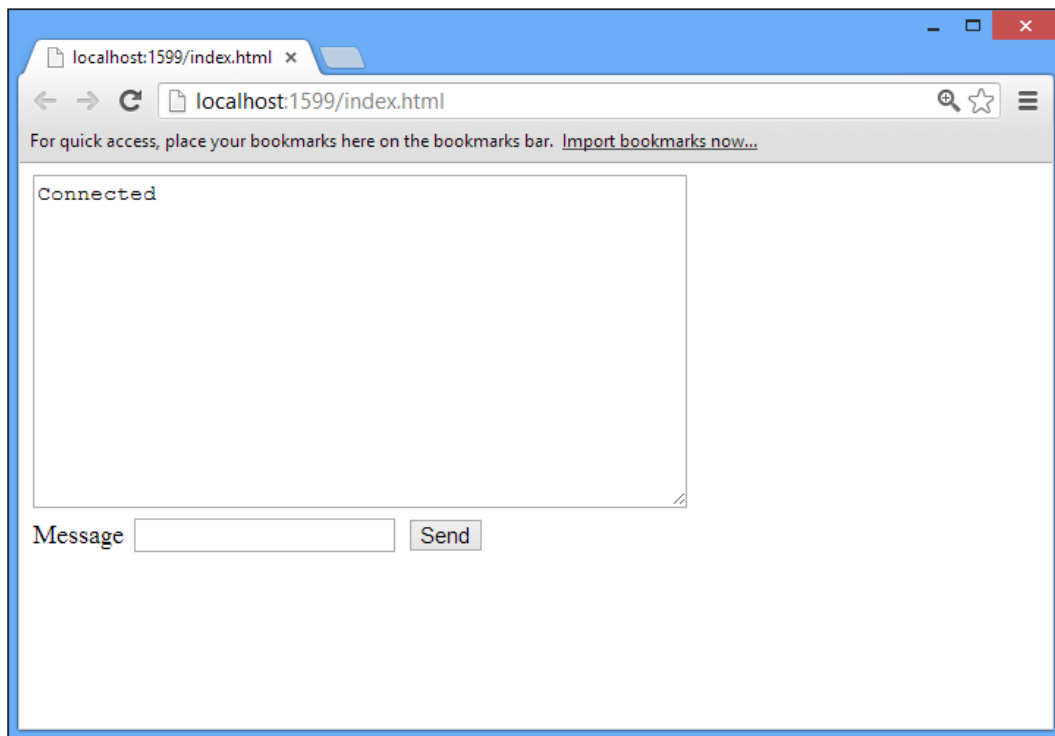
16. Obviously the JavaScript console won't do as a user interface, so let's add some simple HTML to represent our chat window, a textbox for typing in messages, and a button that the user can click when the message is ready to be sent.

```
<textarea id="chatWindow" style="width:400px; height:200px;"></textarea>
<div>
<label>Message</label>
<input id="messageTextBox" type="text" />
<button id="sendButton">Send</button>
</div>
```

17. We can now go back to the script we wrote and add some functionality. First we'll add a `Connected` string to the chat window when we are connected to the server. Then we hook up the **Send** button's click event so that we can send messages to the server from the textbox. When the message has been sent, we'll reset the textbox to be empty and ready for the next message.

```
connection.start().done(function () {  
    $("#chatWindow").val("Connected\n");  
    $("#sendButton").click(function () {  
        connection.send($("#messageTextBox").val());  
        $("#messageTextBox").val("")  
    });  
});
```

18. Building and running with *Ctrl + F5*, you should now see something as shown in the following screenshot:



19. Now we need to add some code to the `ChatConnection` class to deal with messages being received. Firstly, we need to add the `System.Threading.Tasks` statement. Then in our `ChatConnection` class, we override a method called `OnReceived`. This method is the one that gets called when data is received by any client. In this method we will take anything coming in and just broadcast it directly to all connected clients.

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

namespace SignalRChat
{
    public class ChatConnection : PersistentConnection
    {
        protected override Task OnReceivedAsync(IRequest request, string connectionId, string data)
        {
            return Connection.Broadcast(data);
        }
    }
}
```

20. On the client we can now hook up the received event on our connection. We'll take the data that we receive and just add it to the chat window.

```
connection.received(function (data) {
    $("#chatWindow").val($("#chatWindow").val() + data + "\n");
});
```

21. Your client JavaScript code should look like the following screenshot by now:

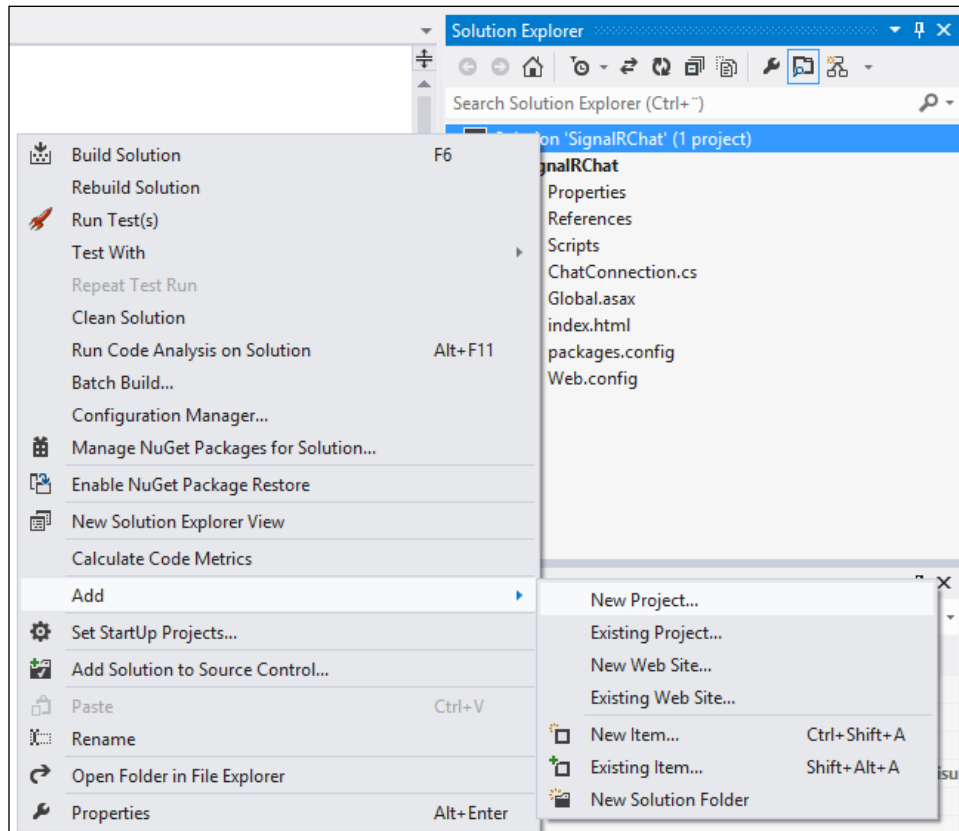
```
<script type="text/javascript">
    $(function () {
        var connection = $.connection("/chat");

        connection.start().done(function () {
            $("#chatWindow").val("Connected\n");
            $("#sendButton").click(function () {
                connection.send($("#messageTextBox").val());
                $("#messageTextBox").val("");
            });
        });

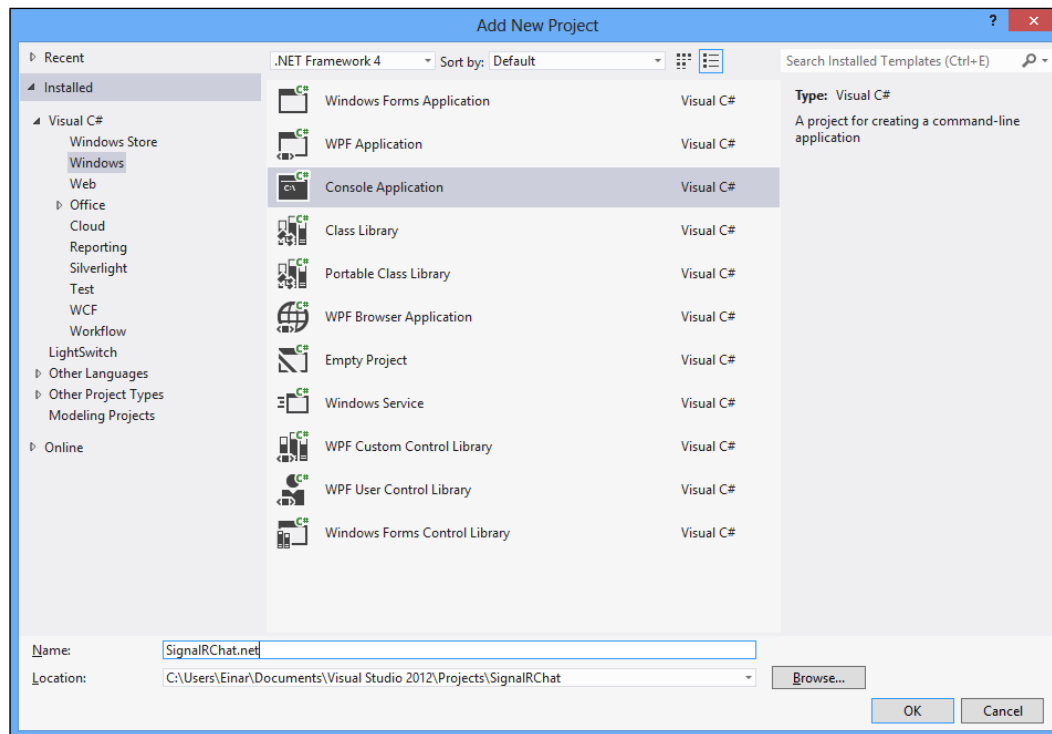
        connection.received(function (data) {
            $("#chatWindow").val($("#chatWindow").val() + data + "\n");
        });
    });
</script>
```

You should now be able to run the application and send messages from the client to the server. Opening up two browsers pointed at the same address should now also present any message you send to both browsers.

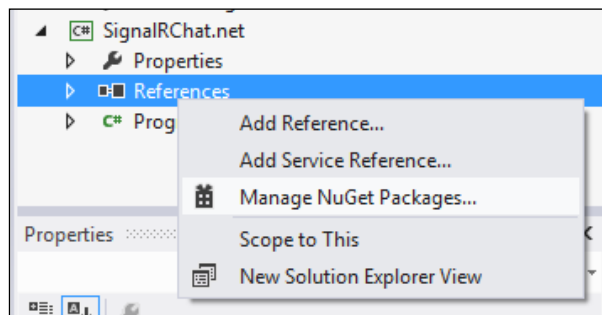
22. With the Web client up and running, we move straight into adding another client; a console .NET application. By right-clicking on your solution in Visual Studio, you can now select **Add | New Project**.



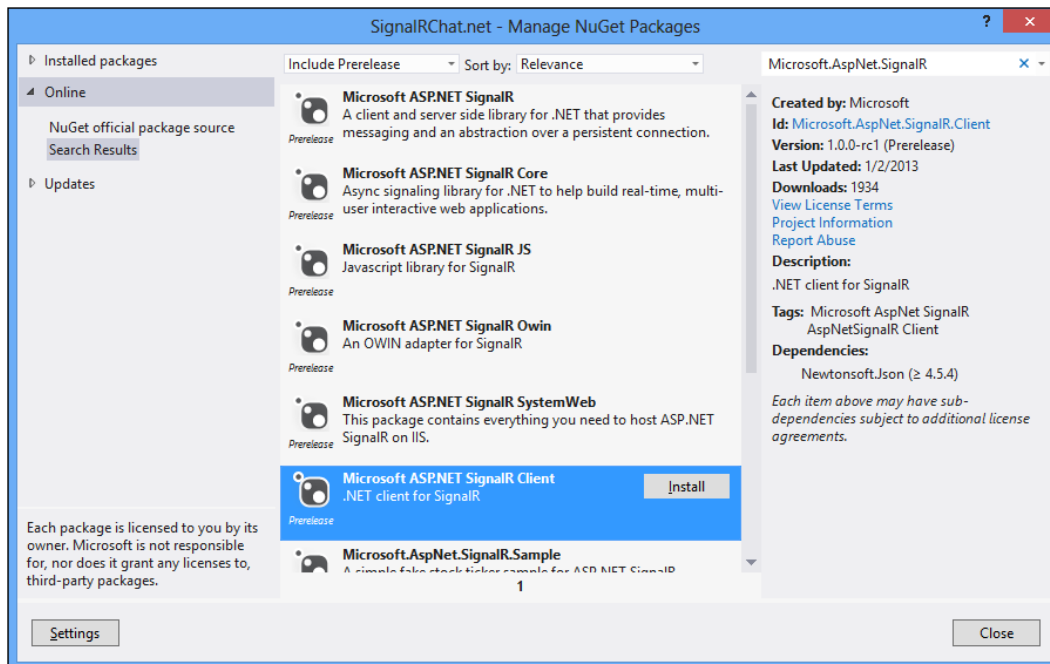
23. Select the **Console Application** template under the **Windows** category.
Give it the name `SignalRChat.net`.



24. Again we're going to use NuGet to get the necessary SignalR client library for working with a .NET console client.



25. Select **Online** and go the search field and type in `Microsoft.AspNet.SignalR`. This time we select the **Microsoft ASP.NET SignalR Client** package and install it.



26. Find the `Main()` method in the `Program.cs` file. This is where we will put the client logic we need for our chat at this stage.
27. Firstly, we'll need to add a namespace import at the top for SignalR using `Microsoft.AspNet.SignalR.Client`.
28. To connect, we will need a connection instance pointing back to the server. Make note of the URL, `http://localhost:1599/chat`, that you have in the browser for the HTML client. This is the URL we use for our connection. Inside the `Main()` method add the following code:

```
Var connection = new Connection
    (http://localhost:1599/chat);
```

29. We then go and hook up the received event, as with the JavaScript client and deal with the data received. Simply output it to the console:

```
Connection.Received += data => Console.WriteLine
    ("Received : "+data);
```

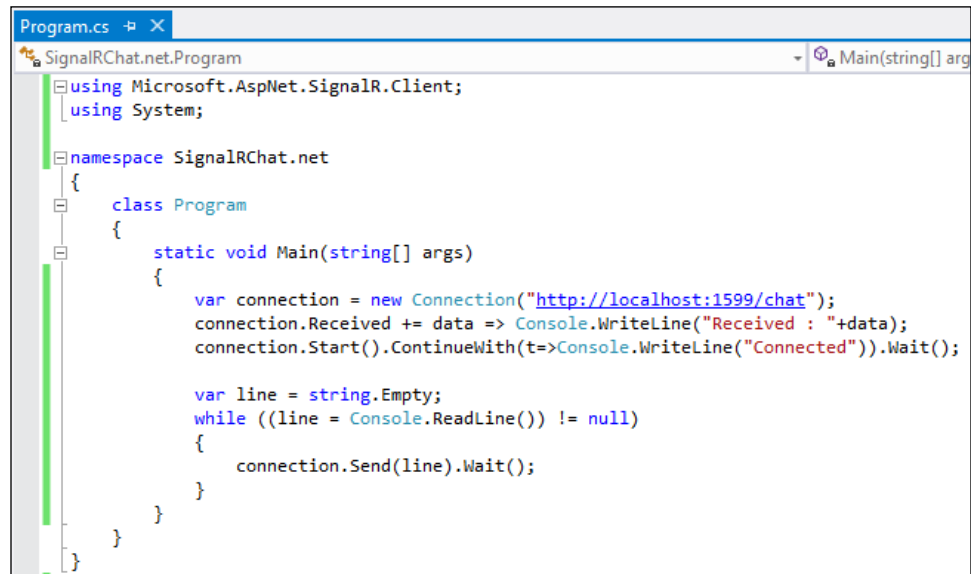
30. Now we want to start the connection and print out a statement when it is connected and wait until it's ready. We don't start sending anything before it is done.

```
Connection.Start().ContinueWith(t=>Console.WriteLine  
    ("Connected")).Wait();
```

31. Then we want to read lines from the console and send any input from the user to the connection.

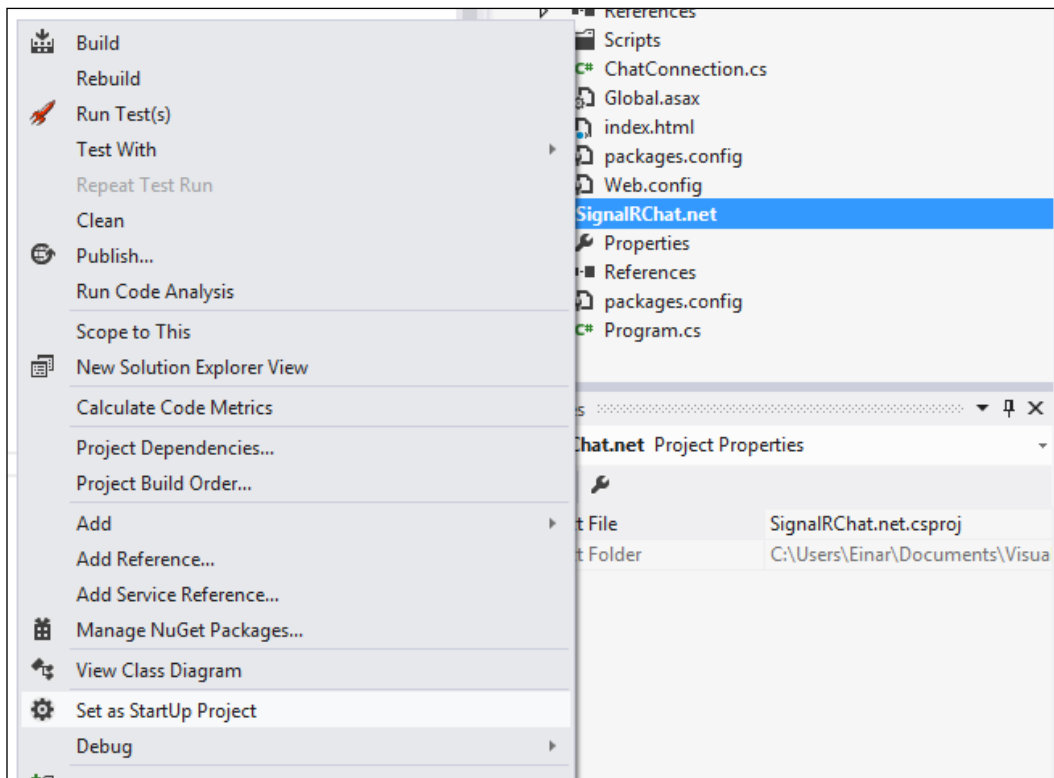
```
Var line = string.Empty;  
While((line = Console.ReadLine()) != null)  
{  
    connection.Send(line).Wait();  
}
```

32. Your Main() method should look as follows:

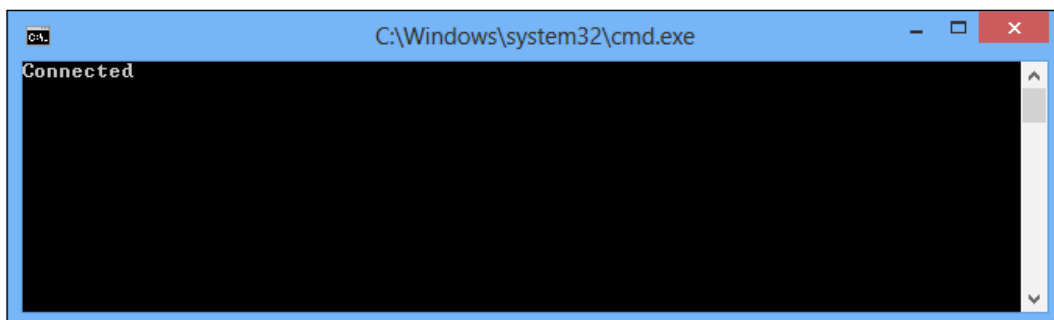


```
Program.cs [X]  
SignalRChat.net.Program  
using Microsoft.AspNet.SignalR.Client;  
using System;  
  
namespace SignalRChat.net  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            var connection = new Connection("http://localhost:1599/chat");  
            connection.Received += data => Console.WriteLine("Received : "+data);  
            connection.Start().ContinueWith(t=>Console.WriteLine("Connected")).Wait();  
  
            var line = string.Empty;  
            while ((line = Console.ReadLine()) != null)  
            {  
                connection.Send(line).Wait();  
            }  
        }  
    }  
}
```

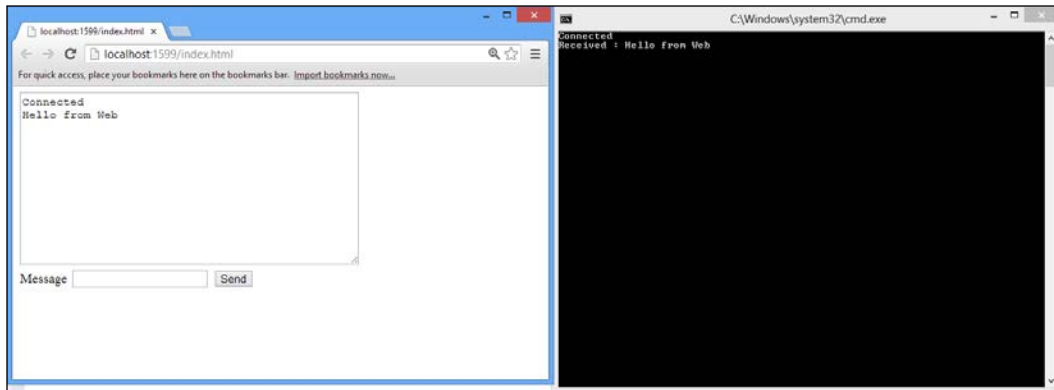
33. While your web app is running, you can select **SignalRChat.net** to be the startup project and start it.



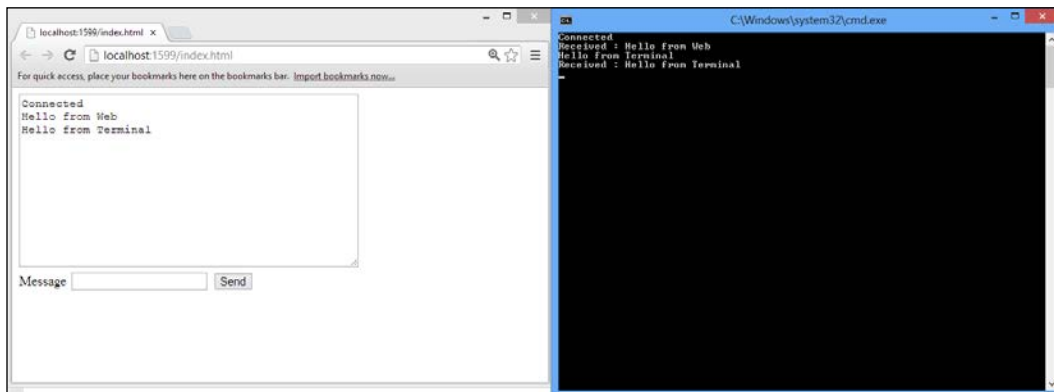
34. You should see a black console window with the Connected string.



35. You should also be able to send messages between the console and the Web. Anything written in the Web client should show up in the console and vice versa.



36. Typing in the console window and pressing *Enter* should now present the result in the Web client and also echo itself in the console window.



Summary

We've seen how easy it is to get started with SignalR and set up a persistent connection. Although there are differences between JavaScript and C#, it's still very easy to do. With the little effort put in this chapter, we are already sending a message for our chat application across two platforms. Moving forward from this point, we will be looking at how we connect and send messages with a different abstraction, Hubs in SignalR, providing a way to expose functionality on the server more naturally.

3

Hubs

This chapter will cover how you connect a client with a server in a very different way, making it seem like you can call code directly on the client from the server and vice versa.

Topics covered in this chapter are as follows:

- Setting up a Hub on the server
- Working of the Hubs
- Consuming a Hub from a .net client
- Consuming a Hub from a JavaScript client

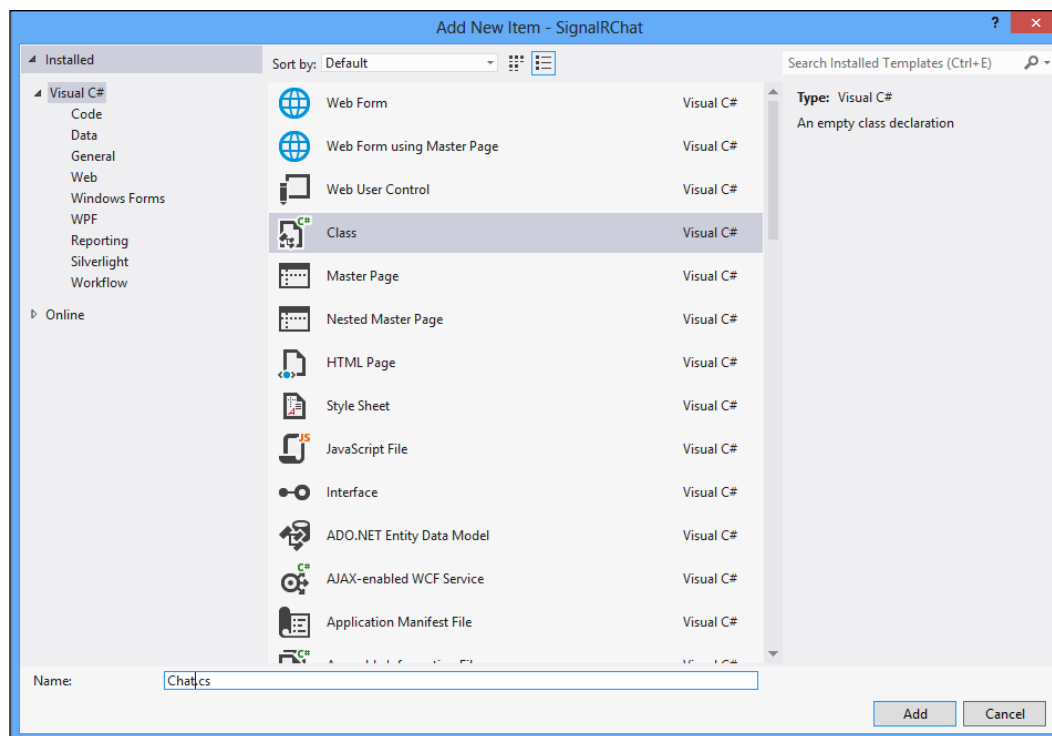
At this stage, the developer should be able to use Hubs, and our sample app will now be able to chat properly.

Moving up one level

While `PersistentConnection` seems very easy to work with, it is the lowest level in SignalR. It does provide the perfect abstraction for keeping a connection open between a client and a server, but that's just about all it does provide. Working with different operations is not far from how you would deal with things in a regular socket connection, where you basically have to parse whatever is coming from a client and figure out what operation is being asked to be performed based on the input. SignalR provides a higher level of abstraction that removes this need and you can write your server-side code in a more intuitive manner. In SignalR, this higher level of abstraction is called a Hub. Basically, a Hub represents an abstraction that allows you to write classes with methods that take different parameters, as you would with any API in your application, and then makes it completely transparent on the client—at least for JavaScript. This resembles a concept called **Remote Procedure Call (RPC)**, with many incarnations of it out there.

For our chat application at this stage, we basically just want to be able to send a message from a client to the server and have it send the message to all of the other clients connected. To do this, we will now move away from the `PersistentConnection` and introduce a new class called `Hub` using the following steps:

1. First, start off by deleting the `ChatConnection` class from your Web project.
2. Now we want to add a Hub implementation instead. Right-click on the **SignalRChat** project and select **Add | New Item**.
3. In the dialog, chose **Class** and give it a name `Chat.cs`.



4. This is the class that will represent our Hub. Make it inherit from Hub:

```
Public class Chat : Hub
```

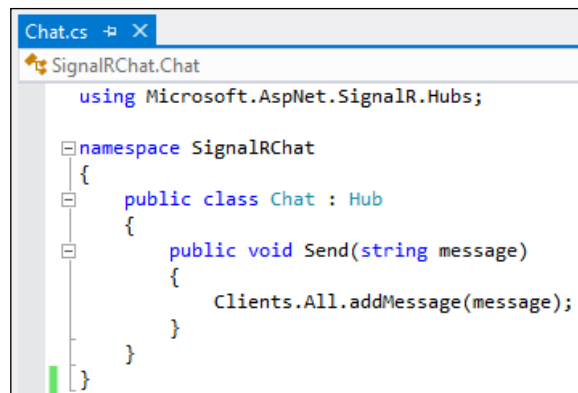
5. Add the necessary import statement at the top of the file:

```
using Microsoft.AspNet.SignalR.Hubs;
```

6. In the class we will add a simple method that the clients will call to send a message. We call the method `Send` and take one parameter into it; a string which contains the message being sent by the client:

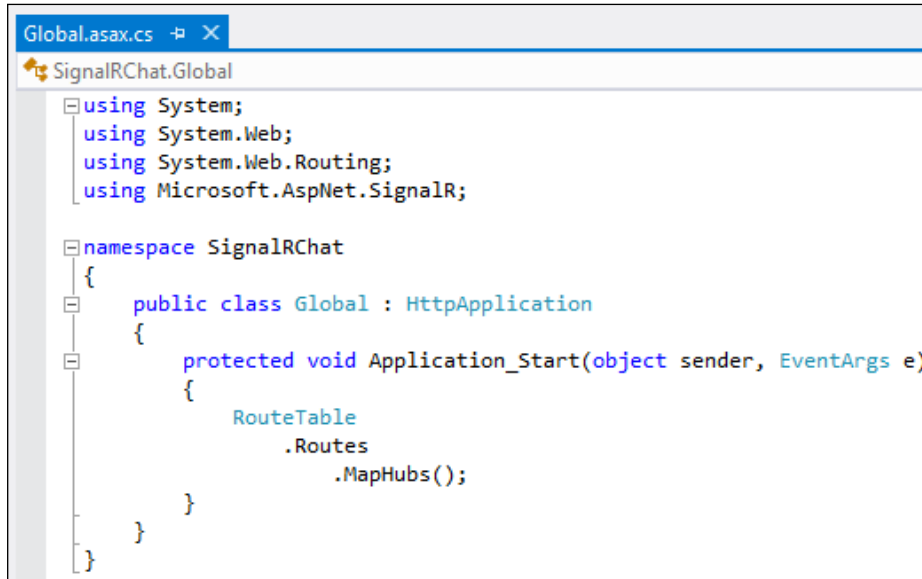
```
Public void Send(string message)
{
}
```

7. From the base class of `Hub`, we get a few things that we can use. For now we'll be using the `Clients` property to broadcast to all other clients connected to the `Hub`. On the `Clients` property, you'll find an `All` property which is dynamic; on this we can call anything and the client will just have to subscribe to the method we call, if the client is interested.



It is possible to change the name of the `Hub` to not be the same as the class name. An attribute called `HubName()` can be placed in front of the class to give it a new name. The attribute takes one parameter; the name you want for your `Hub`. Similarly, for methods inside your `Hub`, you can use an attribute called `HubMethodName()` to give the method a different name.

8. The next thing we need to do is to go into the `Global.asax.cs` file, and make some changes. Firstly, we remove the `.MapConnection(...)` line and replace it with a `.MapHubs()` line. This will make all Hubs in your application automatically accessible from a default URL. All Hubs in the application will be mapped to `/signalr/<name of hub>`; so more concretely the path will be: `http://<your-site>:port/signalr/<name of hub>`. We're going with the defaults for now. It should cover the needs on the server-side code.



```
Global.asax.cs -> X
SignalRChat.Global

using System;
using System.Web;
using System.Web.Routing;
using Microsoft.AspNet.SignalR;

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

9. Moving into the JavaScript/HTML part of things, SignalR comes with a JavaScript proxy generator that can generate JavaScript proxies from your Hubs mapped using `.MapHubs()`. This is also subject to the same default URL but will follow the configuration given to `.MapHubs()`. We will need to include a script reference in the HTML code right after the line that references the SignalR JavaScript file. We add the following:

```
<script src="/signalr/hubs" type="text/javascript"></script>
```

This will include the generated proxies for our JavaScript client. What this means is that we get whatever is exposed on a Hub generated for us and we can start using it straight away.

10. Before we get started with the concrete implementation for our web client, we can move all of the custom code that we implemented in *Chapter 1, Revitalizing the Rich Client*, for `PersistentConnection` altogether.

11. We then want to get to our proxy, and work with it. It sits on the connection object that SignalR adds to jQuery. So, for us, that means an object called `chat` will be there.

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

12. On the `chat` object, sit two important properties, one representing the client functions that get invoked when the server "calls" something on the client. And the second one is the property representing the server and all of the functionalities that we can call from the client. Let's start by hooking up the client and its methods. Earlier we implemented in the Hub sitting on the server a call to `addMessage()` with the message. This can be added to the client property inside the `chat` Hub instance:

```
chat.client.addMessage = function (message) {
    $("#chatWindow").val($("#chatWindow").val() + message + "\n");
}
```

13. Basically, whenever the server calls that method, our client counterpart will be called. Now what we need to do is to start the Hub and print out when we are connected to the chat window:

```
$.connection.hub.start().done(function() {
    $("#chatWindow").val("Connected\n");
});
```

14. Then we need to hook up the `click` event on the button and call the server to send messages. Again, we use the server property sitting on the chat hub instance in the client, which corresponds to a method on the Hub:

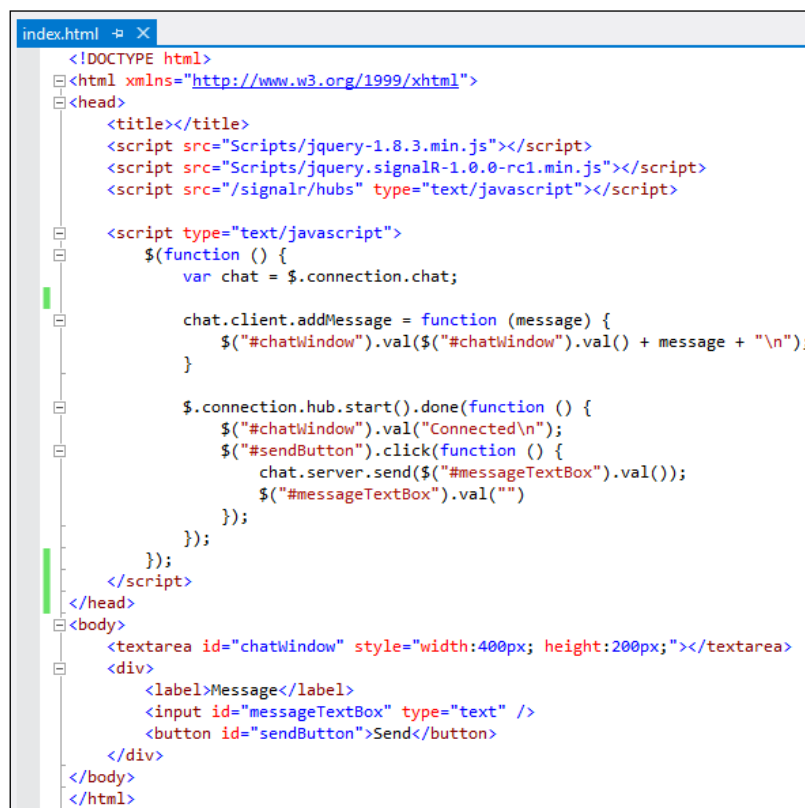
```
$("#sendButton").click(function() {
    chat.server.send($("#messageTextBox").val());
    $("#messageTextBox").val("");
});
```

15. You should now have something that looks as follows:

```
$.connection.hub.start().done(function () {
    $("#chatWindow").val("Connected\n");
    $("#sendButton").click(function () {
        chat.server.send($("#messageTextBox").val());
        $("#messageTextBox").val("")
    });
});
```

You may have noticed that the send function on the client is in camelCase and the server-side C# code has it in PascalCase. SignalR automatically translates between the two case types. In general, camelCase is the preferred and the most broadly used casing style in JavaScript – while Pascal being the most used in C#.

16. You should now be having a full sample in HTML/JavaScript that looks like the following screenshot:



17. Running it should produce the same result as before, with the exception of the .NET terminal client, which also needs alterations. In fact, let's just get rid of the code inside `Program.cs` and start over. The client API is a bit rougher in C#; this comes from the more statically typed nature of C#. Sure, it is possible – technically – to get pretty close to what has been done in JavaScript, but it hasn't been a focal point for the SignalR team.
18. Basically, we need a different connection than the `PersistentConnection` class. We'll be needing a `HubConnection` class. From the `HubConnection` class we can create a proxy for the chat Hub:

```
var hubConnection = new HubConnection("http://localhost:1599");  
var chat = hubConnection.CreateHubProxy("chat");
```

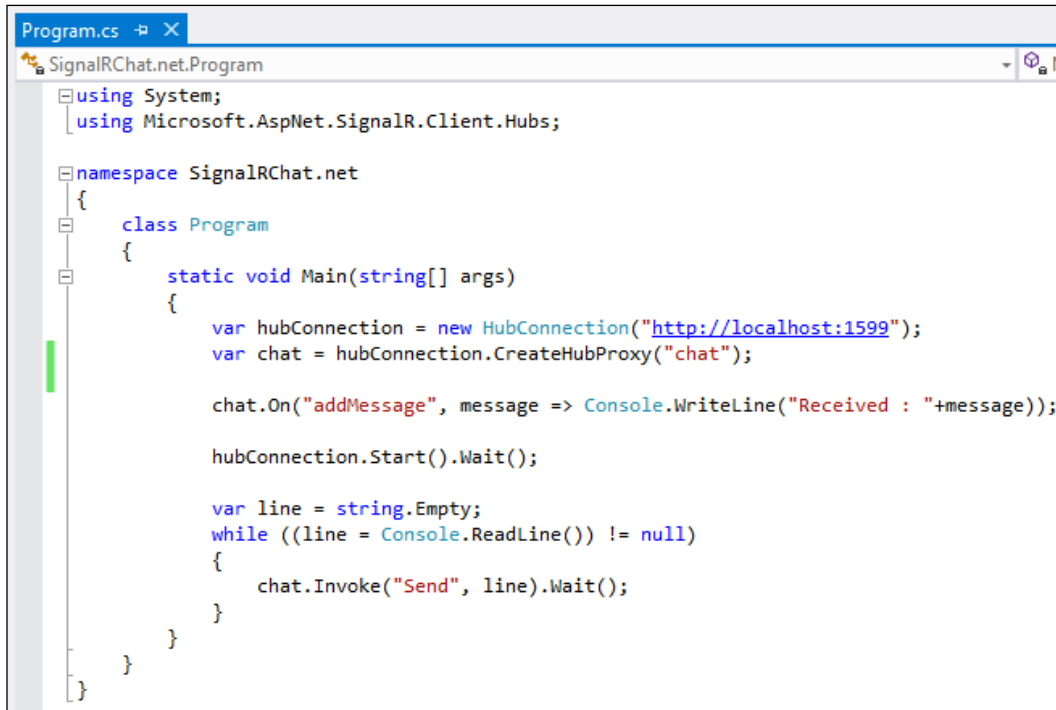
19. As with JavaScript, we can hook up client-side methods that get invoked when the server calls any client. Although as mentioned, not as elegantly as in JavaScript. On the chat Hub instance, we get a method called `On()`, which can be used to specify a client-side method corresponding to the client call from the server. So we set `addMessage` to point to a method which, in our case, is for now just an inline lambda expression.

```
chat.On("addMessage", message => Console.WriteLine("Received : "+message));
```

20. Now we need, as with `PersistentConnection`, to start the connection and wait until it's connected:
`hubConnection.Start().Wait();`
21. Now we can get user input and send it off to the server. Again, as with client methods called from the server, we have a slightly different approach than with JavaScript; we call the `Invoke` method giving it the name of the method to call on the server and any arguments. The `Invoke()` method does take a parameter, so you can specify any number of arguments which will then be sent to the server:

```
var line = string.Empty;  
while ((line = Console.ReadLine()) != null)  
{  
    chat.Invoke("Send", line).Wait();  
}
```


22. The finished result should look something like the following screenshot, and now work in full correspondence with the JavaScript chat:



```
Program.cs
SignalRChat.net.Program

using System;
using Microsoft.AspNet.SignalR.Client.Hubs;

namespace SignalRChat.net
{
    class Program
    {
        static void Main(string[] args)
        {
            var hubConnection = new HubConnection("http://localhost:1599");
            var chat = hubConnection.CreateHubProxy("chat");

            chat.On("addMessage", message => Console.WriteLine("Received : "+message));

            hubConnection.Start().Wait();

            var line = string.Empty;
            while ((line = Console.ReadLine()) != null)
            {
                chat.Invoke("Send", line).Wait();
            }
        }
    }
}
```

Summary

Exposing our functionality through Hubs makes it easier to consume on the client, at least on JavaScript based clients, due to the proxy generation. It basically brings it to the client as if it was on the client. With the Hub you also get the ability to call the client from the server in a more natural manner. One of the things often important for applications is the ability to filter out messages so you only get messages relevant for your context. Groups in the next chapter will cover this, groups is the technique used in SignalR to accomplish this.

4 Groups

This chapter will cover how you can group connections together and target specific groups when sending messages. The topics covered include:

- Establishing groups on the server
- Sending messages from the client to specific groups

At this stage, the developer should be able to create groups and put connections into these groups.

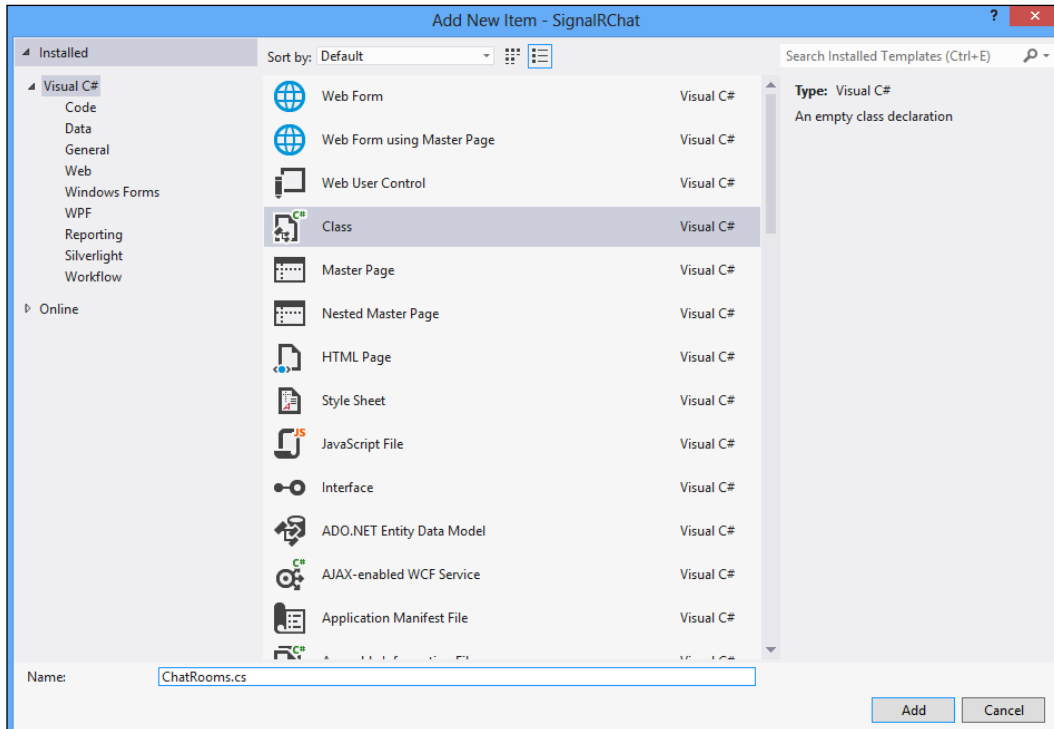
Getting specific with groups

Many scenarios require you to not broadcast to everyone, and be more specific in who receives the message. SignalR provides an abstraction for these called **groups**. Groups hold connections that will receive messages. The API for sending messages is just the same, but you just choose to go to a specific group instead of going to everyone.

For our chat applications we want to use groups to create chat rooms, specific groups which will receive chat messages that none of the other rooms/groups will receive. In order to get this working we will need to change our application slightly:

1. First we will need a class to manage the chat rooms that get created by any clients so that we keep track of them for any other clients that connect.

2. Right-click on the project and navigate to **Add | New Item**. In the dialog box, choose **Class** and give it the name `ChatRooms.cs`.



3. Basically, this class will just hold a list of strings that represent the different chat rooms by name for now:

```
Static List<string> _rooms = new List<string>();
```
4. We then want to just add the ability to add a chat room and get all the available chat rooms, and also check if a room already exists. By default, we will also add a well known chat room called `Lobby`. The class should look as follows:

```
using System.Collections.Generic;

namespace SignalRChat
{
    public class ChatRooms
    {
        static List<string> _rooms = new List<string>();

        static ChatRooms()
        {
            _rooms.Add("Lobby");
        }

        public static void Add(string name)
        {
            _rooms.Add(name);
        }

        public static bool Exists(string name)
        {
            return _rooms.Contains(name);
        }

        public static IEnumerable<string> GetAll()
        {
            return _rooms;
        }
    }
}
```

The line at the top of the class, `Static List<string> _rooms = new List<string>()`; is using an the built-in `List<string>`, which is not thread-safe. This means that you can run into exceptions if you have two users joining a chat room at the same time. There are implementations out there that deal with this, and a quick search on the phrase `ConcurrentList C#` will yield a few options. Also, there is a `SynchronizedCollection<>` type found in `System.Collections.Generic` after adding a reference to `System.ServiceModel` that can be used.

5. Now, let's go back to the `HubChat.cs` that we created in *Chapter 3, Hub*. We will need to add some new functionality to this class, and also change some existing functionalities to make it support chat rooms. Let's start by adding a method that we can call from the client to create a chat room:

```
public void CreateChatRoom(string room)
{
    if (!ChatRooms.Exists(room))
    {
        ChatRooms.Add(room);
        Clients.All.addChatRoom(room);
    }
}
```

6. Next, we want to add support for clients to be able to join a room and receive any messages published to it. This is where we use the `Groups` property from the Hub and just add the connection of the client connected to the group with the same name as the room we want to join:

```
public void Join(string room)
{
    Groups.Add(Context.ConnectionId, room);
}
```

With that the client will only receive messages for the groups it is a part of and none of the others.

7. Now that we have a subsystem for dealing with rooms, and clients can connect to those rooms, we want to be able to tell the clients that are getting connected which rooms are available. On the Hub there is a virtual method, `OnConnected()`, which we can override for getting notified when a client connects. There are also methods to know when clients disconnect and reconnect. But for now we only need the `OnConnected()` method. In this method, we basically get all the rooms that have been created and send them off to the client that got connected using the `Caller` property on the `Clients` property which is sitting on the Hub:

```
public override Task OnConnected()
{
    foreach (var room in ChatRooms.GetAll())
        Clients.Caller.addChatRoom(room);

    return base.OnConnected();
}
```

Your Hub should now look as follows:

```
public class Chat : Hub
{
    public void Join(string room)
    {
        Groups.Add(Context.ConnectionId, room);
    }

    public void CreateChatRoom(string room)
    {
        if (!ChatRooms.Exists(room))
        {
            ChatRooms.Add(room);
            Clients.All.addChatRoom(room);
        }
    }

    public void Send(string room, string message)
    {
        Clients.Group(room).addMessage(room, message);
    }

    public override Task OnConnected()
    {
        foreach (var room in ChatRooms.GetAll())
            Clients.Caller.addChatRoom(room);

        return base.OnConnected();
    }
}
```

8. Moving to the `index.html` file of your web application, we want to make some changes and add some new features. Let's start by altering the look of our app and introduce a list of available chat rooms as well as a way of adding a chat room. We will wrap it all up nicely in a table:

```
<table>
  <tr style="vertical-align:top;">
    <td>
      <select id="chatRoomsList" size="2" style="width:150px; height:206px;">
      </select>
    </td>
    <td>
      <textarea id="chatWindow" style="width:400px; height:200px;"></textarea>
      <div>
        <label>Message</label>
        <input id="messageTextBox" type="text" />
        <button id="sendButton">Send</button>
      </div>
      <div>
        <label>Start new chat room</label>
        <input id="chatRoomTextBox" type="text" />
        <button id="createChatRoomButton">Create</button>
      </div>
    </td>
  </tr>
</table>
```

9. We will then continue by introducing a variant that holds the current chat room; we will put it the beginning of our script block and initialize it with Lobby as our default chat room:

```
var currentChatRoom = "Lobby";
```

10. Now we want to take the code that adds a message to the chatWindow textbox and put it into a function we can reuse:

```
function addMessageToChatWindow(message) {  
    $("#chatWindow").val($("#chatWindow").val() + message + "\n");  
}
```

11. Then we will need a function that can take a string with the chat room name and add it to the <select/> list:

```
function addChatRoomToList(chatRoom) {  
    $("#chatRoomsList").append("<option value="+chatRoom+">" + chatRoom + "</option>");  
}
```

12. Then we move onto adding the client counterpart of the addChatRoom() function that we just inserted into the server code:

```
chat.client.addChatRoom = function (chatRoom) {  
    addChatRoomToList(chatRoom);  
    if (chatRoom == currentChatRoom) {  
        $("#chatRoomsList").val(currentChatRoom);  
        addMessageToChatWindow("Welcome to " + currentChatRoom);  
    }  
}
```

13. Now, since we changed how the server calls its clients when a message is received, we need to change the client. We add an argument at the beginning that holds the room it belongs to, and we also filter it out. The reason for filtering is that we will only look at one room at a time.

```
chat.client.addMessage = function (room, message) {  
    if (room === currentChatRoom) {  
        addMessageToChatWindow(message);  
    }  
}
```

14. Let's hook up the button for creating a chat room and get the content of the textbox and send it to the server:

```
$("#createChatRoomButton").click(function () {  
    chat.server.createChatRoom($("#chatRoomTextBox").val());  
    $("#chatRoomTextBox").val("")  
});
```

15. Whenever a chat room is selected from the list, we want to join that chat room and set the current chat room on the client. In addition, we want to clear the chat window since we're no longer in that room:

```
$("#chatRoomsList").change(function () {  
    currentChatRoom = $("#chatRoomsList option:selected").val();  
    chat.server.join(currentChatRoom);  
    clearChatRoomForRoomChange(currentChatRoom);  
});
```

16. The logic for sending to the server needs a slight change; it needs to have the current chat room as the first argument to match the change on the server, and the second argument will be the message:

```
$("#sendButton").click(function () {  
    chat.server.send(currentChatRoom,$("#messageTextBox").val());  
    $("#messageTextBox").val("")  
});
```


Your whole script should look as follows:

```
<script type="text/javascript">
    var currentChatRoom = "Lobby";

    function addChatRoomToList(chatRoom) {
        $("#chatRoomsList").append("<option value="+chatRoom+">" + chatRoom + "</option>");
    }

    function addMessageToChatWindow(message) {
        $("#chatWindow").val($("#chatWindow").val() + message + "\n");
    }

    function clearChatRoomForRoomChange(chatRoom) {
        $("#chatWindow").val("");
        addMessageToChatWindow("Welcome to " + chatRoom);
    }

    $(function () {
        var chat = $.connection.chat;

        chat.client.addChatRoom = function (chatRoom) {
            addChatRoomToList(chatRoom);
            if (chatRoom == currentChatRoom) {
                $("#chatRoomsList").val(currentChatRoom);
                addMessageToChatWindow("Welcome to " + currentChatRoom);
            }
        }

        chat.client.addMessage = function (room, message) {
            if (room === currentChatRoom) {
                addMessageToChatWindow(message);
            }
        }

        $.connection.hub.start().done(function () {
            $("#chatWindow").val("Connected\n");
            chat.server.join("Lobby");
            $("#sendButton").click(function () {
                chat.server.send(currentChatRoom,$("#messageTextBox").val());
                $("#messageTextBox").val("");
            });

            $("#createChatRoomButton").click(function () {
                chat.server.createChatRoom($("#chatRoomTextBox").val());
                $("#chatRoomTextBox").val("");
            });

            $("#chatRoomsList").change(function () {
                currentChatRoom = $("#chatRoomsList option:selected").val();
                chat.server.join(currentChatRoom);
                clearChatRoomForRoomChange(currentChatRoom);
            });
        });
    });
</script>
```

17. Similar changes need to be made for our console application. We start by establishing a variable to hold the current chat room:

```
var currentRoom = "Lobby";
```

18. Then we need to change the way we deal with `addMessage` using the new room argument, and also do similar filtering as we did in JavaScript; don't print out anything from rooms we aren't in:

```
chat.On("addMessage", (string room, string message) => {  
    if( room == currentRoom )  
        Console.WriteLine("Received : " + message);  
});
```

19. Then we hook up the `addChatRoom` event and print out any rooms that are added:

```
chat.On("addChatRoom", room => Console.WriteLine("Room added : "+room));
```

20. After we have connected to the Hub we want to join the Lobby room, so we add the following line of code:

```
chat.Invoke("Join", currentRoom).Wait();
```

21. Inside our `while` loop where we wait for input from the user, we want to add some new features. We want the user to be able to create chat rooms, so we add the ability for the user to do by creating `<room name>` in the console.

```
if (line.StartsWith("/create"))  
{  
    var room = line.Substring("/create".Length + 1);  
    chat.Invoke("CreateChatRoom", room);  
}
```

22. Then we want to add the ability to join an existing room:

```
if (line.StartsWith("/join"))
{
    var room = line.Substring("/join".Length + 1);
    chat.Invoke("Join", room).Wait();
    currentRoom = room;
}
```

23. Any other input should just be sent to the server, but now with the room as the first argument:

```
else
{
    chat.Invoke("Send", currentRoom, line).Wait();
}
```

The whole application code should look as follows:

```
static void Main(string[] args)
{
    var hubConnection = new HubConnection("http://localhost:1599");
    var chat = hubConnection.CreateHubProxy("chat");

    var currentRoom = "Lobby";

    chat.On("addMessage", (string room, string message) => {
        if (room == currentRoom)
            Console.WriteLine("Received : " + message);
    });

    chat.On("addChatRoom", room => Console.WriteLine("Room added : "+room));

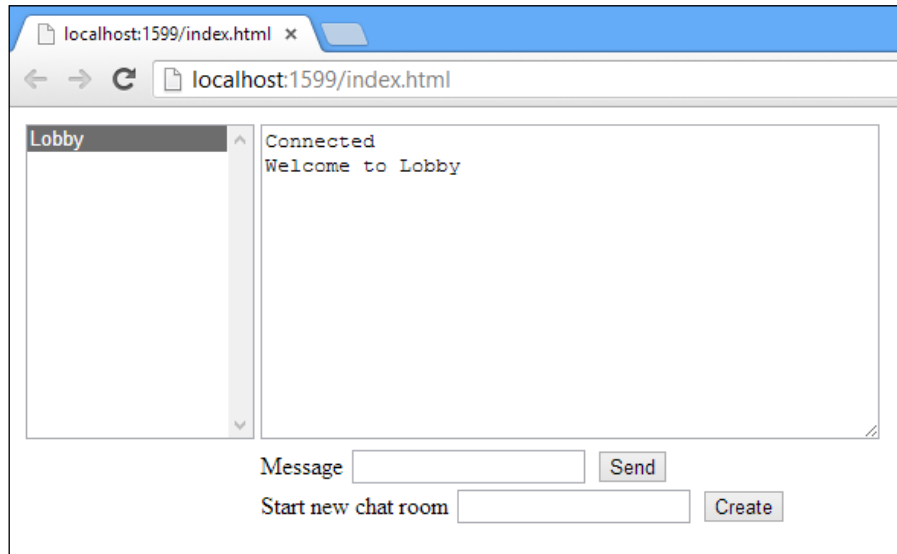
    hubConnection.Start().Wait();

    Console.WriteLine("Connected");

    chat.Invoke("Join", currentRoom).Wait();

    var line = string.Empty;
    while ((line = Console.ReadLine()) != null)
    {
        if (line.StartsWith("/create"))
        {
            var room = line.Substring("/create".Length + 1);
            chat.Invoke("CreateChatRoom", room);
        }
        if (line.StartsWith("/join"))
        {
            var room = line.Substring("/join".Length + 1);
            chat.Invoke("Join", room).Wait();
            currentRoom = room;
        }
        else
        {
            chat.Invoke("Send", currentRoom, line).Wait();
        }
    }
}
```

The Web app should look as follows:



Summary

Often you find yourself not wanting to broadcast your messages to all the clients; groups in SignalR gives you the control you want for grouping connections together and only sending messages to the group(s) you want. You should now be able to apply grouping as a technique for filtering.

5

State

This chapter will cover how you can have a state in the client that is available automatically on the server. At this stage, the developer should be able to have a state on the client that is also on the server.

Becoming stateful

SignalR brings a lot to the table for abstractions; we've seen so far the abstractions of persistent connections and Hubs. Diving further into Hubs you'll see that there is even more. One of the cool things one can add is round-trip state. This is the state that is carried alongside any traffic going from a client to the server. So whenever you call something from the client to the server or the server calls something on the client, any state you have specified will be transferred along with the call being made. This is something that will be really convenient, especially when you have information that is not necessarily relevant for the method you need to call, but something that you might want to have while cross cutting any method you call.

For now, we're just going to use it for something very simple – to keep track of the current chat room for the client. This way we don't need to have it on the method call for sending a message, it will just be available on the server as well as on the client. Perform the following steps to become stateful:

1. Let's start by changing the server a bit. Open the server-side representation of the `Hub—Chat.cs`.
2. Change the method signature by removing the first parameter for room that we had put in in the previous chapter:

```
public void Send(string message)
```


3. Then add the line for getting the current chat room from the caller's state. The method should look as follows:

```
public void Send(string message)
{
    var room = Clients.Caller.currentChatRoom;
    Clients.Group(room).addMessage(room, message);
}
```


That's all we are going to do on the server for now. But it is worth mentioning that any state can also be written, it's not just for getting state from the client to the server. And as you've already seen, the `Caller` property is of dynamic type, so you can put anything in it, and it will just be there on the client.

4. Open the `index.html` file. We need to change all of the places that dealt with the current chat room variant that we put at the top of the script block. We will now remove the line that sat globally at the top declaring the `currentChatRoom` variant. Then we need to change all of the code that was using this variant. We are now going to be using the state present on the chat Hub directly. Let's start with the code that adds a chat room, and get the current chat room from the Hub's state instead:



```
chat.client.addChatRoom = function (chatRoom) {
    var currentChatRoom = chat.state.currentChatRoom;
    addChatRoomToList(chatRoom);
    if (chatRoom == currentChatRoom) {
        $("#chatRoomsList").val(currentChatRoom);
        addMessageToChatWindow("Welcome to " + currentChatRoom);
    }
}
```



5. When adding a message coming from the server we performed a check whether the client was in the chat room or not. We need to change that as well:

```
chat.client.addMessage = function (room, message) {  
    if (room === chat.state.currentChatRoom)   
        addMessageToChatWindow(message);  
    }  
}
```

6. The last pieces of the puzzle for the JavaScript client are the initialization and when we change the chat room:

```
$.connection.hub.start().done(function () {  
    $("#chatWindow").val("Connected\n");  
     chat.state.currentChatRoom = "Lobby";  
    chat.server.join("Lobby");  
    $("#sendButton").click(function () {  
        chat.server.send($("#messageTextBox").val());  
        $("#messageTextBox").val("")  
    });  
  
    $("#createChatRoomButton").click(function () {  
        chat.server.createChatRoom($("#chatRoomTextBox").val());  
        $("#chatRoomTextBox").val("")  
    });  
  
    $("#chatRoomsList").change(function () {  
         var currentChatRoom = $("#chatRoomsList option:selected").val();  
        chat.state.currentChatRoom = currentChatRoom;  
        chat.server.join(currentChatRoom);  
        clearChatRoomForRoomChange(currentChatRoom);  
    });  
});
```

7. Note that the `send()` function call has changed from using two arguments to one.

8. For a .NET client, this is a bit different; there isn't a state property on the Hub, but an indexer. The Hub can be indexed with the name of the state variable to get and set state. This results in a couple of changes. We start by removing the `currentRoom` variable we had, and now stick it directly on the Hub. Any usage of `currentRoom` must now be changed to use the indexer on the Hub:

```
static void Main(string[] args)
{
    var hubConnection = new HubConnection("http://localhost:1599");
    var chat = hubConnection.CreateHubProxy("chat");

    chat["currentChatRoom"] = "Lobby";
    chat.On("addMessage", (string room, string message) => {
        if (room == (string)chat["currentChatRoom"])
            Console.WriteLine("Received : " + message);
    });

    chat.On("addChatRoom", room => Console.WriteLine("Room added : "+room));

    hubConnection.Start().Wait();

    Console.WriteLine("Connected");

    chat.Invoke("Join", (string)chat["currentChatRoom"]).Wait();

    var line = string.Empty;
    while ((line = Console.ReadLine()) != null)
    {
        if (line.StartsWith("/create"))
        {
            var room = line.Substring("/create".Length + 1);
            chat.Invoke("CreateChatRoom", room);
        }
        if (line.StartsWith("/join"))
        {
            var room = line.Substring("/join".Length + 1);
            chat.Invoke("Join", room).Wait();
            chat["currentChatRoom"] = room;
        }
        else
        {
            chat.Invoke("Send", line).Wait();
        }
    }
}
```

9. Now that we have the current chat room sitting as a state on all of our calls to the chat Hub, we can also implement a way of logging out of a chat room when we're joining one. To do this, make the `Join` method look as follows:

```
public void Join(string room)
{
    if (!string.IsNullOrEmpty(Clients.Caller.currentChatRoom))
        Groups.Remove(Context.ConnectionId, Clients.Caller.currentChatRoom);

    Groups.Add(Context.ConnectionId, room);
}
```

We basically just add a `Remove` word to the `Groups` property with the current chat room.

Summary

From time to time there are bits of information that one could centralize, and not have to pass along on all function and method calls. As shown in this chapter, the current chat room is a good example of such a state. You should now be able to add a state that roundtrips from the server to the client and gives you back the opportunity to simplify your own code.

6

Security

This chapter will cover how you can secure your SignalR connections, require the user to be authenticated with specific roles for calls coming from the client, and also how you can communicate this back to the client in a graceful manner.

The topics covered in this chapter are:

- How to require authenticated users for your connection to accept connections
- How to require a specific role for your connection to accept incoming messages

At this stage the developer should be familiar with securing their SignalR connections.

Becoming private

Security is something all applications need to have a relationship with, in one way or the other. Take a chat like the one we're building for instance, you might want to have private chat rooms or want the entire chat to be private. You might have operations that are only allowed by users with a certain role. Luckily, SignalR has out of the box support for the most common scenarios, and is very extensible if you have more complex scenarios. This chapter will take you through enabling Forms authentication, a common scenario for applications. You could use Windows authentication and others as well, but for our application we're using Forms.

First of all, since our web application is built from HTML files, and not ASPX or ASP.NET MVC controllers, we need to be able to have security kick in for these as well, so that we get redirected to a login page when not authenticated. There are a couple of approaches one can choose from to make the security pipeline of ASP.NET kick in for static files such as HTML files. One could be to enable all HTTP modules to run for all requests, but that would mean a potential performance hit for static content. So instead, we're going to tell ASP.NET to deal with the .html and .htm files specifically:

1. We will need to make a few changes to the Web.config file sitting in the web project to accomplish this. In the <compilation> tag sitting at the top, we need to add the page build providers for the extensions we want to support:

```
<compilation debug="true" targetFramework="4.0">
  <buildProviders>
    <add extension=".html" type="System.Web.Compilation.PageBuildProvider" />
    <add extension=".htm" type="System.Web.Compilation.PageBuildProvider" />
  </buildProviders>
</compilation>
```

2. At the bottom right before the <runtime> tag, we will be adding a web server section, which is specific to IIS7 and higher, for configuring pretty much the same as we did for compilation:

```
<system.webServer>
  <handlers>
    <add name="HTML" path="*.html" verb="GET, HEAD, POST, DEBUG"
        type="System.Web.UI.PageHandlerFactory" resourceType="Unspecified" requireAccess="Script" />
    <add name="HTM" path="*.htm" verb="GET, HEAD, POST, DEBUG"
        type="System.Web.UI.PageHandlerFactory" resourceType="Unspecified" requireAccess="Script" />
  </handlers>
</system.webServer>
```

3. Then, inside the <system.web> tag again, right below the <compilation> tag, we will be adding our security. First we set the authentication to Forms and add a form for our application with some attributes configuring its behavior. We also need to add in an <authorization> tag denying all anonymous users but allowing any logged in users:

```
<authentication mode="Forms">
  <forms name=".signalRChat" loginUrl="login.html" protection="All" path="/" timeout="30"></forms>
</authentication>

<authorization>
  <deny users="?"/>
  <allow users="*" />
</authorization>
```

4. To be able to log in from the login page that we will be creating, we will need an HTTP handler for authenticating and giving us the authentication cookie for any subsequent requests. We'll create the handler shortly. But for now, let's just configure it to allow requests even if we're not logged in. Add the following code snippet right before the `<system.webserver>` tag in `web.config`:

```
<location path="SecurityHandler.ashx">
  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
```

Your `web.config` file should now look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0">
      <buildProviders>
        <add extension=".html" type="System.Web.Compilation.PageBuildProvider" />
        <add extension=".htm" type="System.Web.Compilation.PageBuildProvider" />
      </buildProviders>
    </compilation>

    <authentication mode="Forms">
      <forms name=".signalRChat" loginUrl="login.html" protection="All" path="/" timeout="30"></forms>
    </authentication>

    <authorization>
      <deny users="?" />
      <allow users="*" />
    </authorization>
  </system.web>

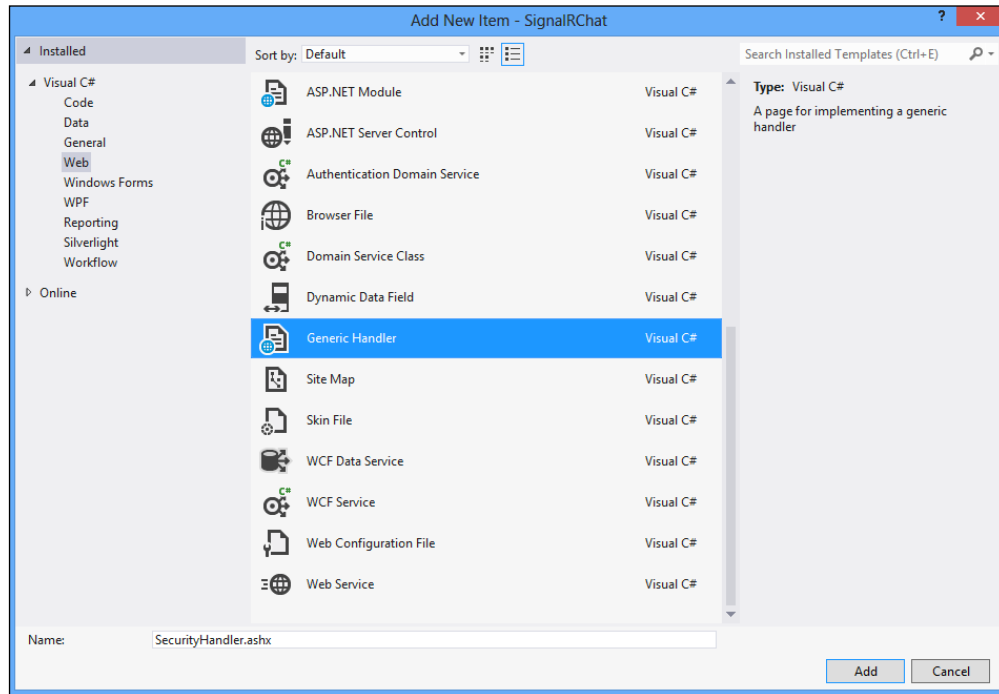
  <location path="SecurityHandler.ashx">
    <system.web>
      <authorization>
        <allow users="*" />
      </authorization>
    </system.web>
  </location>

  <system.webServer>
    <handlers>
      <add name="HTML" path="*.html" verb="GET, HEAD, POST, DEBUG"
        type="System.Web.UI.PageHandlerFactory" resourceType="Unspecified" requireAccess="Script" />

      <add name="HTM" path="*.htm" verb="GET, HEAD, POST, DEBUG"
        type="System.Web.UI.PageHandlerFactory" resourceType="Unspecified" requireAccess="Script" />
    </handlers>
  </system.webServer>

  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="Microsoft.Owin.Host.SystemWeb" publicKeyToken="31bf3856ad364e35" culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-1.0.0.0" newVersion="1.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

- Let's add the security handler that we just configured. Right-click on the web project and navigate to **Add | New item**. Select **Web** and then select **Generic Handler**. Give it the name `SecurityHandler.ashx`:



- We will, for simplicity, just be hardcoding our users, passwords, and roles. At the top of the `SecurityHandler.ashx` file, add the following code snippet:

```
Dictionary<string, string> _usersAndPassword = new Dictionary<string, string>()
{
    { "SomeCreator", "1234" },
    { "SomeChatter", "1234" }
};

Dictionary<string, string[]> _usersAndRoles = new Dictionary<string, string[]>()
{
    {
        "SomeCreator", new[] { "Creator" }
    }
};
```

- Let's add the simple authentication methods for dealing with the users. The authentication will result in a `FormsAuthentication` cookie that we generate in the `AuthenticateUser` method that follows. The cookie will hold the username and all the roles for the user:

```

bool IsValidUser(string userName, string password)
{
    foreach (var user in _usersAndPassword.Keys)
        if (user.ToLowerInvariant() == userName.ToLowerInvariant())
            if (_usersAndPassword[user] == password)
                return true;

    return false;
}

string[] GetRolesForUser(string userName)
{
    foreach (var user in _usersAndRoles.Keys)
        if (user.ToLowerInvariant() == userName.ToLowerInvariant())
            return _usersAndRoles[user];

    return new string[0];
}

void AuthenticateUser(
    HttpContext context,
    string userName,
    params string[] roles)
{
    var ticket = new FormsAuthenticationTicket(1, userName,
        DateTime.Now,
        DateTime.Now.AddMinutes(30),
        false,
        string.Join(";", roles));
    var cookieString = FormsAuthentication.Encrypt(ticket);
    var cookie = new HttpCookie(FormsAuthentication.FormsCookieName, cookieString);
    context.Response.Cookies.Add(cookie);
}

```

8. The handler will need an implementation in `ProcessRequest()` that deals with the incoming authentication. Again, for simplicity, we will be using clear text passwords sitting inside an HTTP form. Of course, it's recommended that you do something a bit more involved than this, especially if you're not using SSL to secure your connection.

```

public void ProcessRequest(HttpContext context)
{
    var userName = context.Request.Form["userName"];
    var password = context.Request.Form["password"];

    if (IsValidUser(userName, password))
    {
        var roles = GetRolesForUser(userName);
        AuthenticateUser(context, userName, roles);
        context.Response.StatusCode = (int)HttpStatusCode.OK;
    }
    else
    {
        context.Response.StatusCode = (int)HttpStatusCode.Forbidden;
    }
}

```


The entire `SecurityHandler` class should look as follows:

```
public class SecurityHandler : IHttpHandler
{
    Dictionary<string, string> _usersAndPassword = new Dictionary<string, string>()
    {
        { "SomeCreator", "1234" },
        { "SomeChatter", "1234" }
    };

    Dictionary<string, string[]> _usersAndRoles = new Dictionary<string, string[]>()
    {
        {
            "SomeCreator", new[] { "Creator" }
        }
    };

    public void ProcessRequest(HttpContext context)
    {
        var userName = context.Request.Form["userName"];
        var password = context.Request.Form["password"];

        if (IsValidUser(userName, password))
        {
            var roles = GetRolesForUser(userName);
            AuthenticateUser(context, userName, roles);
            context.Response.StatusCode = (int)HttpStatusCode.OK;
        }
        else
        {
            context.Response.StatusCode = (int)HttpStatusCode.Forbidden;
        }
    }

    bool IsValidUser(string userName, string password)
    {
        foreach (var user in _usersAndPassword.Keys)
        {
            if (user.ToLowerInvariant() == userName.ToLowerInvariant())
            {
                if (_usersAndPassword[user] == password)
                {
                    return true;
                }
            }
        }

        return false;
    }

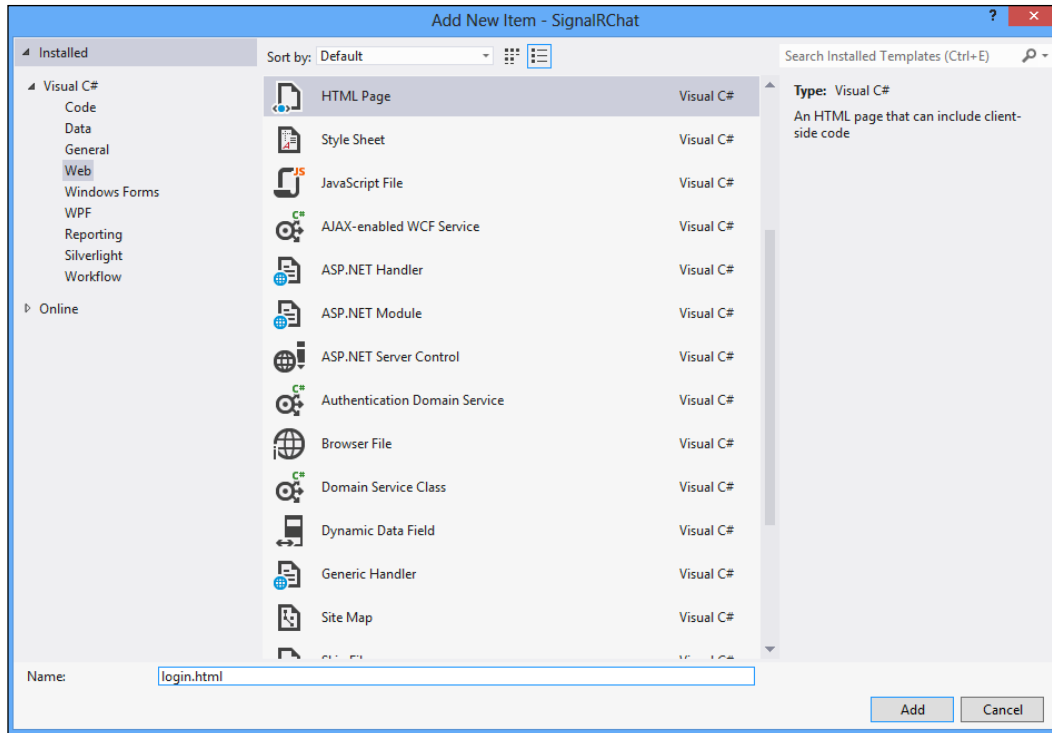
    string[] GetRolesForUser(string userName)
    {
        foreach (var user in _usersAndPassword.Keys)
        {
            if (user.ToLowerInvariant() == userName.ToLowerInvariant())
            {
                return _usersAndRoles[user];
            }
        }

        return new string[0];
    }

    void AuthenticateUser(HttpContext context, string userName, params string[] roles)
    {
        var ticket = new FormsAuthenticationTicket(1, userName, DateTime.Now, DateTime.Now.AddMinutes(30), false, string.Join(";", roles));
        var cookieString = FormsAuthentication.Encrypt(ticket);
        var cookie = new HttpCookie(FormsAuthentication.FormsCookieName, cookieString);
        context.Response.Cookies.Add(cookie);
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

9. Now we will need a page to log us in; the page we configured in `web.config` as the redirect URL when users try to access the site. Right-click on the web project, navigate to **Add | New Item**, select **Web**, and then select **HTML Page**. Name the page `login.html`:



10. In the page we will add a fieldset inside the body holding the input for authentication. This could also have been a form pointing directly to the `SecurityHandler` class, but we will be doing this programmatically instead:

```
<body>
  <fieldset>
    <legend>Login</legend>
    <label for="userNameInput">Username</label><input id="userNameInput" type="text" /><br />
    <label for="passwordInput">Password</label><input id="passwordInput" type="password" /><br />
    <button id="loginButton">Login</button>
  </fieldset>
</body>
```

11. Then we add the following scripts inside the <head> element in the login.html file. The script basically hooks up the **Login** button's click event, and makes a post using jQuery to the SecurityHandler. If it's successful, we redirect to the index.html page; if not, we show an alert message:

```
<script src="Scripts/jquery-1.8.3.min.js"></script>
<script type="text/javascript">
    $(function () {
        $("#loginButton").click(function () {
            var userName = $("#userNameInput").val();
            var password = $("#passwordInput").val();

            $.post("SecurityHandler.ashx", {
                userName: userName,
                password: password
            }).done(function (e) {
                window.location.href = "index.html"
            }).fail(function (e) {
                alert("Not allowed to log in");
            });
        });
    });
</script>
```

The entire login.html should look as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/jquery-1.8.3.min.js"></script>
    <script type="text/javascript">
        $(function () {
            $("#loginButton").click(function () {
                var userName = $("#userNameInput").val();
                var password = $("#passwordInput").val();

                $.post("SecurityHandler.ashx", {
                    userName: userName,
                    password: password
                }).done(function (e) {
                    window.location.href = "index.html"
                }).fail(function (e) {
                    alert("Not allowed to log in");
                });
            });
        });
    </script>
</head>
<body>
    <fieldset>
        <legend>Login</legend>
        <label for="userNameInput">Username</label><input id="userNameInput" type="text" /><br />
        <label for="passwordInput">Password</label><input id="passwordInput" type="password" /><br />
        <button id="loginButton">Login</button>
    </fieldset>
</body>
</html>
```

12. Running the chat should now lead you to the login page and you will not be able to get to the `index.html` page. The hub is now protected as well; so going directly to the URL won't get you there. But we will be explicitly securing the hub. SignalR comes with an attribute called `Authorize()`, similar to the one you find in ASP.NET MVC and so on. There are other mechanisms for securing hubs, but we won't go into that in this book.

The `Authorize()` attribute can be used for both hubs and methods on a hub. It has a couple of options that can be passed to it, such as `Users` and `Roles` holding comma delimited required users and/or roles. But it also has a property called `RequireOutgoing` that tells SignalR what direction it should be securing. By default it is only incoming, but by setting it to `true` it will also be outgoing. We will set it to `true`, so that we secure both directions:

```
[Authorize(RequireOutgoing=true)]
public class Chat : Hub
{
```

13. In addition to requiring authenticated users for the hub, we will be adding a specific role requirement for the `CreateChatRoom()` method that sits on the hub:

```
[Authorize(Roles="Creator")]
public void CreateChatRoom(string room)
{
```

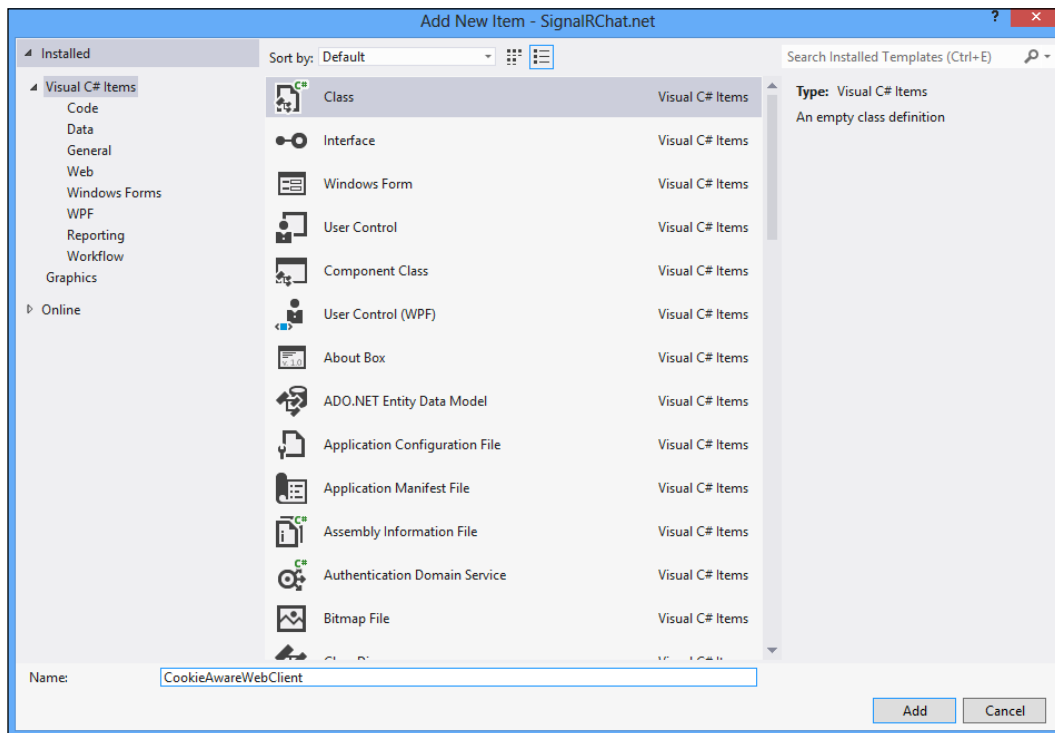
14. Now, the final piece of the puzzle; SignalR uses the underlying credential information found on threads in .NET. This means we will have to set this information on the current thread based on the cookie generated by the security handler. Open the `Global.asax.cs` file and add the following method to that class. This will get the cookie, decrypt it, and put the identity and roles into the `HttpContext` class found in `System.Web`:

```
protected void Application_AuthenticateRequest(object sender, EventArgs e)
{
    if (HttpContext.Current.User != null)
    {
        if (Request.IsAuthenticated == true)
        {
            var ticket = FormsAuthentication.Decrypt(
                Context.Request.Cookies[FormsAuthentication.FormsCookieName].Value);
            var roles = ticket.UserData.Split(';');
            var id = new FormsIdentity(ticket);
            Context.User = new System.Security.Principal.GenericPrincipal(id, roles);
        }
    }
}
```

15. Running the web app should now lead you straight to the `login.html` page. Log in as one of the users, such as `SomeCreator` and its password `1234`, and on logging in you should be redirected to `index.html` where you can do everything you could before. Verify that everything is working by closing the browser and logging in as `SomeChatter` and try to create a new chat room; a new room should not appear.

Now, our .NET client won't be able to do anything at this stage, in fact, the application will just crash. So we need to authenticate from this application as well. Since we don't have any fancy user interface, we will be hardcoding it in this application, but it will at least show us how it's done.

Since our authentication is using a cookie, we will need the cookie in all of our SignalR requests from this client as well. There are a few ways we could do that, but the simplest is probably to implement a specialized `WebClient` that gets the cookie container. Right-click on the `SignalRChat.net` project, navigate to **Add | New Item**, and then select **Class**. Name it `CookieAwareWebClient.cs`:



16. The web client will expose a `CookieContainer` class that will hold the cookies for any requests going through this `WebClient` class. In order to access the cookies, we override the `GetWebRequest()` method so we can intercept it and add our `CookieContainer` as the recipient of any cookies received during the request:

```
using System;
using System.Net;

namespace SignalRChat.net
{
    public class CookieAwareWebClient : WebClient
    {
        public readonly CookieContainer CookieContainer = new CookieContainer();

        protected override WebRequest GetWebRequest(Uri address)
        {
            var request = base.GetWebRequest(address);
            if (request is HttpWebRequest)
                ((HttpWebRequest)request).CookieContainer = CookieContainer;
            return request;
        }
    }
}
```

17. The next step is to take the variable `Site` that sits at the top of the `Main()` method inside the `Program.cs` file and make it a constant at the top of that class:

```
const string Site = "http://localhost:1599";
```

18. In the `Program.cs` class we will now add an `Authenticate()` method that will perform the authentication against the server and return the cookies. The method will simulate what a browser does when posting a form. So we will create a name-value collection that holds our username and password and upload it, which will result in the client receiving the cookies:

```
static CookieContainer Authenticate(string userName, string password)
{
    var postData = new NameValueCollection();
    postData.Add("userName", userName);
    postData.Add("password", password);

    var url = string.Format("{0}/SecurityHandler.ashx", Site);
    var webClient = new CookieAwareWebClient();
    var result = webClient.UploadValues(url, postData);

    return webClient.CookieContainer;
}
```

19. All we need to do now is add the cookies into the `HubConnection` class, so that SignalR will use these on all requests made to the server:

```
var hubConnection = new HubConnection(Site);  
hubConnection.CookieContainer = Authenticate("SomeCreator", "1234");
```

Summary

Security is something that all applications must take into consideration. SignalR just taps into existing infrastructure, both for the client and the server side, making this possible. All we need to do is authenticate and use the infrastructure to our advantage to get our app secured. You should not only be able to apply security in the form described in this chapter, but also get an idea on how to move forward with even more security, such as applying SSL. The next big step now is to make our application scale. With the scale-out options of SignalR, one should be capable of truly scaling to any need. The next chapter will go into depth on how to scale with different options, even into the cloud.

7

Scaling out

This chapter will cover how to scale SignalR across multiple servers. The topics covered in this chapter are as follows:

- The basics of messaging and how SignalR deals with them
- Using SQL Server for scaling out
- Using Azure Service Bus for scaling out
- Using Redis for scaling out

At this stage the developer should be familiar with why one needs to scale out in this manner, and how to do it.

Scaling out

Underneath the covers, SignalR wraps all communication between server and clients into messages holding all of the information with its origin, what the message is for, and the content of the message. By default these messages are kept in memory in the process that hosts your SignalR-based solution. That means that having two servers will not make inter-process communication happen, so one client sitting on one server and another on a second one would not know about each other's messages. With the flexibility of SignalR at the core level of its dealing with these well-defined interfaces, it is fairly simple to make it scale out for different technologies. This is something the SignalR team has done as well; they do provide the ability to scale out in different ways. You can get out of the box support for using a Microsoft SQL Server for temporary storage of messages between servers, or use Windows Azure Service Bus to distribute the messages or even the popular Redis to do this.

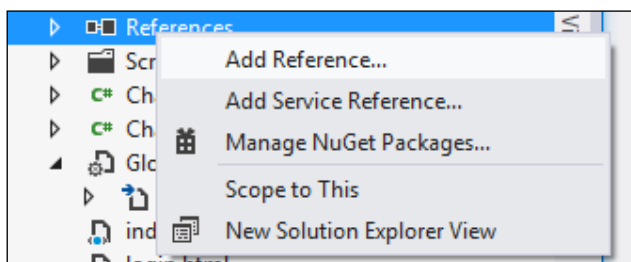
SQL Server

Unfortunately at the time of writing this book, the SQL Server support had not been wrapped up in a NuGet package, so we will need to download the source of SignalR and build it to get the assembly we need to be able to enable this using the following steps:

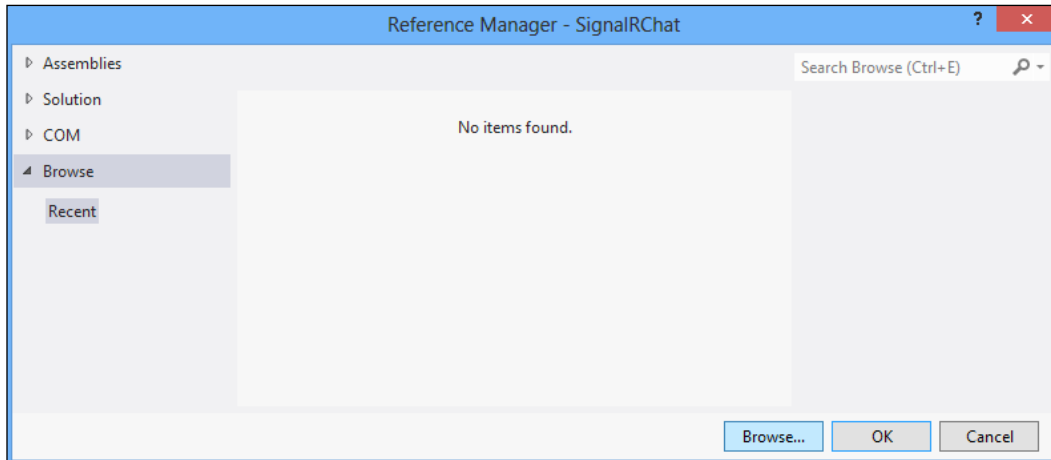
1. Point your browser to <http://github.com/SignalR/SignalR> and either clone the repository if you have Git or GitHub for Windows already installed, or click on the **ZIP** button to download a ZIP file with all of the sources in it:



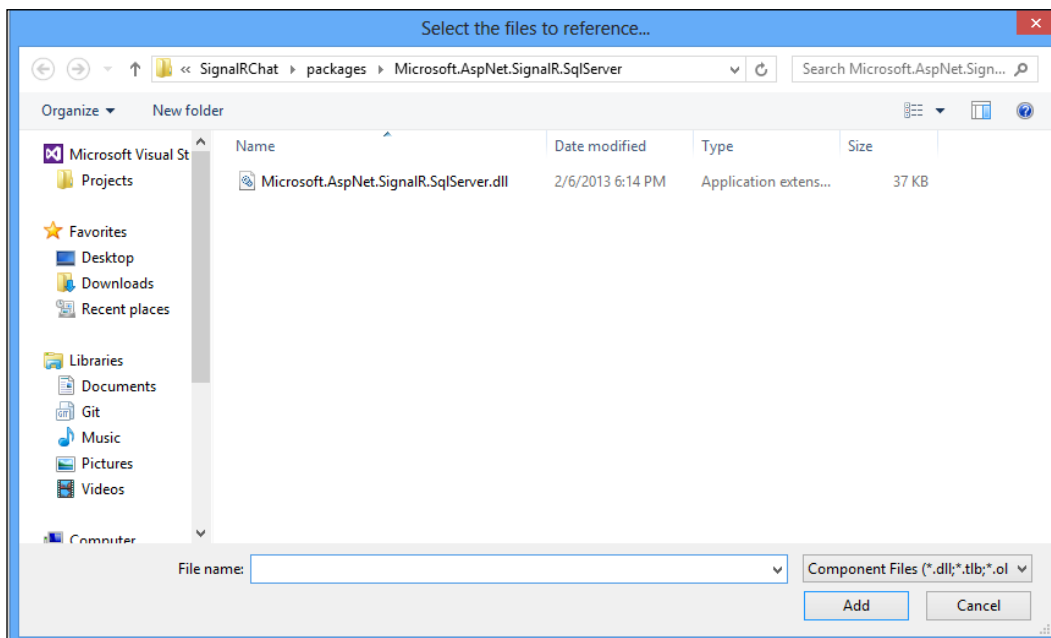
2. Open the `Microsoft.AspNet.SignalR.sln` file directly in Visual Studio 2010/2012 and just build it. Then go back to the folder where you put SignalR and navigate into the `src\Microsoft.AspNet.SignalR.SqlServer\bin\Debug` folder and grab the `Microsoft.AspNet.SignalR.SqlServer.dll` file that sits there and put it into a folder in the chat solution, for instance, in the `packages` folder that sits there.
3. Now that we have the DLL file, we need to reference it in our project. Right-click on the references inside the web project and select **Add Reference**:



4. Select **Browse** in the dialog to find the DLL file that we just got by building SignalR:



5. Create a folder called `Microsoft.AspNet.SignalR.SqlServer` under the packages folder, to follow the convention that was already established by using NuGet for the other packages:

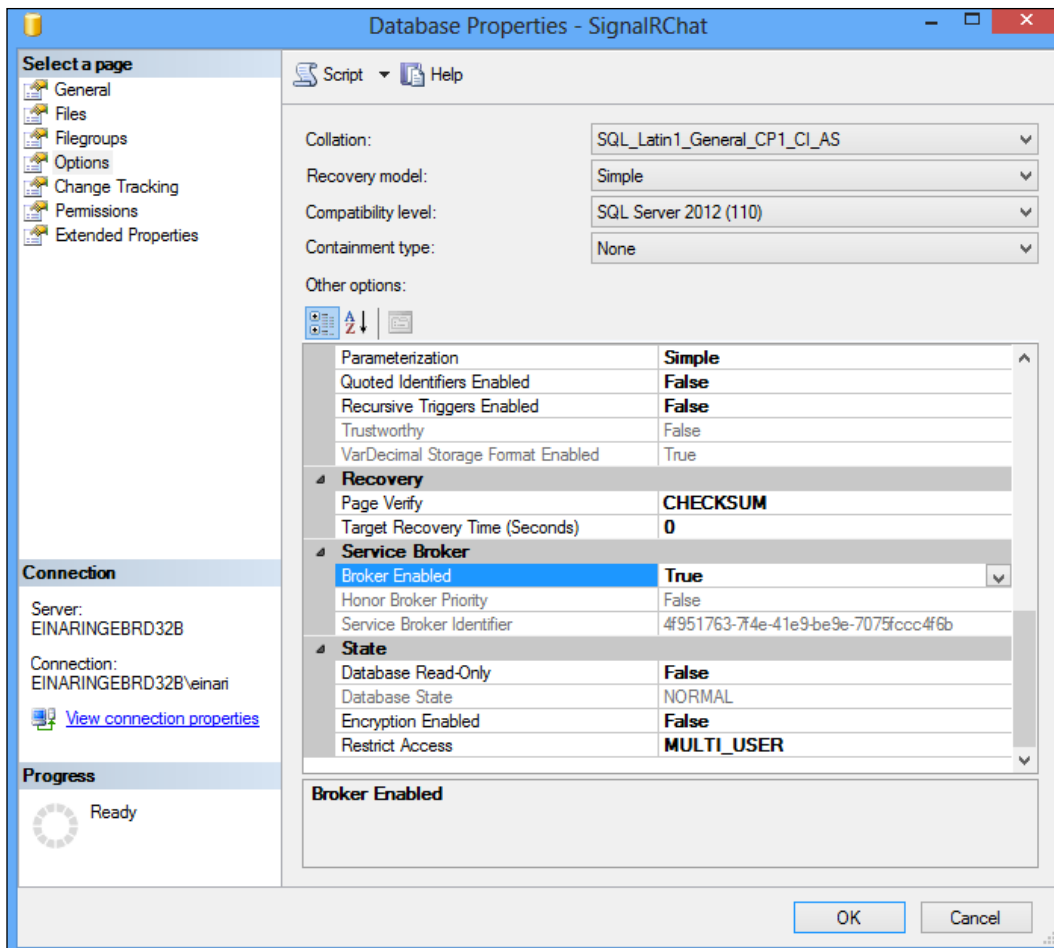


6. Enabling SQL Server in the code is very easy; it is basically one method call. There is an extension method called `UseSqlServer()` that we are going to use which will give the connection string to the database. Of course, you will have to create a database in your SQL Server instance at this point and point `connectionstring` to your instance and your database. I've used SQL Server 2012 and `connectionstring` is pointing to a local instance and a database called `SignalRChat`:

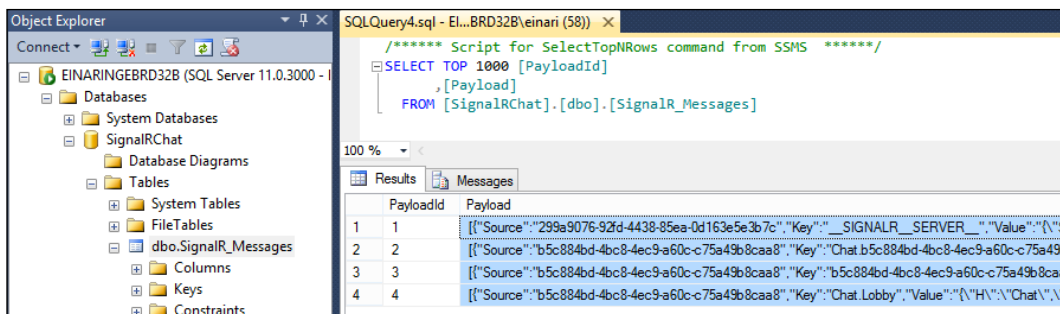
```
namespace SignalRChat
{
    public class Global : HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable
                .Routes
                .MapHubs();

            GlobalHost
                .DependencyResolver
                .UseSqlServer(
                    "Data Source=(local);"+
                    "Initial Catalog=SignalRChat;"+
                    "Integrated Security=True"
                );
        }
    }
}
```

7. In order for SignalR to be able to use SQL Server as a messaging backend, we need to enable something called **Service Broker** for our database. After creating your database, right-click on it in the **SQL Server Management Studio** window and select **Properties**. In the **Options** page, scroll down until you find the **Service Broker** section and enable the **Broker Enabled** flag:



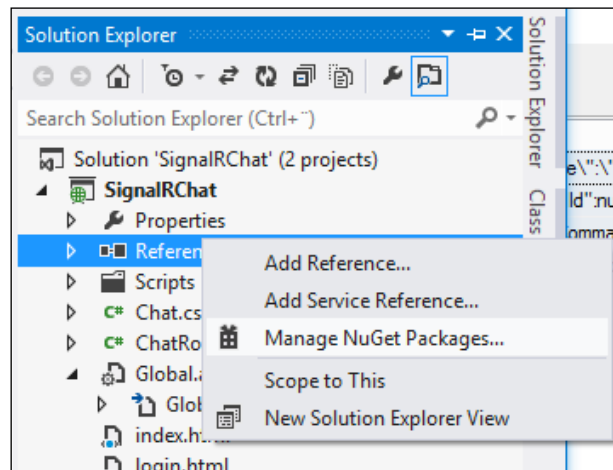
8. We should now be able to run our application and it will generate messages in the SignalR_Messages table:



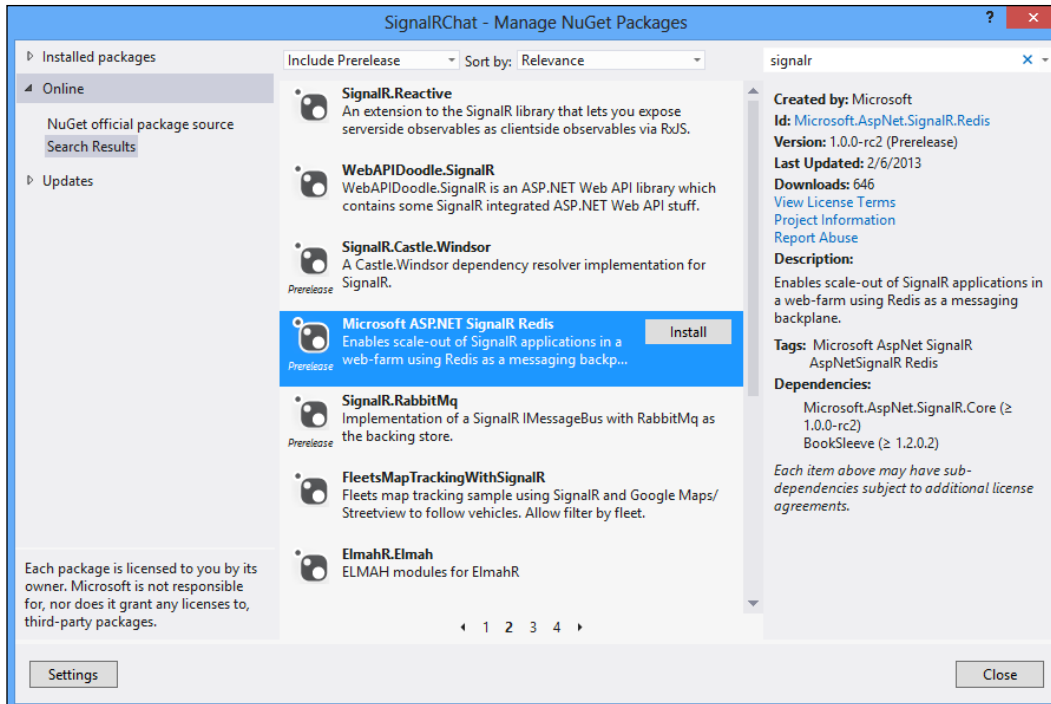
Redis

Redis is another option that can be used with SignalR for scaling out. Redis is an open source distributed key-value store. It is very popular in the Unix space, and has also been adopted by Microsoft. It's fairly easy to get running on Azure or other cloud options. If you want to try things out with Redis locally, the following is the procedure:

1. Download the source that Microsoft has published through their Open Tech initiative on Github at <https://github.com/Microsoft/redis>. Follow the guide there, build and run it. Once it is running, we can get going with configuring our chat application for Redis instead of the SQL solution.
2. We start off by adding a NuGet package for Redis, right-click on the **References** option in the **Web** project and select **Manage NuGet Packages**:



- Make sure that the **include prerelease** option is selected and you select **Online**, then type in `signalr` and navigate in the results until you find the package named **Microsoft ASP.NET SignalR Redis** and click on **Install** in it:

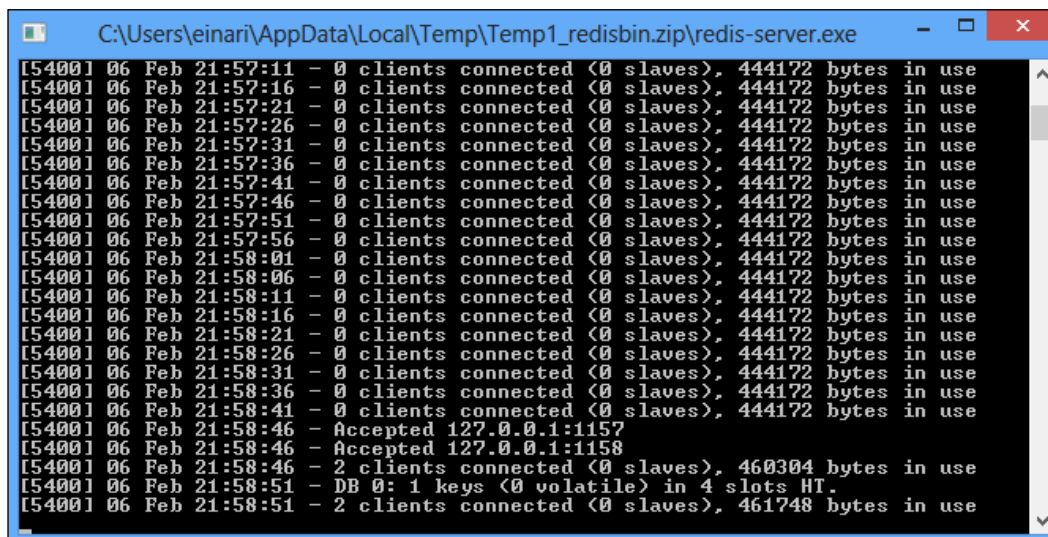


- As with SQL Server, the configuration is really simple. Use the `.UseRedis()` extension method and point it to your Redis server:

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

            GlobalHost
                .DependencyResolver
                .UseRedis(
                    "localhost",
                    6379,
                    "",
                    new[] { "signalr.key" });
        }
    }
}
```

5. You should see the result in the Redis console output directly as follows:

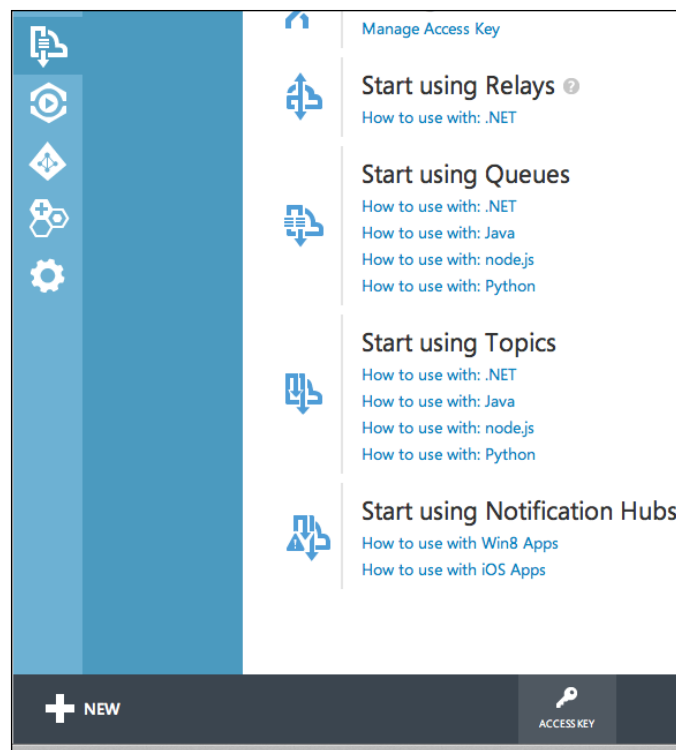


```
C:\Users\einari\AppData\Local\Temp\Temp1_redisbin.zip\redis-server.exe
[5400] 06 Feb 21:57:11 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:16 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:21 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:26 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:31 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:36 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:41 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:46 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:51 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:57:56 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:01 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:06 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:11 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:16 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:21 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:26 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:31 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:36 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:41 - 0 clients connected (0 slaves), 444172 bytes in use
[5400] 06 Feb 21:58:46 - Accepted 127.0.0.1:1157
[5400] 06 Feb 21:58:46 - Accepted 127.0.0.1:1158
[5400] 06 Feb 21:58:46 - 2 clients connected (0 slaves), 460304 bytes in use
[5400] 06 Feb 21:58:51 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[5400] 06 Feb 21:58:51 - 2 clients connected (0 slaves), 461748 bytes in use
```

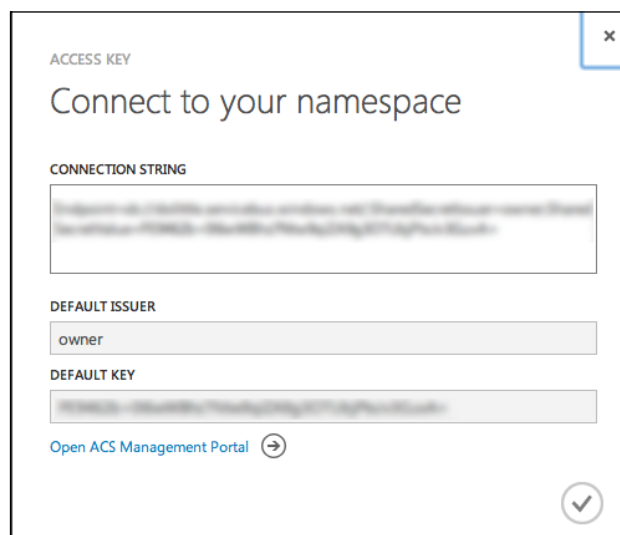
Azure

A third option that's available out of the box is the usage of Azure's Service Bus, a distributed messaging system for Microsoft's cloud solution—Azure. We will cover it briefly in this book as it requires you to have the Azure SDK installed to implement it fully. Once you have installed the Azure SDK, you will need to add a cloud project to your solution and add the web project as a website to the cloud project. When you have all that done you need to set the cloud project as the startup project. The reasoning behind this is that it needs to be running inside the Azure emulator to be able to this, so it's relying on infrastructure to do this. Perform the following steps to do so:

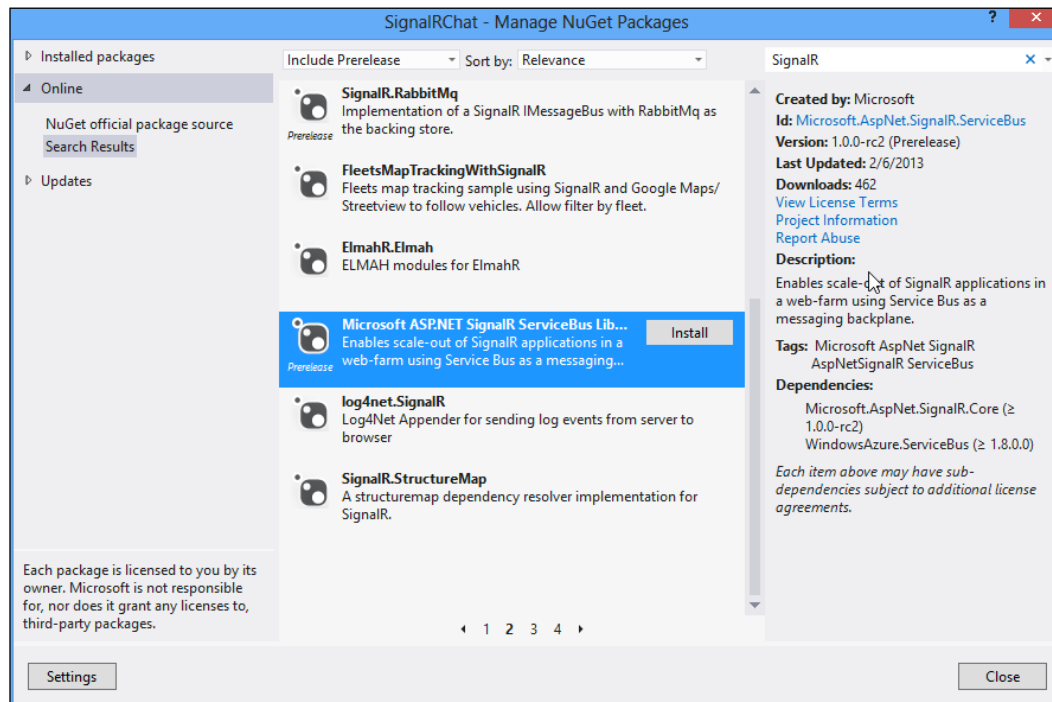
1. Log into your Windows Azure portal and go to **Service Bus**. At the bottom you'll find something called **Access Key**; click on it:



2. Here you'll get the `connectionstring` value that you'll need to be able to connect to the Service Bus from the application; copy this:



- Back in Visual Studio, we're going to add a NuGet package to the project. Right-click on **References** inside the **Web** project and select **Manage NuGet Packages**, make sure **Include Prerelease** is selected and you've selected **Online**, and then enter **SignalR** in the search. Browse through until you find the **Microsoft ASP.NET SignalR ServiceBus Library**; click on **Install**:



- Now, we simply do as we did with Redis and SQL Server – call an extension method for configuring the usage of the service bus – now it's time to paste in the `connectionstring` value that you copied earlier:

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

            var connectionString = "Endpoint=sb://dolittle.servicebus.
GlobalHost
                .DependencyResolver
                .UseWindowsAzureServiceBus(connectionString, 0);
        }
    }
}
```

Summary

When forgotten, often a source of debugging nightmare, scales out. State kept in memory on one server is not available on another, leading to weird scenarios and result. This is also vital when applying SignalR in a multiserver environment. There is no guarantee as to which server the SignalR is connecting to, and also if the client needs to reconnect, so the scale-out option is very vital to the story. With the different options described in this chapter you should now be able to scale in a on-premise solution as well as in the cloud. Moving on from here, we'll dive into monitoring and the ability to do diagnostics when using SignalR.

8

Monitoring

This chapter will highlight the following tools that exist to help you monitor the traffic in SignalR:

- Using Fiddler
- Enabling performance counters

At this stage, the developer should be capable of monitoring their SignalR-enabled apps and see the traffic more easily.

Looking under the covers with monitoring

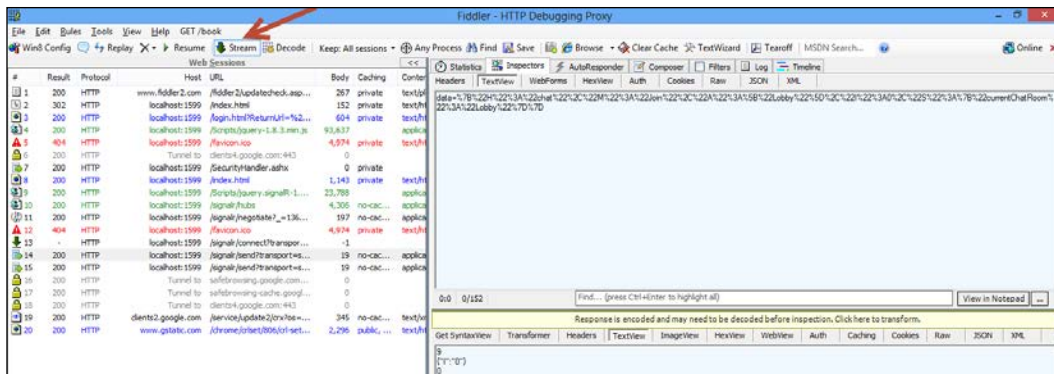
Anyone who has been in the software industry more than 15 minutes knows that things don't necessarily go the way one plans out, or things don't work exactly how one thinks. Sometimes we need to attach a debugger or monitoring tool of some kind to figure out what is going on.

Fiddler

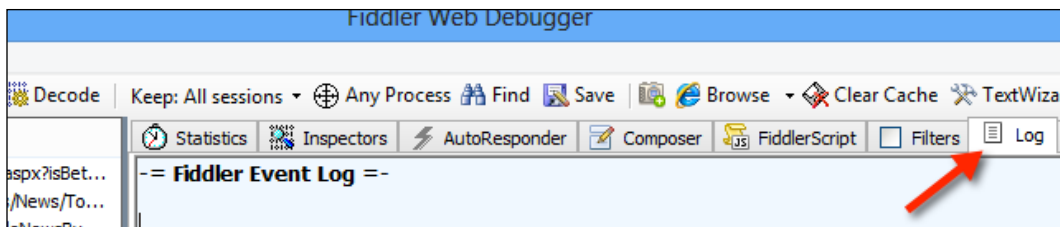
A popular, free, and very good tool for debugging HTTP traffic is Fiddler. You can download it for free from <http://fiddler2.com>. It gives you the opportunity to monitor all HTTP requests happening on your computer.

The following list gives a small review of Fiddler:

1. Fiddler sets itself as a proxy for all traffic, and in order to get the best experience from it you need to enable streams, otherwise SignalR will fall back to long-polling, but not immediately – typically after three-five seconds:



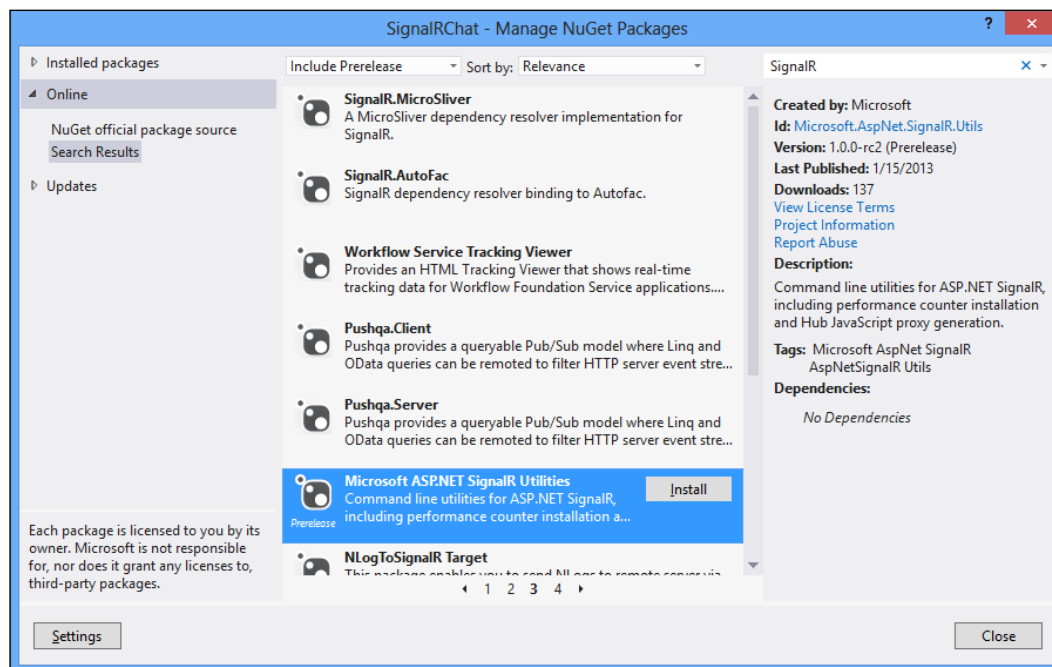
2. If the browser and server support web sockets, SignalR might choose to use them as its preferred transport. In this case, you want to open the Log tab:



Performance counters

Monitoring messages on a higher level to see the throughput of your application, number of failing messages, and such is vital when putting a system into production. SignalR has a utilities project that gives you performance counters that can be installed on the server(s) that host your application, as follows:

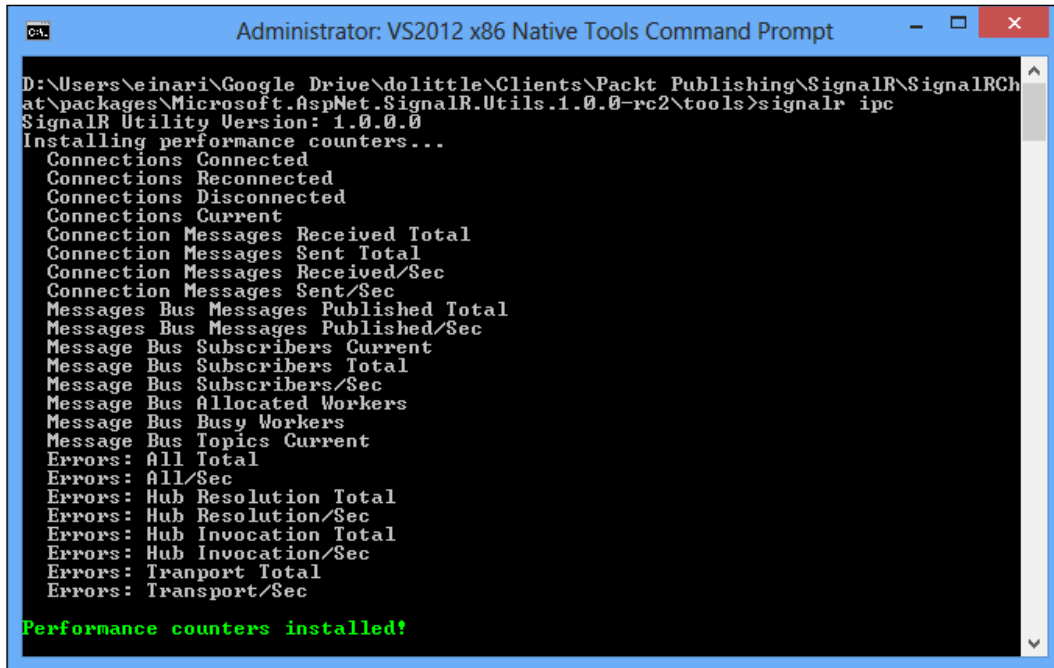
1. The utility is available through NuGet as a package. So right-click on the references of any of the projects and select **Manage NuGet packages**, find the package called **Microsoft ASP.NET SignalR Utilities** and install it:



In order to install the performance counters, we need to open a command prompt in **Administrator** mode.

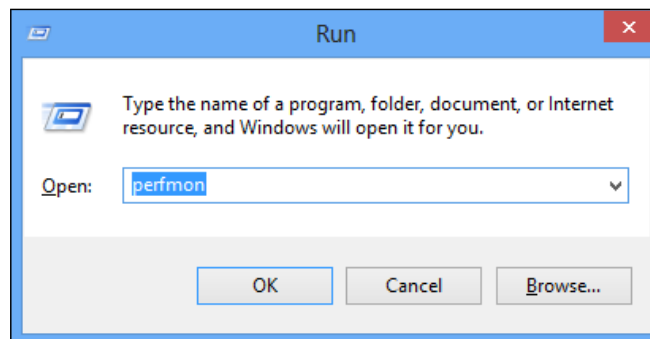
2. Navigate to the path of your solution; inside it you should find a folder called `packages`, and inside it a folder called `Microsoft.AspNet.SignalR.Utils.1.0.0.1` or similar, depending on the version you installed; within this you'll find a folder called `tools`.

- Now that you've navigated through all of these, enter `signalr ipc` and press *Enter*. This will install all the performance counters:

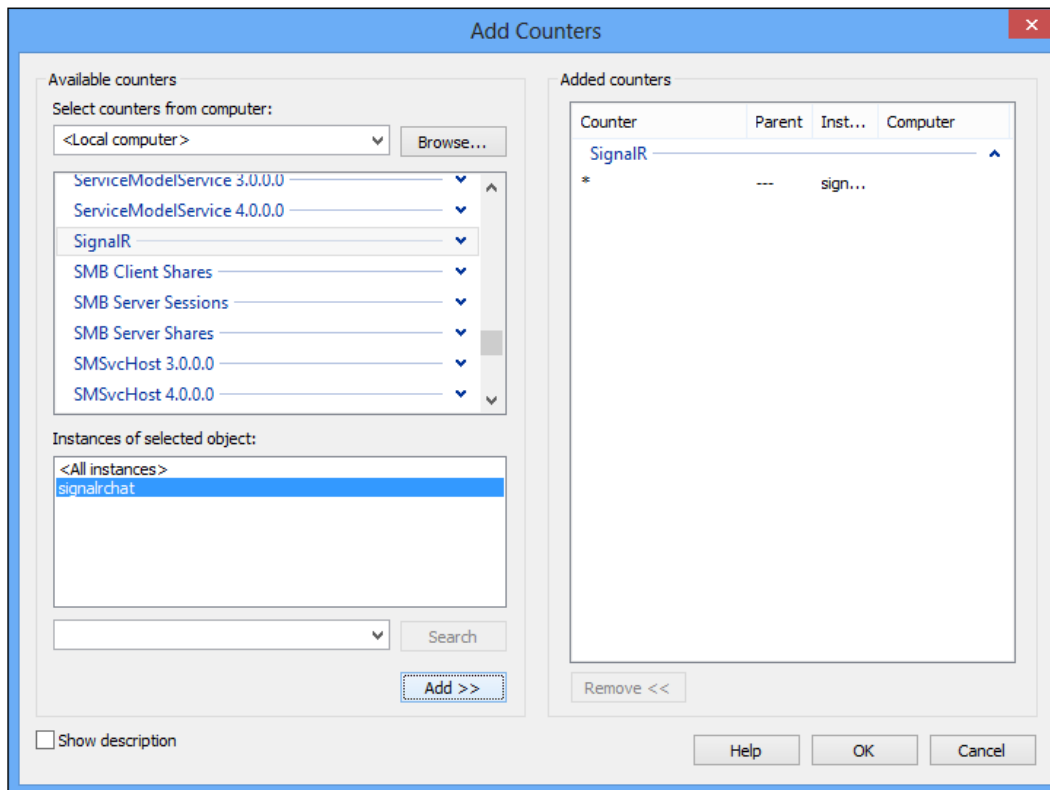


```
Administrator: VS2012 x86 Native Tools Command Prompt
D:\Users\einari\Google Drive\dolittle\Clients\Packt Publishing\SignalR\SignalRCh
at\packages\Microsoft.AspNet.SignalR.Utils.1.0.0-rc2\tools>signalr ipc
SignalR Utility Version: 1.0.0.0
Installing performance counters...
Connections Connected
Connections Reconnected
Connections Disconnected
Connections Current
Connection Messages Received Total
Connection Messages Sent Total
Connection Messages Received/Sec
Connection Messages Sent/Sec
Messages Bus Messages Published Total
Messages Bus Messages Published/Sec
Message Bus Subscribers Current
Message Bus Subscribers Total
Message Bus Subscribers/Sec
Message Bus Allocated Workers
Message Bus Busy Workers
Message Bus Topics Current
Errors: All Total
Errors: All/Sec
Errors: Hub Resolution Total
Errors: Hub Resolution/Sec
Errors: Hub Invocation Total
Errors: Hub Invocation/Sec
Errors: Transport Total
Errors: Transport/Sec
Performance counters installed!
```

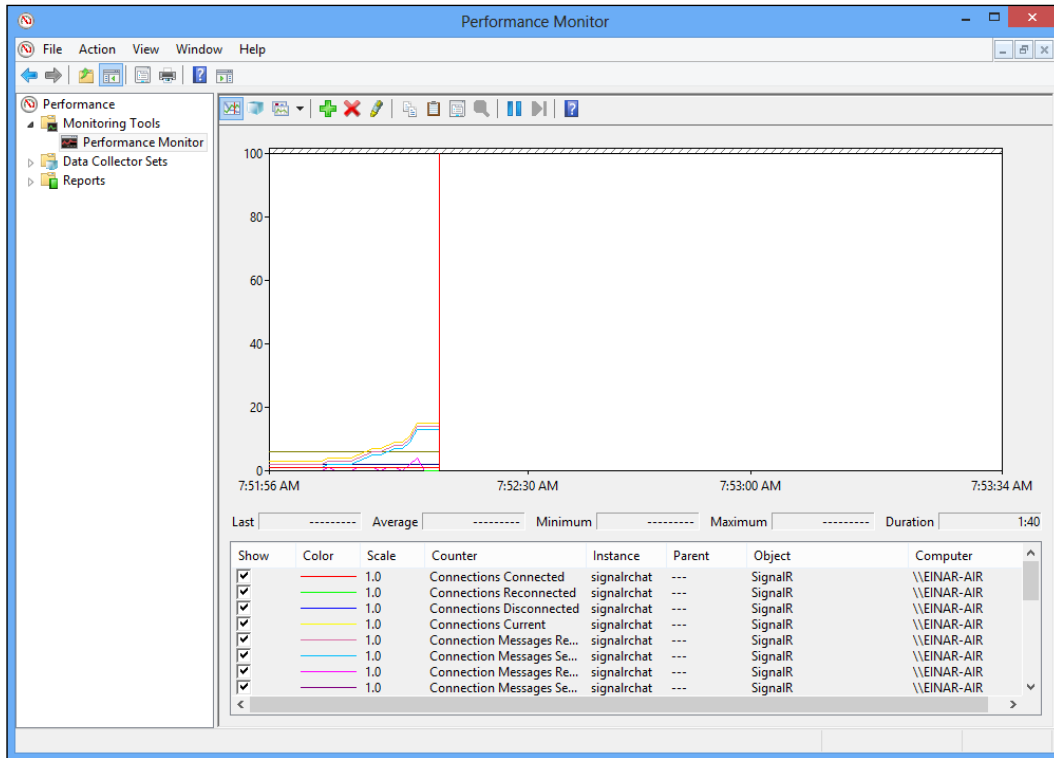
- To see the performance counters, we need to open perfmon (**Start | Run** or for Windows 8 press WinKey+R) and type `perfmon` and press *Enter*:



5. Inside perfmon you can expand the monitoring tools and click on the **Performance Monitor** node and you should see a graph. Click on the big + button at the top so that you can add the SignalR counters you want to look at. If you have your application running, you should see it in the **Instances of selected object** filter list:



6. Once added, you can try out the app by sending messages and see the result in the graph.



7. After putting your solution into production, you might want to consider disabling any performance counters enabled on your server by entering `signalr upc` in the console:

```
Administrator: Developer Command Prompt for VS2012
C:\Projects\SignalRChat\packages\Microsoft.AspNet.SignalR.Utils.1.0.1\tools>signalr.exe upc
SignalR Utility Version: 1.0.0.0
Performance counters uninstalled!
C:\Projects\SignalRChat\packages\Microsoft.AspNet.SignalR.Utils.1.0.1\tools>
```

Summary

Using Fiddler and the performance counters you should now be able to both debug and find potential bottlenecks in your system. Although it might feel a bit primitive and different tools from what might be used to from just developing vanilla web apps, they are very detailed and should be of great assistance. Moving on from the experience of using the built-in WebDev (Cassini), IISExpress, or even a full IIS server, we will take on how you can host a SignalR inprocess in your own application using something called OWIN.

9

Hosting a server using OWIN

This chapter will cover how to set up code to host SignalR using OWIN.

Topics covered in this chapter are:

- Getting started with OWIN
- Setting up the server and making it run
- Connecting the chat client to the self-hosted server

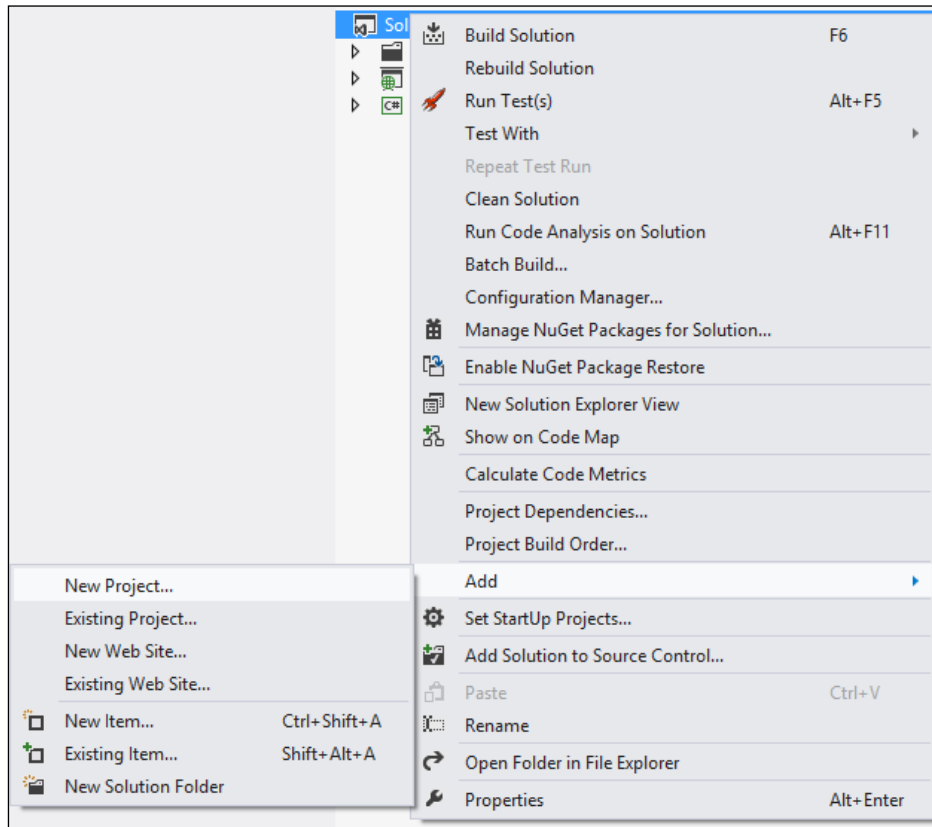
At this stage, the developer should be familiar with how the server works and how to set it up in their own app. They should have a working sample of the chat working with the OWIN server.

Self hosting

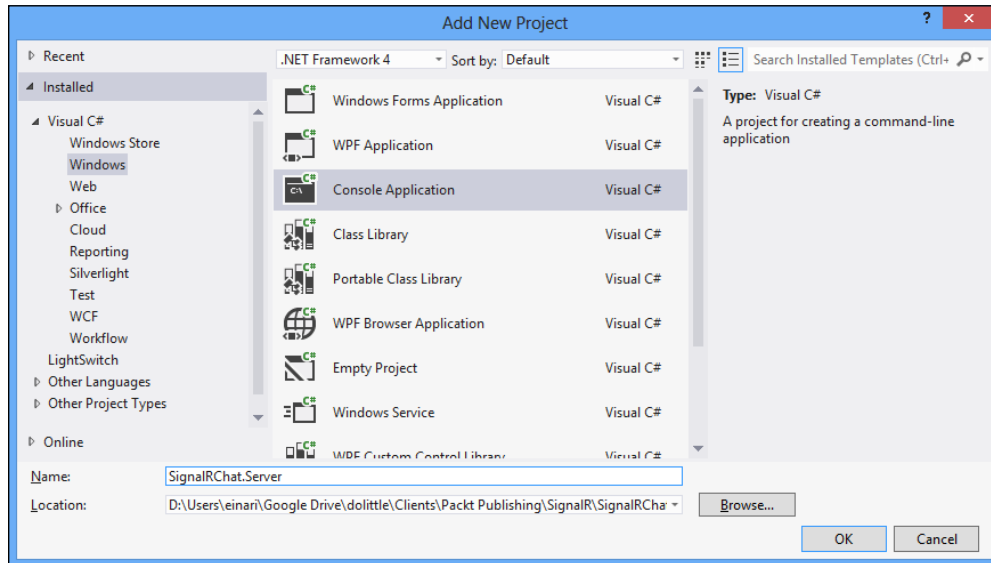
Sometimes you really don't want to have a big footprint on your application when you're deploying. You don't want to have the IIS dependency or other web server software, you just want your own executable and that's it. SignalR does support this out of the box, using something called **OWIN (Open Web Interface for .NET)**. OWIN defines a standard interface between web servers running .NET and web applications.

Perform the following steps for self hosting:

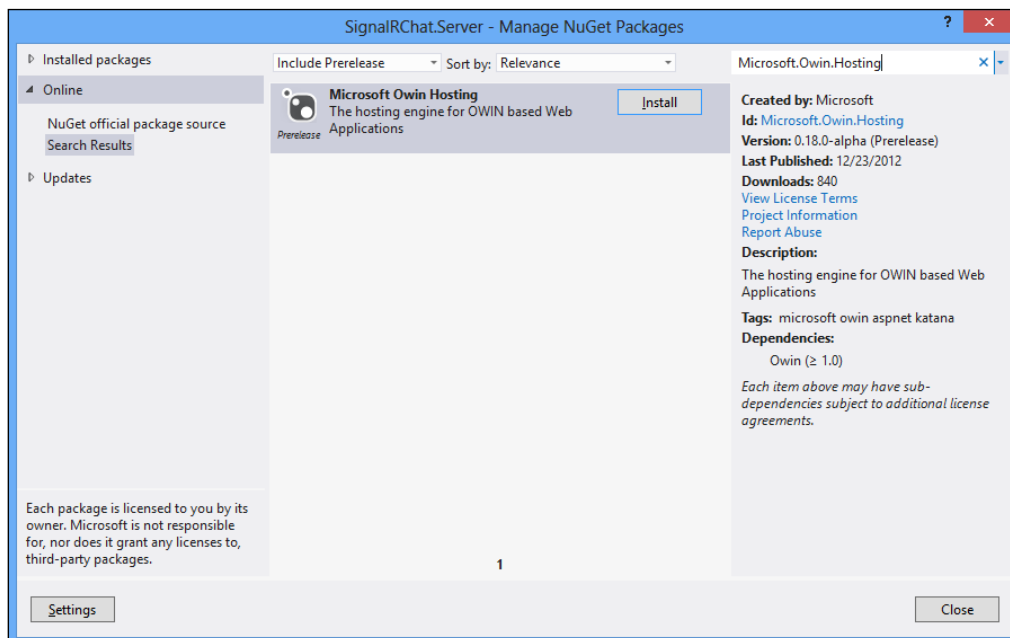
1. Let's start by creating a new console application. Right-click on the **Solution** option and navigate to **Add | New Project**:



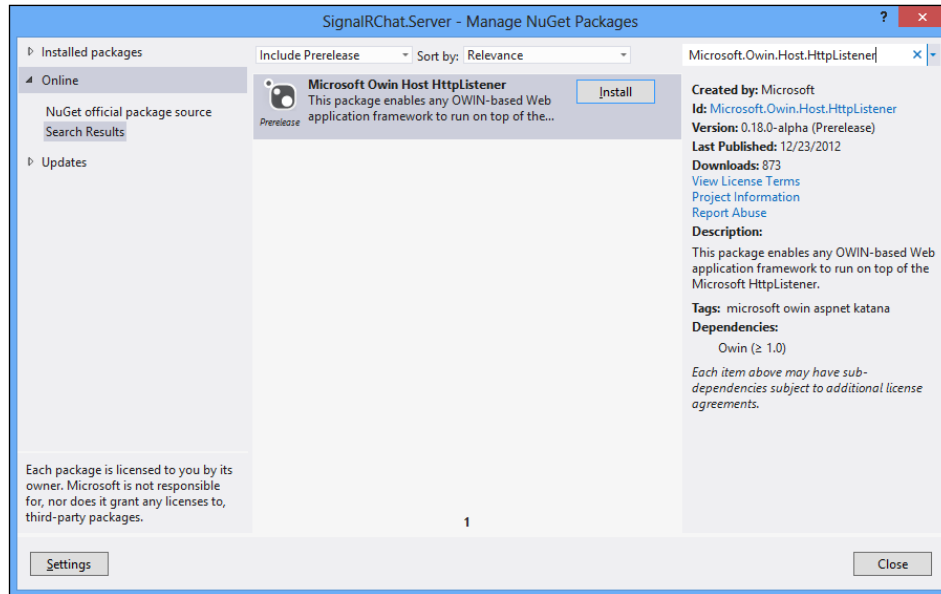
- We're going to add another console application; call it `SignalRChat.Server`:



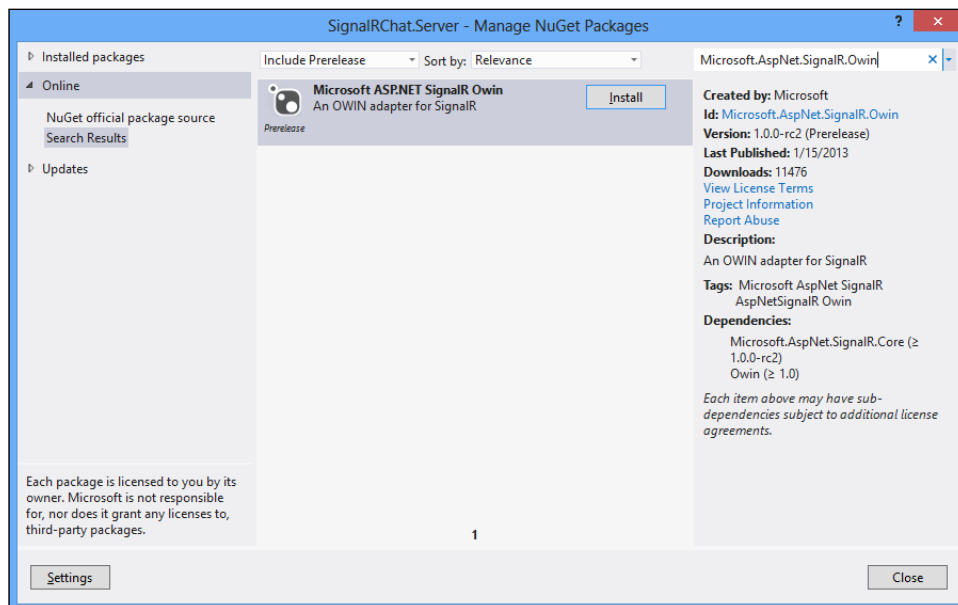
- Now we need to get some NuGet packages. Inside the new project, right-click on **References**, and choose **Manage NuGet Packages**, type in `Microsoft.Owin.Hosting`, and click on **Install**:



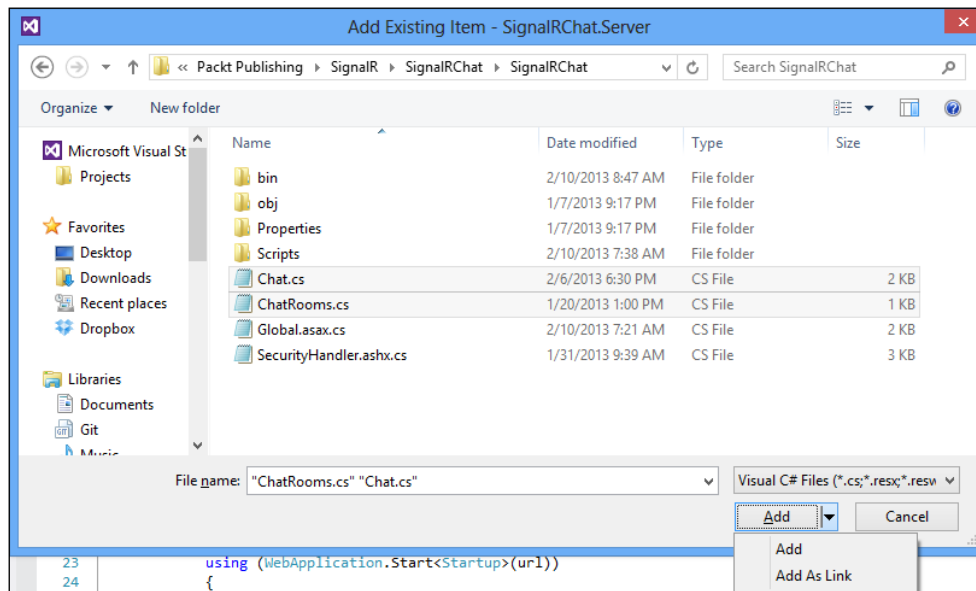
- The next package we're going to need is the **Microsoft.Owin.Host.HttpListener**. Repeat the previous step, but now filter with this package:



- The third and last package that we're going to need is the **Microsoft.AspNet.SignalR.Owin** package. Do as before but filter by this package.



- The next thing we're going to do is to reuse the chat Hub that we have already created. Right-click on the **SignalRChat.Server** project, and navigate to **Add | Existing Item**. Navigate to the **SignalRChat** folder, multi-select **Chat.cs** and **ChatRooms.cs**, and click on the little arrow to the right of the **Add** button, and choose **Add As Link**:



- Inside the **Program** class we're going to use **OWIN** to start **WebApplication**. The **Start** method takes a generic parameter to a class that we'll create which will configure the application. The parameter that is used, is the URL from which the application will be accessible.

```
class Program
{
    static void Main(string[] args)
    {
        var url = "http://localhost:8080";
        using (WebApplication.Start<Startup>(url))
        {
            Console.WriteLine("Server running on {0}", url);
            Console.ReadLine();
        }
    }
}
```


8. Then we need to add a class called `Startup`; right-click on the `SignalRChat.Server` project and navigate to **Add | New Item**, select class, and call it `Startup.cs`. The configuration is going to be simple, we need to map all the Hubs with a configuration enabling cross domain, and we are also going to need some security handling, since we have secured the Hub. We will create the security inspection handler in a second:

```
public class Startup
{
    public void Configuration(IAppBuilder builder)
    {
        builder.Use(typeof(SecurityInspectionHandler));
        var hubConfiguration = new HubConfiguration()
        {
            EnableCrossDomain = true,
            EnableJavaScriptProxies = true,
            EnableDetailedErrors = true,
        };
        builder.MapHubs(hubConfiguration);
    }
}
```

9. Now, we will be putting in a new class called `SecurityInspectionHandler`; add it to `SignalRChat.Server`. This handler will only "fake" security – basically saying that all requests are from the same user, which it obviously should not be – but it proves how you can intercept and deal with security. Normally you want to keep metadata related to the connection ID of the client connected to tell whether or not it is authenticated.

```
public class SecurityInspectionHandler
{
    Func<IDictionary<string, object>, Task> _appFunc;

    public SecurityInspectionHandler(Func<IDictionary<string, object>, Task> appFunc)
    {
        _appFunc = appFunc;
    }

    public Task Invoke(IDictionary<string, object> environment)
    {
        environment["server.User"] =
            new GenericPrincipal(
                new GenericIdentity("SomeCreator"), new[] { "Creator" });
        return _appFunc.Invoke(environment);
    }
}
```

10. Open the `index.html` file in the Web project, and set the Hub URL explicitly, right before the `start()` function when the Hub connection is called:

```
$.connection.hub.url = "http://localhost:8080/signalr";  
$.connection.hub.start().done(function () {
```

11. In the `SignalRChat.Net` project, open the `program.cs` file, and change the Site URL to point to the new self-hosted server:

```
const string Site = "http://localhost:8080";
```

Both the clients should now be able to connect to the new server.

Summary

Hosting any web solution in your own process can be very useful in many scenarios; with the details in this chapter, you should be well on your way to doing just that and having SignalR be your transport for communication. With everything pretty much in place with our chat and also having all of the possible server options, it's time to start looking at another client—Windows 8 Store applications.

10

WinJS and Windows 8

This chapter will cover how to get started with Windows 8 and SignalR using JavaScript.

The topics covered in this chapter are:

- Using SignalR without using proxy
- What it takes to port the chat application to Windows 8

At this stage, the developer should be familiar with how to use SignalR in a WinJS-enabled Windows 8 application. The sample app built throughout the book will now be made ready for Windows 8.

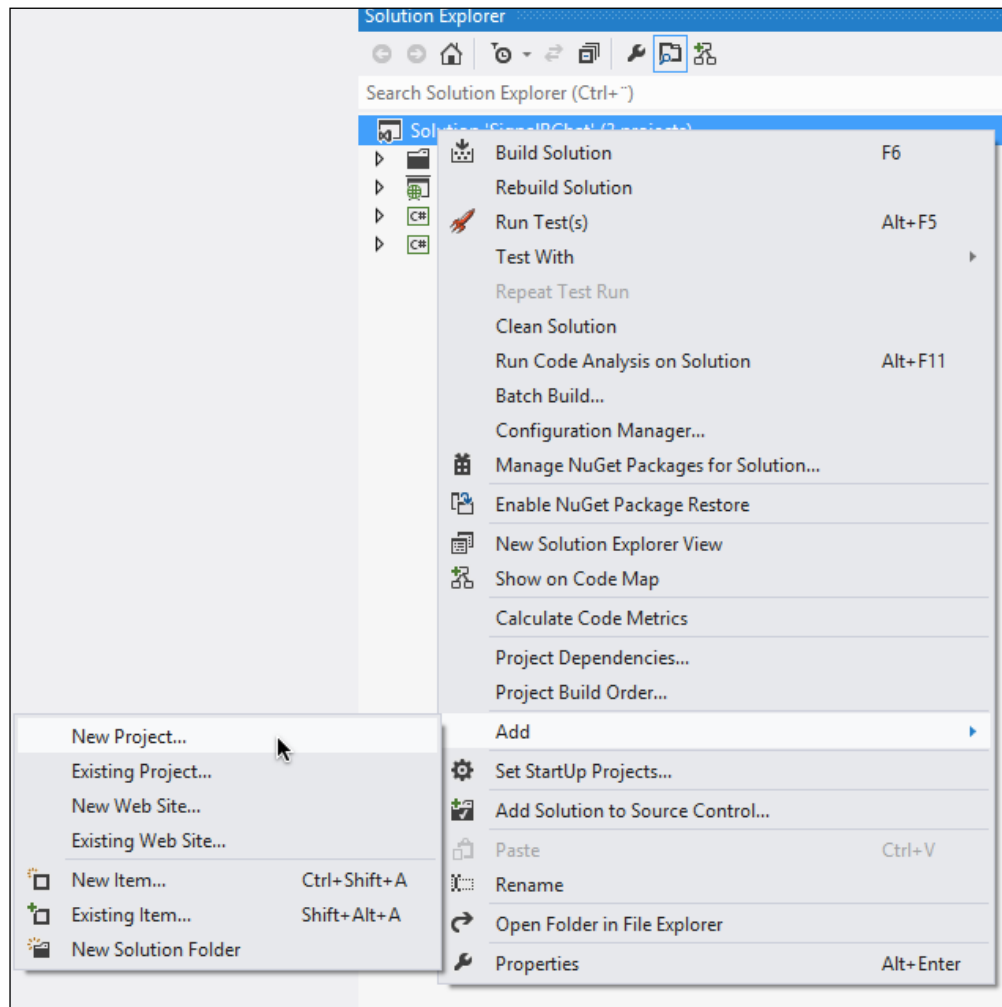
WinJS

When Microsoft launched Windows 8, it came with two worlds – the good old Windows we've known for years (except for the **Start** menu), and a new world that has gone through a few names – ending with modern UI. This new world of tile-based Windows is optimized for touch, and has a new programming model to go with it, in fact, a set of new modern APIs that are common between C++, C#, and VB developers, but also bringing JavaScript into the mix – WinJS.

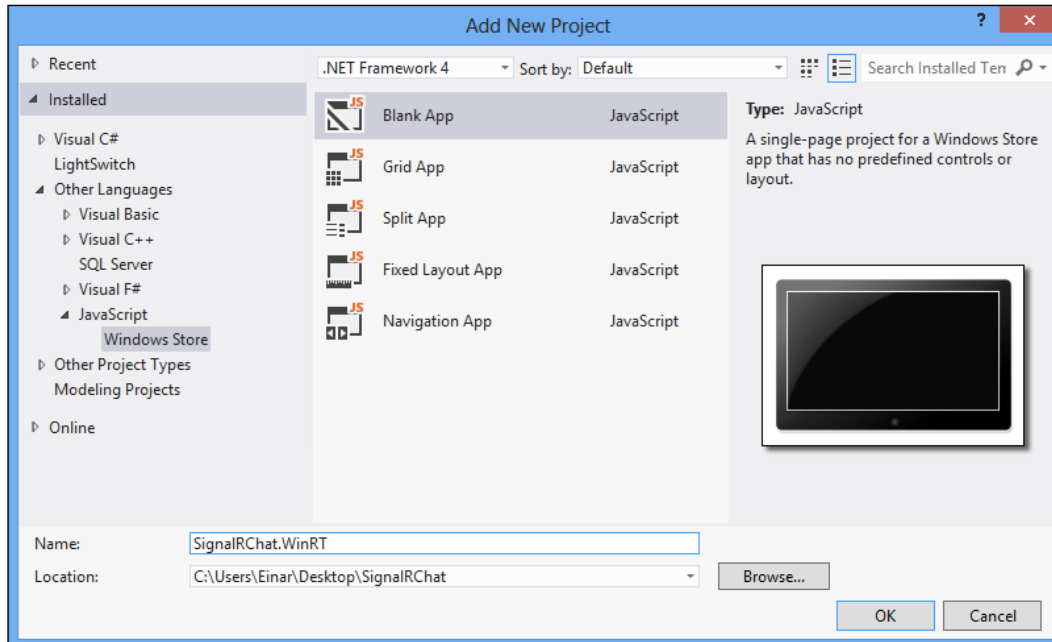
With WinJS, we're now able to create Windows applications that can be published to the Windows Store using web technologies – HTML, CSS, and JavaScript. In this chapter, we're going to take the existing app we have and make it work on Windows 8 as a Windows Store app by taking the existing JavaScript and HTML code that we have, without any modification, and use it in a Windows 8 Store app for JS.

SignalR does have a client for Windows 8 apps written in C# as well, but this book will not go into details about it. The purpose is just to show that SignalR can in fact be used in apps like this as well, which can be done by performing the following steps:

1. First of all, we need a new project to be added to our solution. Right-click on the **Solution** option, and go to **Add | New Project**:



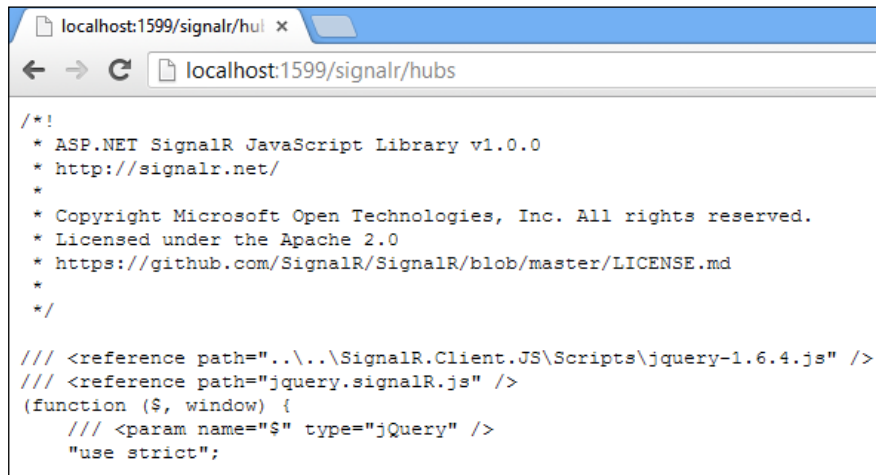
2. Navigate to **Other Languages | JavaScript | Windows Store** in the left pane; then create a new **Blank App** application—call it `SignalRChat.WinRT`:



3. The next things that we want to have are the proxies for the Hub, which make it so convenient to work with SignalR and Hubs. We don't have to have the proxies, but they do make it simpler to work with. But, we can't connect to the web server to have the proxies be generated for us at runtime, we need them in our application package for the app to work. This is a limitation of Windows 8 apps.
4. There are basically two ways we can go about doing this—getting it directly from the web browser or using the tool that comes with the `utilities` project that we used in the previous chapter. The `signalR.exe` file from the previous chapter has a `ghp` option that can be used to point it towards your web server and then it will output the appropriate JavaScript file. But for this sample, we'll be going directly with a browser and just getting the proxy.

5. Make sure that your SignalR chat web project is running, and then open a browser, and point it to the following URL:

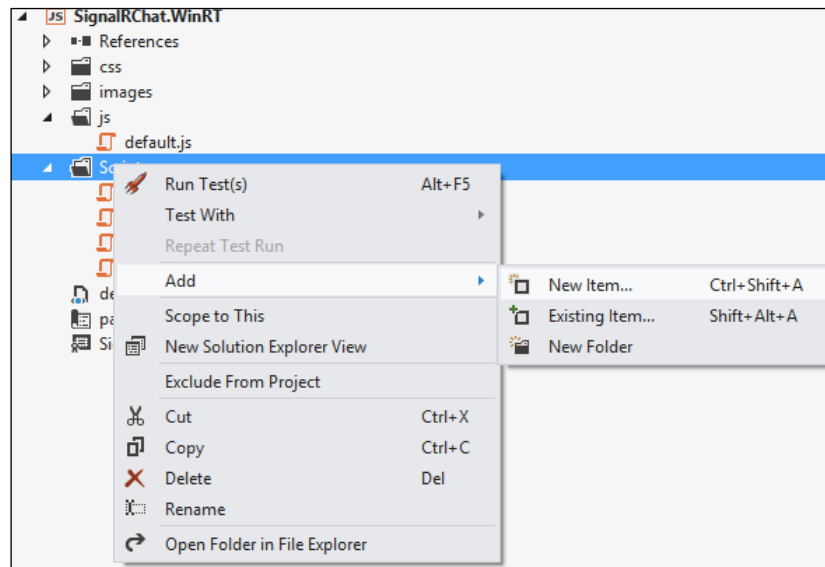
`http://localhost:1599/signalr/hubs` (Change the port if yours is different)



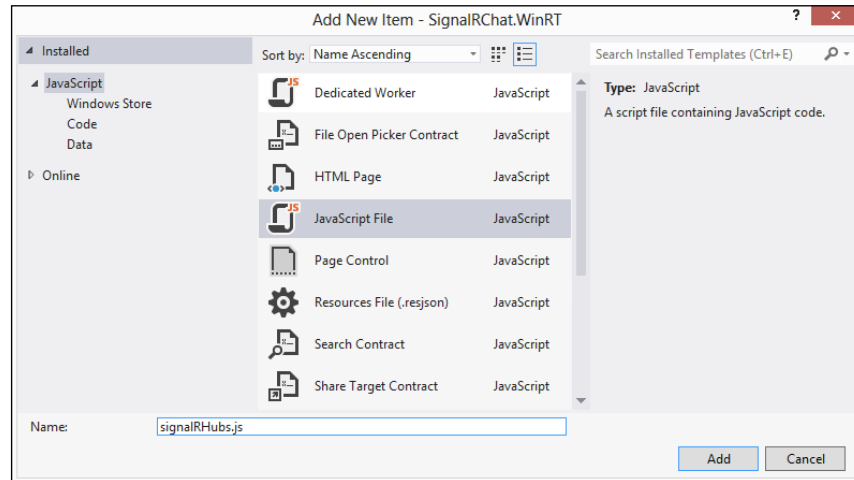
The screenshot shows a web browser window with the address bar displaying `localhost:1599/signalr/hubs`. The page content is the SignalR JavaScript Library v1.0.0, which includes a comment block with the library name, version, and license information, followed by a script tag for jQuery and a function definition for the SignalR client.

```
/*!  
 * ASP.NET SignalR JavaScript Library v1.0.0  
 * http://signalr.net/  
 *  
 * Copyright Microsoft Open Technologies, Inc. All rights reserved.  
 * Licensed under the Apache 2.0  
 * https://github.com/SignalR/SignalR/blob/master/LICENSE.md  
 */  
  
/// <reference path="../../../SignalR.Client.JS/Scripts/jquery-1.6.4.js" />  
/// <reference path="jquery.signalR.js" />  
(function ($, window) {  
    /// <param name="$" type="jQuery" />  
    "use strict";
```

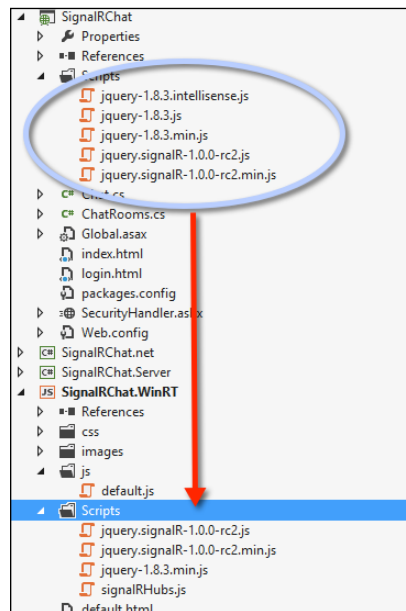
6. Select all the text and copy it to the clipboard for now.
7. Back in Visual Studio, we're going to add a script file for the proxy. Make sure that you have a Scripts folder in the WinRT project and right-click on it, and go to **Add | New Item**:



8. We're creating a JavaScript file and call it `signalRHubs.js`; click on **Add**:



9. In the new file, paste the code you copied from the browser and save the file, and then close it.
10. The next thing we're going to need are the jQuery scripts and the SignalR scripts. Copy them from the `SignalRChat` project into the new project by selecting them and dragging them down to the `Scripts` folder in the new project:



11. We need to provide references to the libraries we just copied into the project in the `<head/>` element in the `index.html` file:

```
<head>
  <meta charset="utf-8" />
  <title>SignalRChat.WinRT</title>

  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

  <script src="Scripts/jquery-1.8.3.min.js"></script>
  <script src="Scripts/jquery.signalR-1.0.0-rc2.min.js"></script>
  <script src="Scripts/signalRHubs.js"></script>

  <!-- SignalRChat.WinRT references -->
  <link href="/css/default.css" rel="stylesheet" />
  <script src="/js/default.js"></script>
</head>
```

12. Now it's time to bring the code from the web app into the WinRT project. Open the `index.html` file and locate the `<script/>` tag with our code for the application in it. We will start by taking the loose functions we had, and putting them into the app directly. Of course, there are ways of being able to reuse code across these projects, but we're not going to do that; we will be just settling for copy/paste for now.
13. Open the `default.js` file located in the JS folder of the WinRT project and paste in the loose functions from the `index.html` file directly under `use strict`:

```
// For an introduction to the Blank template, see the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
    "use strict";

    function addChatRoomToList(chatRoom) {
        $("#chatRoomsList").append("<option value=" + chatRoom + ">" + chatRoom + "</option>");
    }

    function addMessageToChatWindow(message) {
        $("#chatWindow").val($("#chatWindow").val() + message + "\n");
    }

    function clearChatRoomForRoomChange(chatRoom) {
        $("#chatWindow").val("");
        addMessageToChatWindow("Welcome to " + chatRoom);
    }
}
```

14. Now we're going to take the code that sits inside the `$(function() {` construct in `index.html`, and make that a new function right after the `clearChatRoomForRoomChange()` function that you just pasted in.
15. The function is going to be called `setupChat()` and should look like the following screenshot after you've pasted the content into it:

```
function setupChat() {
    var chat = $.connection.chat;
    chat.client.addChatRoom = function (chatRoom) {
        var currentChatRoom = chat.state.currentChatRoom;
        addChatRoomToList(chatRoom);
        if (chatRoom == currentChatRoom) {
            $("#chatRoomsList").val(currentChatRoom);
            addMessageToChatWindow("Welcome to " + currentChatRoom);
        }
    }

    chat.client.addMessage = function (room, message) {
        if (room === chat.state.currentChatRoom) {
            addMessageToChatWindow(message);
        }
    }

    $.connection.hub.url = "http://localhost:8080/signalr";
    $.connection.hub.start().done(function () {
        $("#chatWindow").val("Connected\n");


        chat.state.currentChatRoom = "Lobby";
        chat.server.join("Lobby").fail(function (e) {
            addMessageToChatWindow(e);
        });
        $("#sendButton").click(function () {
            chat.server.send($("#messageTextBox").val());
            $("#messageTextBox").val("")
        });

        $("#createChatRoomButton").click(function () {
            chat.server.createChatRoom($("#chatRoomTextBox").val());
            $("#chatRoomTextBox").val("")
        });

        $("#chatRoomsList").change(function () {
            var currentChatRoom = $("#chatRoomsList option:selected").val();
            chat.state.currentChatRoom = currentChatRoom;
            chat.server.join(currentChatRoom);
            clearChatRoomForRoomChange(currentChatRoom);
        });
    }).fail(function (e) {
        var dialog = new Windows.UI.Popups.MessageDialog("Couldn't connect");
        dialog.showAsync();
    });
}
```

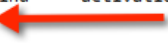
It is important to take notice of the URL that is being connected to. It is the self hosted one. This can easily be changed to go to any other URL, the web version for instance.

```
$.connection.hub.url = "http://localhost:8080/signalr";  
$.connection.hub.start().done(function () {  
    $("#chatWindow").val("Connected\n");
```



16. Now we need to call the `setupChat()` function at the appropriate time, which is after the application has been notified that it has been activated:

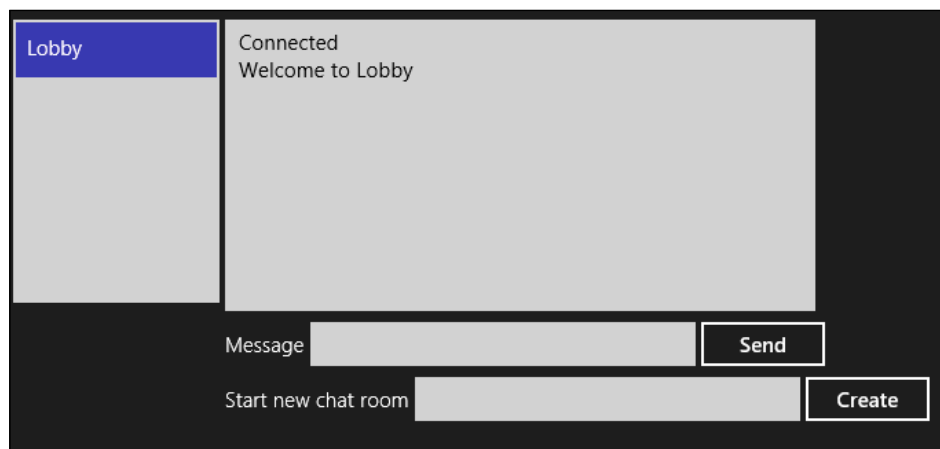
```
app.onactivated = function (args) {  
    if (args.detail.kind === activation.ActivationKind.launch) {  
        setupChat();  
  
        if (args.detail.previousExecutionState !== activation.ApplicationExecutionState.terminated) {  
            // TODO: This application has been newly launched. Initialize  
            // your application here.  
        } else {  
            // TODO: This application has been reactivated from suspension.  
            // Restore application state here.  
        }  
        args.setPromise(WinJS.UI.processAll());  
    }  
};
```



17. The grand finalé—the HTML. Copy the content of the body from `index.html` into the body of `default.html`:

```
<table>  
  <tr style="vertical-align:top;">  
    <td>  
      <select id="chatRoomsList" size="2" style="width:150px; height:206px;">  
      </select>  
    </td>  
    <td>  
      <textarea id="chatWindow" style="width:400px; height:200px;"></textarea>  
      <div>  
        <label>Message</label>  
        <input id="messageTextBox" type="text" />  
        <button id="sendButton">Send</button>  
      </div>  
      <div>  
        <label>Start new chat room</label>  
        <input id="chatRoomTextBox" type="text" />  
        <button id="createChatRoomButton">Create</button>  
      </div>  
    </td>  
  </tr>  
</table>
```

18. Running your app should now yield the result as shown in the screenshot that follows, and you now have a WinRT client of the chat:



Although this was a fairly short path to the goal of having a WinRT app running, you probably want to look in more depth at what the best practices of making WinRT apps are when it comes to look and feel. Even though it is possible to use standard HTML tags, Microsoft has provided a set of controls that feel more native to the Windows 8 platform, and should enhance the experience of your app.

Summary

With this chapter, we've managed to create a Microsoft Windows 8 Store based app for WinRT in very little effort by reusing most of our knowledge from HTML and JavaScript development. SignalR worked pretty much out of the box with the exception of the Hub proxy generation, which we could just get directly, and embed in the application.

All in all, you should now be able to build SignalR apps for different clients and servers, apply message filtering, security, and also make them scale out across multiple servers.

Index

Symbols

<authorization> tag 60
<compilation> tag 60
.MapHubs() line 36
<runtime> tag 60
<system.webserver> tag 61
<system.web> tag 60
.UseRedis() extension method 77

A

Access Key 78
addChatRoom event 49
addMessage() 37
All property 35
Application_Start method 20
Authenticate() method 69
AuthenticateUser method 62
Authorize() attribute 67
Azure 78, 80

C

Caller property 44
chat applications 41
ChatConnection class 21, 22, 26
clearChatRoomForRoomChange()
 function 105
client property 35, 37, 44
client state
 on server 53-57
Comet
 about 13
 URL 13
Connected string 25
connectionstring value 74, 79, 80

Console Application template 28
CookieContainer class 69
CreateChatRoom() method 67
currentRoom variable 56

E

events
 about 13
 server-sent events, URL 13
example code
 URL, for downloading 21

F

Fiddler
 about 84
 URL, for downloading 83
FormsAuthentication cookie 62

G

GetWebRequest() method 69
Global Application Class (Global.asax) 20
Global.asax.cs file 67
groups
 about 41, 43
 creation, application modified for 41-51
Groups property 57

H

HTTP
 about 12
 URL 12
HttpContext class 67
Hub 33-40

HubConnection class 39, 70
HubMethodName() attribute 35
HubName() attribute 35
Hypertext Transfer Protocol. *See* HTTP

I

include prerelease option
 messaging 77
Instances of selected object filter list 87
Invoke() method 39

M

Main() method 29, 30, 69
Manage NuGet Packages window 19
Massachusetts Institute of Technology (MIT) 9
messaging 71
Microsoft.ASP.NET.SignalR.Owin
 package 94

N

NuGet package
 adding, for Redis 76

O

OnConnected() method 44
On() method 39
OnReceived method 26
OWIN 91

P

performance counters 85-88
persistent connection
 about 17
 Application_Start method 20
 ChatConnection class 22
 code, adding to ChatConnection class 26
 Console Application template 28
 Global Application Class (Global.asax) 20
 jQuery, adding 23
 Main() method 30
 Manage NuGet Packages window 19
 NuGet installing, URL 18

 OnReceive method 26
 SignalR client library 28
PersistentConnection 17, 33
ProcessRequest() 63

R

RDP 10
Redis
 about 76-78
 Github, URL 76
 NuGet package, adding 76
Remote Desktop Protocol. *See* RDP
Remote Procedure Call (RPC) 33
RequireOutgoing property 67
Rich clients 11

S

security
 <authorization> tag 60
 <compilation> tag 60
 <runtime> tag 60
 <system.webserver> tag 61
 <system.web> tag 60
 about 59
 Authenticate() method 69
 AuthenticateUser method 62
 Authorize() attribute 67
 CookieContainer class 69
 CreateChatRoom() method 67
 FormsAuthentication cookie 62
 GetWebRequest() method 69
 Global.asax.cs file 67
 HttpContext class 67
 HubConnection class 70
 Main() method 69
 ProcessRequest() 63
 RequireOutgoing property 67
 SecurityHandler.ashx file 62
 SecurityHandler class 64, 65
 WebClient class 69
 web.config file 61
 Web.config file, modifying 60
SecurityHandler.ashx file 62
SecurityHandler class 64, 65
SecurityInspectionHandler class 96

self hosting

- Microsoft.ASP.NET.SignalR.Owin package 94
- NuGet packages 93
- SecurityInspectionHandler class 96
- start() function 97
- Startup class 96
- steps 91-97

send() function 55

Service Broker 74

setupChat() function 105

SignalR

- Fiddler 83, 84
- messaging 71
- performance counters 85-88
- security 59
- WinJS and Windows 8 99-107

software development

- Whys, URL 14

SQL Server

- about 72
- SignalR source, building 72

start() function 97

Startup class 96

stateful

- steps 53-57

T

terminal 8

Terminal Server Edition 10

Think Different phrase 14

U

user

- workstation 10

UseSqlServer() extension method 74

W

web application

- modifying, to create groups 41-51

web browser 12

WebClient class 69

web.config file 61

Whys

- URL 14

Windows 8

- and WinJS 99-107

WinJS

- about 99
- and Windows 8 99-107

X

X11. See X Window System

XHR 12

XMLHttpRequest. See XHR

X Window System 9

Z

ZIP button 72



Thank you for buying **SignalR: Real-time Application Development**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

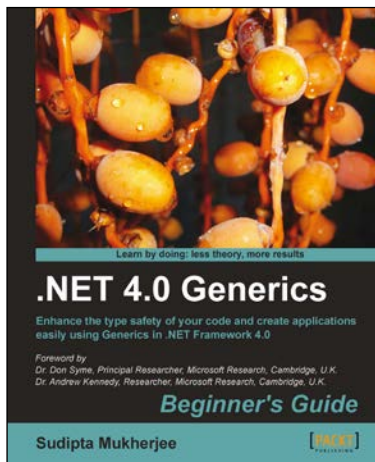
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



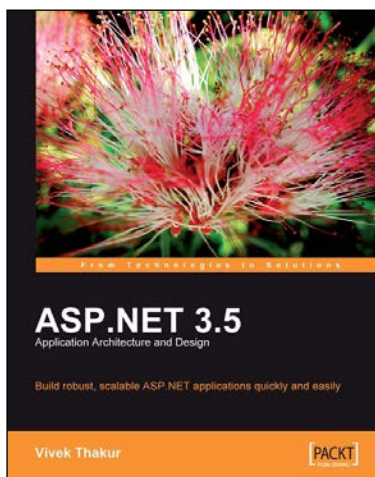
.NET 4.0 Generics Beginner's Guide

ISBN: 978-1-84969-078-2

Paperback: 396 pages

Enhance the type safety of your code and create applications easily using Generics in .NET Framework 4.0

1. Learn how to use Generics' methods and generic collections to solve complicated problems.
2. Develop real-world applications using Generics
3. Know the importance of each generic collection and Generic class and use them as per your requirements



ASP.NET 3.5 Application Architecture and Design

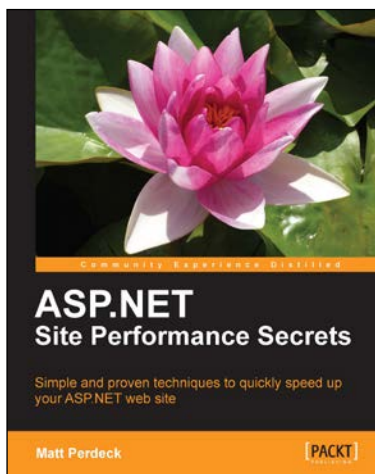
ISBN: 978-1-84719-550-0

Paperback: 260 pages

Build robust, scalable ASP .NET applications quickly and easily

1. Master the architectural options in ASP.NET to enhance your applications
2. Develop and implement n-tier architecture to allow you to modify a component without disturbing the next one
3. Design scalable and maintainable web applications rapidly
4. Implement ASP.NET MVC framework to manage various components independently

Please check www.PacktPub.com for information on our titles



ASP.NET Site Performance Secrets

ISBN: 978-1-84969-068-3

Paperback: 456 pages

Simple and proven techniques to quickly speed up your ASP.NET web site

1. Speed up your ASP.NET website by identifying performance bottlenecks that hold back your site's performance and fixing them
2. Tips and tricks for writing faster code and pinpointing those areas in the code that matter most, thus saving time and energy
3. Drastically reduce page load times



Microsoft SQL Azure Enterprise Application Development

ISBN: 978-1-84968-080-6

Paperback: 420 pages

Build enterprise-ready applications and projects with SQL Azure

1. Develop large scale enterprise applications using Microsoft SQL Azure
2. Understand how to use the various third party programs such as DB Artisan, RedGate, ToadSoft etc developed for SQL Azure
3. Master the exhaustive Data migration and Data Synchronization aspects of SQL Azure.
4. Includes SQL Azure projects in incubation and more recent developments including all 2010 updates

Please check www.PacktPub.com for information on our titles